# Quantitative program reasoning with graded modal types

DOMINIC ORCHARD, University of Kent, United Kingdom
VILEM-BENJAMIN LIEPELT, University of Kent, United Kingdom
HARLEY EADES III, Augusta University, USA

In programming, data is often considered to be infinitely copiable, arbitrarily discardable, and universally unconstrained. However this view is naïve: some data encapsulates resources that are subject to protocols (e.g., file and device handles, channels); some data should not be arbitrarily copied or communicated (e.g., private data). Linear types provide a partial remedy by delineating data in two camps: "resources" to be used but never copied or discarded, and unconstrained values. However, this binary distinction is too coarse-grained. Instead, we propose the general notion of *graded modal types*, which in combination with linear and indexed types, provides an expressive type theory for enforcing fine-grained resource-like properties of data. We present an advanced type system drawing together these aspects (linear, graded, and indexed) embodied in a fully-fledged functional language implementation, called Granule. We detail the type system, including its metatheoretic properties, and explore examples in the concrete language. This work advances the wider goal of expanding the reach of type systems to capture and verify a broader set of program properties.

## 1 INTRODUCTION

Static type systems are often described as one of the world's most widely applied verification techniques [Jhala and Majumdar 2009]. One possible reason is that they pair well with modern continuous development practises, exhibiting the desirable characteristics of *continuous verification* [O'Hearn 2018]: they (1) provide rapid feedback (2) give comprehensive feedback that is easily related to code (sometimes!) and (3) scale to large code bases. Subsequently, various approaches in the literature seek to expand the remit of types to encompass a broader set of program properties at the type-level, e.g., refinement types [Denney 1998; Freeman and Pfenning 1991; Vazou et al. 2014], dependent types [Hanna et al. 1990; Martin-Löf and Sambin 1984], deductive-verification-style types [Nanevski et al. 2008], amongst many others. In this paper, we seek to further increase the power of types by providing type-based quantitative analysis of data usage in programs.

Most programming languages (and type systems) take the view that data is unrestricted: it can be replicated, discarded, and manipulated without constraint. However, this view leads to many possible software errors. Some data is subject to confidentiality requirements and therefore should not be copied arbitrarily. Some data acts a proxy for an external resource (e.g., a hardware device or a file) and therefore is sensitive to the order of operations applied to it. Some data is subject to logical or temporal restrictions, e.g., during metaprogramming. Dually, some programs are sensitive to the size and nature of their inputs, with non functional-requirements dependent on their data. Thus, the reality is that (some) data acts *as a resource*, subject to constraints. Such ideas appear for example in recent work on Rust [Matsakis and Klock II 2014], incorporating a static view of *ownership* and *borrowing* of references, acknowledging that some data should not just be copied and propagated arbitrarily. This work aims to directly embed the notion of data as a resource into a type system in a way that can be specialised to different notions of resource and dataflow property. We

Authors' addresses: Dominic Orchard, School of Computing, University of Kent, Canterbury, Kent, CT2 7NF, United Kingdom, D.A.Orchard@kent.ac.uk; Vilem-Benjamin Liepelt, School of Computing, University of Kent, Canterbury, Kent, CT2 7NF, United Kingdom, v.liepelt@kent.ac.uk; Harley Eades III, School of Computer and Cyber Sciences, Augusta University, Augusta, Georgia, USA, HEADES@augusta.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

show that an expressive system is provided by combining linear types, indexed types (lightweight dependent types) and the more recent notion of *graded modal types*.

Linear types, in their strictest sense, treat data like a physical resource which must be used once, and then never again [Girard 1987; Wadler 1990; Walker 2005]. For example, we can type the identity function, since it binds a variable then uses it, whereas the K combinator (building a constant function) $\lambda x.\lambda y.x$ is not linearly typed as $y$ is never used. This is rather restrictive, so linear logic also provides a modal operator, usually written as !, which captures and tracks non-linear, unconstrained values. This provides a binary view: either values are linear (like a resource) or non-linear (like the traditional view of data). However, in programming, non-linearity rapidly permeates a program. Bounded Linear Logic (BLL) instead provides a more fine-grained view by replacing ! with an indexed family of modal operators, indexed by terms describing the upper bound on usage [Girard et al. 1992], e.g. $!_2$ describes values that can be used at most twice. The proof rules then manipulate these indices, accounting for contraction, weakening, and composition.

Various recent work has generalised the approach of BLL, providing a family of modalities whose indices are drawn from an arbitrary semiring, thereby allowing various different program properties to be tracked in a single system, e.g., bounded reuse, strictness, data constructor use, sensitivity, and scheduling constraints for hardware synthesis [Brunel et al. 2014; Ghica and Smith 2014; Petricek et al. 2013]. The indices to ! are often described as *coeffects*, capturing how a program depends on its variables and what it does with them.[1] Semantically, these semiring-indexed modalities are captured by *graded exponential comonads* [Gaboardi et al. 2016a].

In this work, we propose the general terminology of *graded modal types* to capture notions of semiring-indexed ! modalities as well *graded monads* [Katsumata 2014; Orchard et al. 2014; Smirnov 2008]. Graded monads generalise monads (the Curry-Howard counterpart to a possibility modality [Benton et al. 1998]) to a monoid-indexed form, capturing effect-system like reasoning. In general, a graded modality provides an indexed family of operators with structure over the indices, witnessing proof/type rules. Through the Curry-Howard lens, a graded modal type system then carries information about the structure of programs, and in concert with a suitably expressive type system, provides a mechanism for specifying and verifying properties of programs not captured by existing type systems (even with, say, dependent types).

We develop this idea, presenting a type system that takes linear types and indexed types as the basis. Indexed types provide lightweight dependent types for capturing dependencies between values. On top of this we integrate graded modalities in two flavours: graded comonads (graded necessity) and graded monads (graded possibility). We focus mainly on the graded comonadic part, which is heavily integrated with linearity. We refer to their combination as *graded linearity* (by analogy with *bounded linearity* in BLL). We make the following contributions:

- We present Granule: an eager functional language in the tradition of Haskell/ML which is the first language to integrate linear, graded, and indexed types. We demonstrate its power through various examples (Section 2 and 8). We address interesting questions about interactions between grading and standard language features, e.g. pattern matching.
- We define a novel graded modal type theory that integrates previous work on coeffects/graded comonads, with multiple modalities in one system, and polymorphic and indexed types (Section 3 and 4). We provide an operational model (Section 6) and meta-theoretic results (Section 7).
- We provide a bidirectional type checking algorithm for our system (Section 5) which is at the heart of our implementation. This involves interaction between standard type-checking techniques and an SMT solver, discharging theorems from equations over the indices of graded modalities.

[1] The *coeffect* terminology is by duality to *effect* systems which describe how a program changes its environment.

The concluding sections of this paper discuss related work in more detail and then the next steps in taking this work forward to provide a new generation of resource-aware programming languages.

## 2 A TASTE OF GRANULE

We start with an *amuse-bouche* in the form of a literate Granule program to whet our appetite for—and strengthen our intuitions about—linear types and graded modalities. We will see examples where graded linearity allows us to document, discover, and enforce program properties, complementing and extending the reasoning we get from parametric polymorphism and indexed types.

Syntactically, Granule is cognate with Haskell and ML. Programs comprise mutually recursive definitions with type signatures. Functions are defined as a sequence of equations, using pattern matching to distinguish their cases. The TEX source of this document contains two embedded Granule files—one for well typed examples, the other, marked by ✗, for the ill typed; everything here is real code. We invite the reader to run the type checker and interpreter themselves to benefit from an interactive experience where code can be experimented with.[2]

### 2.1 Linearity

To ease into the syntax, the following are two well-typed polymorphic functions in Granule:

```
id : ∀ {t : Type} . t → t
id x = x
```

```
swap : ∀ {t : Type, s : Type} . (t, s) → (s, t)
swap (x,y) = (y,x)
```

Polymorphic type variables are explicit, given with their kind. These functions are both linear: they use their inputs exactly once. The `id` function is the linear function *par excellence* and `swap` just switches around its arguments in a pair. From `swap` we can deduce that the structural *exchange* rule is allowed. However, the other two structural rules, *weakening* and *contraction*, are not available by default, as witnessed by the following two functions being rejected:

✗
```
drop : ∀ {t : Type} . t → ()
drop x = ()
```

✗
```
copy : ∀ {t : Type} . t → (t, t)
copy x = (x,x)
```

The Granule interpreter `gr` gives us the following errors, respectively:

```
Linearity error: 2:1:
Linear variable x is never used.
```

```
Linearity error: 2:10:
Linear variable x is used more than once.
```

By universally quantifying over the type of the inputs, we are claiming that for any type `t` we can annihilate or copy its values. If our types include resources (things that cannot be dropped or copied arbitrarily), then this statement cannot hold. In order to reason about—and program with—resource-like data, we must renounce *thoughtless* `dropping` and `copying`. On the surface, this seems like an extreme restriction, but we have already regained considerable power: linearity aids in providing safe interfaces to stateful interactions with the harsh world outside of our own runtime. A well-known example is that of file handles, which must always be closed and never used thereafter [Walker 2005]. We can easily support this in Granule via its linear types.

Granule provides an interface to files, which includes the following operations:

```
openHandle  : ∀ {m : HandleType} . IOMode m → String → (Handle m) <IO>
readChar    : Handle R → (Handle R, Char) <IO>
closeHandle : ∀ {m : HandleType} . Handle m → () <IO>
```

[2]The `gr` toolchain can be downloaded from (URL redacted). To see type checker output for ill-typed examples pass `gr` the name of the environment: `--literate-env-name grill` (**gr**anule **ill**-typed). The frontend accepts both Unicode symbols and their ASCII counterparts, e.g. `forall` for ∀ and `->` for →. The documentation contains a full table of equivalences.

The types employ polymorphism over `HandleType`, an algebraic data type promoted to the type-level for statically enforcing modes. The `openHandle` function creates a handle, and its dual `closeHandle` destroys a handle. Linearity means we can never *not* close a handle: we must use `closeHandle` to erase it. The `readChar` function takes a readable handle and returns a pair of a readable handle and a character. Logically, `readChar` can be thought of as consuming and producing a handle, though at runtime these are really the same handle. The `<IO>` type is a modality, written postfix, which captures I/O side effects akin to Haskell's `IO` monad [Jones 2003]. We explain `<IO>` more later as it approximates a more fine-grained graded modality. We now give two programs, using Granule's notation for sequencing effectful computations akin to Haskell "do" notation:

```
twoChars : (Char, Char) <IO>
twoChars = let
  h ← openHandle ReadMode "somefile";
  (h, c1) ← readChar h;
  (h, c2) ← readChar h;
  () ← closeHandle h
  in pure (c1, c2)
```

```
bad : Char <IO>
bad = let
  h1 ← openHandle ReadMode "somefile";
  h2 ← openHandle ReadMode "another";
  () ← closeHandle h1;
  (h1, c) ← readChar h1
  in pure c
```

On the left, `twoChars` opens a handle, reads two characters from it and closes it, returning the two characters in an I/O context. The **`pure`** function here lifts a pure value into the `<IO>` type (akin to `return` in Haskell). On the right, `bad` opens two handles, then closes the first, reads from it, and returns the resulting character without closing the second handle. This program gets rejected with several linearity errors: $h_1$ is used more than once and $h_2$ is discarded. Thus, a lack of weakening and contraction already readily supports safe programming against stateful protocols.

However, many programs require discarding and copying. We could adopt a binary view, sorting objects into a kind of unrestricted types and a kind of linear types. Then we could type `drop` and `copy` by quantifying over all `t` of the kind 'unrestricted types'. Separation of these two worlds comes at a cost however: either definitions are kind monomorphic, meaning they are inflexible and not reusable, as in the system of Wadler [1990], or we have to wrestle with a system of linearity-kind polymorphism to achieve reuse as in Quill [Morris 2016] or System F$^\circ$ [Mazurak et al. 2010]. Instead, Granule has just one (linear) kind of types with graded modalities for propagating *requirements* and *capabilities* for non-linearity in a controlled way. We choose to be restrictive by default and use modalities to relax resource requirements, just as in linear logic [Girard 1987]. This strict-as-default view fits the rule that *the more polymorphic our inputs, the less we can assume about them*, which strengthens our type-based reasoning in a rather pleasing way, as we will discover in this section.

We can thus rewrite the ill-typed `drop` and `copy` into less polymorphic, well-typed versions using a necessity-like modality in Granule *à la* linear logic:

```
drop' : ∀ {t : Type} . t [] → ()
drop' [x] = ()
```

```
copy' : ∀ {t : Type} . t [] → (t, t)
copy' [x] = (x, x)
```

The postfix "box" constructor `[]` can be thought of as the equivalent of linear logic's ! for unrestricted use. Our syntax is an allusion to necessity (□) from modal logic. Since the parameters are now modal, of type `t []`, we can use an "unboxing" pattern to bind a variable of `x` of type `t`, which can now be discarded or copied freely in the bodies of the functions. Note that a value of type `t []` is itself still subject to linearity: it must be used. Whilst this modality provides us with a non-linear binding for `x`, it however gives a rather coarse-grained view: we cannot distinguish the different forms of non-linearity employed by `copy'` and `drop'`, which have the same type for their parameter. Instead, we achieve this distinction via *graded modalities*.

## 2.2 Graded modalities

To track fine-grained resource information, modalities in Granule are *graded* by elements of a *resource algebra* whose operations capture program structure. One built-in resource algebra counts variable use via the natural numbers semiring. This enables more precisely typed `copy` and `drop`:

```
drop'' : ∀ {a : Type} . a [0] → ()
drop'' [x] = ()
```

```
copy'' : ∀ {a : Type} . a [2] → (a, a)
copy'' [x] = (x, x)
```

The function definitions replay `drop'` and `copy'`, but the types now provide exact specifications of the amount of non-linearity: `0` and `2`. We will see various other graded modalities in due course.

In these examples, we know usage statically because there is no branching control flow that leads to different usages depending on function inputs. We introduce algebraic data types (ADTs) next to explore the interaction between graded linearity and familiar functional programming idioms.

## 2.3 Algebraic Data Types and graded linearity

The most remarkable thing about data type definitions in Granule is that there is nothing remarkable about them, for two reasons: (1) data constructors are already linear functions in the sense that each argument appears exactly once in the result; (2) values of all types are treated linearly. Consider the following data type declaration. From now on we omit unambiguous kinds in quantifications.

```
data Maybe t = None | Some t
```

The constructor `Some` is a linear function of type `∀ t. t → Maybe t`: if its constructed value is consumed exactly once, then its argument will be used exactly once. It is no more than a wrapper around existing data (similar to recent work on Linear Haskell [Bernardy et al. 2017]).

The difference between Granule and non-linear languages is more apparent when we use data types. Let's define a function that, given values of type `t` and `Maybe t`, returns a value of type `t`:

```
fromMaybe : ∀ t. t [0..1] → Maybe t → t
fromMaybe [_] (Some x) = x;
fromMaybe [d] None     = d
```

✗

```
fromMaybe' : ∀ t. t → Maybe t → t
fromMaybe' _ (Some x) = x;
fromMaybe' d None     = d
```

On the right, `fromMaybe'`, as we might define it in Haskell, is ill-typed since it discards the default value `d` in the first equation but does not declare nonlinear use. The type `∀ t. t → Maybe t → t` is in fact uninhabited in Granule. We cannot implement the function that always just returns the first argument unless we declare nonlinear use of the second argument. Thus graded linearity captures both intensional and extensional program properties.

On the left, `fromMaybe` is the well-typed version appearing in Granule's Standard Library. The type of its default value `d` is a graded modality with a resource algebra capturing both upper (affine) and lower (relevant) bounds of use as an interval combined via the constructor `_..._`. This gives us a more fine-grained analysis than just upper bounds as in BLL. One of Granule's core features is the composability of analyses for fine-grained tracking in the presence of control-flow (Section 4.2).

## 2.4 Indexed Types

Indexed types give us type-level access to further information about data. Granule supports user-defined indexed types, in a similar style to Haskell's GADTs [Peyton Jones et al. 2006]. We use the well-known example of size-indexed lists (`Vec`) and indexed naturals, but in a novel way.

```
data Vec (n : Nat) (a : Type) where
  Nil : Vec 0 a;
  Cons : a → Vec n a → Vec (n + 1) a
```

```
data N (n : Nat) where
  Z : N 0;
  S : N n → N (n + 1)
```

When defining operations over `Vec`, we notice that some functions have the usual type signatures

we know from non-linear languages, such as append:

```
append : ∀ {t : Type, n : Nat, m : Nat} . Vec n t → Vec m t → Vec (n + m) t
append Nil ys = ys;
append (Cons x xs) ys = Cons x (append xs ys)
```

Indexed types give us the useful property that the length of the output list is indeed the sum of the length of the inputs. But in a linear language this type guarantees even more: *every element from the inputs must appear in the output.* In a nonlinear setting, the implementation of this type could drop and copy values, as long as the output has the correct length.

The types of functions which are nonlinear in the elements look different to their usual counterparts. For example when taking the length of a list, we do not consume its elements. Either we must discard elements as on the left, or we must reconstruct the list and return it, on the right:

```
length : ∀ t, n. Vec n (t [0]) → N n
length Nil = Z;
length (Cons [_] xs) = S (length xs)
```

```
length' : ∀ t, n. Vec n t → (N n, Vec n t)
length' Nil = (Z, Nil);
length' (Cons x xs) =
  let (n, xs) = length' xs in (S n, Cons x xs)
```

Polymorphism is important when reasoning about linearity: when we can pattern match on the concrete values, then we can consume them. We can also "produce" values out of nowhere in the case of constants like `Nil`, `None`, `True`, `42` etc. This allows us on the right-hand side to produce the indexed natural: we deconstruct the list until we reach `Nil`, where we return `Z`, which we then bind to the variable `n` in the inductive case where we reconstruct the list.

Scope also determines whether we can construct or deconstruct values, e.g. we can build `N n` values because its data constructors are in scope, i.e., it is not an *abstract* type. This is familiar from most programming languages, but has further implications in Granule. Consider the file handles example: we aren't able to simply forget values of the abstract data type `Handle` by consuming them via pattern matching; we have to use the interface provided to create and destroy them.

Bringing everything together, we now define a function to left pad a string to a specified length. We begin by implementing a few combinators:

```
rep : ∀ n t. N n → t [n] → Vec n t
rep Z [c] = Nil;
rep (S n) [c] = Cons c (rep n [c])
```

```
sub : ∀ m n. {m ⩾ n} ⇒ N m → N n → N (m - n)
sub m Z = m;
sub (S m') (S n') = sub m' n'
```

The function `rep` takes a number `n` and a value `t`, returning a `Vec n t`. This cannot be linear in `t` since it must make `n` copies of `t`. Using indexing as lightweight dependent types, we specify that the non-linearity of the parameter depends on the size of the output list.

The `sub` function defines subtraction on our indexed naturals. Granule lets us give preconditions in the context of type schemes (before ⇒). These must hold where the function is used. Such predicates are discharged by the external solver. If we include the case "`sub Z (S n') = sub Z n'`" which violates the precondition `m ⩾ n`, then `gr` gives us the following error:

```
Impossible pattern match: 3:1:
Pattern guard for equation of sub is impossible.
Its condition 0 > n0.11 + 1 is unsatisfiable.
```

Finally, we can put the above functions together and define our left pad function:[3]

```
leftPad : ∀ {t : Type, m n : Nat} . {m ⩾ n} ⇒ t [m - n] → N m → Vec n t → Vec m t
leftPad [c] n str = let (m, str) = length' str in append (rep (sub n m) [c]) str
```

The type says that given a target length `m` and an input list with a lesser or equal length `n`, we consume some padding element of type `t` exactly `m - n` times to produce an output list of the target length `m`. In Granule this type alone implies the correct implementation—modulo reorderings and nontermination—via:

(1) *Parametric polymorphism*: ensuring that the implementation cannot depend on the concrete padding items provided or the items of the input list (hence we use lists instead of strings);
(2) *Indexed types*: ensuring the correct size and explaining the exact usage of the padding element;
(3) *Graded linearity*: ensuring that every item in the input list appears exactly once in the output. The type `t [m - n]` of the padding element reveals its exact usage.

The type of `leftPad` in Granule is superficially similar to what we could write in GHC Haskell or a fully dependently-typed language, except for the nonlinearity of the padding element, a minor syntactic addition. However the extra guarantees we get in a graded linear system like Granule's means we get properties for free which we would otherwise have to prove ourselves.

### 2.5 Other graded modalities

In the file handles example we swept past the `<IO>` type constructor. This is an example of an effect-capturing modality (the "diamond" constructor alludes to modal possibility), in the spirit of Haskell's `IO` monad. However, Granule provides *graded monads* [Katsumata 2014], which can give a more fine-grained account of effects. The more precise type signature for `twoChars` is actually `twoChars : (Char, Char) <{Open,Read,IOExcept,Close}>` which tells us its range of possible side effects via a set of labels, and notably that there are no `Write` effects. Thus, Granule provides graded modalities in two flavours: graded necessity/comonads for coeffects (properties of input variables) and graded possibility/monads for effects (properties of output computations).

A further graded modality that we have not seen yet provides a notion of *information-flow security* via a lattice-indexed graded necessity with labels `Public` and `Private`. We can then, for example, define programs like the following which capture the security level of values, and how levels are preserved (or not) by functions:

```
secret : Int [Private]
secret = [1234]

hash : ∀ {l : Level} . Int [l] → Int [l]
hash [x] = [x*x*x]
```

✗
```
main : Int [Public]
main = hash secret
```

✓
```
main : Int [Private]
main = hash secret
```

Section 8 shows more examples, including combining security levels and variable usage. Now that we have a taste for Granule, we set out the type system that enables all of these examples. Section 3 describes a core simply-typed graded linear basis first, before Section 4 defines the full system.

## 3 THE CORE SIMPLY-TYPED GRADED LINEAR CALCULUS

To aid understanding, we first establish a subset of the Granule language, called GrMini, which comprises the linear lambda calculus extended with a graded comonad, resembling the coeffect calculi of Brunel et al. [2014] and Gaboardi et al. [2016b]. Section 4 extends GrMini to Granule

[3] A runnable Granule program that uses this to pad a user given input to the user given length, both read from `stdin`, can be found here: (URL redacted) This is also instructive for its use of existential types in Granule.

core (Gr) with polymorphism, indexed types, multiple different graded modalities, and pattern matching. The typing rules of GrMini are shown later to be specialisations of Gr's typing rules.

Types and terms of GrMini are those of the linear-$\lambda$ calculus with two additional piece of syntax for introducing and eliminating values of the graded necessity type $\Box_r A$:

$$t ::= x \mid t_1\ t_2 \mid \lambda x.t \mid [t] \mid \mathbf{let}\,[x] = t_1\ \mathbf{in}\ t_2 \qquad A, B ::= A \multimap B \mid \Box_r A \qquad (\textit{terms and types})$$

The usual syntax of the $\lambda$-calculus, with variables $x$, is extended with the term-former $[t]$ which promotes a term to be non-linear, typed by the graded modal constructor $\Box_r A$, as shall be seen in the typing rules. The term $\mathbf{let}\,[x] = t_1\ \mathbf{in}\ t_2$ dually provides elimination for graded comonad types. The graded comonadic modality $\Box_r A$ is an indexed family of type constructors whose indices $r$ range over the elements of a *resource algebra*—in this case, a semiring $(\mathcal{R}, +, 0, \cdot, 1)$—whose operations echo the structure of the typing/proof rules. This semiring parameterises GrMini as a meta-level entity. In contrast, Gr can select different resource algebras and thus modalities internally. Here we consider the usual natural numbers semiring as an example and useful aid to understanding.

Typing judgments are of the form $\Gamma \vdash t : A$ with typing contexts $\Gamma$ of the form:

$$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]_r \qquad (\textit{contexts})$$

Contexts are either empty $\emptyset$, or can be extended with a linear variable assumption $x : A$ or a *graded assumption* $x : [A]_r$. For a graded assumption, $x$ can behave non-linearly, with substructural behaviour captured by the semiring element $r$, which describes $x$'s use in a term.

Typing for the linear $\lambda$-calculus fragment is then given by the rules:

$$\frac{}{x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs} \qquad \frac{\Gamma_1 \vdash t_1 : A \multimap B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \textsc{app} \qquad \frac{\Gamma \vdash t : A}{\Gamma, [\Delta]_0 \vdash t : A}\ \textsc{weak}$$

Linear variables are typed in a singleton context, which enforces the behaviour that linear variables cannot be weakened. Abstraction and application are as expected, though application employs a partial context concatenation operation + defined as follows:

**Definition 3.1.** [Context concatenation] Two contexts can be concatenated if they contain disjoint sets of linear assumptions. Furthermore, graded assumptions appearing in both contexts are combined using the additive operation of the semiring $+$. Concatenation + is specified as follows:

$$(\Gamma, x : A) + \Gamma' = (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin |\Gamma'| \qquad \emptyset + \Gamma = \Gamma$$
$$\Gamma + (\Gamma', x : A) = (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin |\Gamma| \qquad \Gamma + \emptyset = \Gamma$$
$$(\Gamma, x : [A]_r) + (\Gamma', x : [A]_s) = (\Gamma + \Gamma'), x : [A]_{(r+s)}$$

Note that this is a declarative specification of + rather than an algorithmic definition, since graded assumptions for the same variable may appear in different positions within the two contexts.

The weak rule provides weakening only for graded assumptions, where $[\Delta]_0$ denotes a context containing only assumptions graded by 0. Context concatenation and weak thus provide contraction and weakening for graded assumptions using $+$ and 0 to witness substructural behaviour corresponding to a split in a dataflow path for a value or the end of a dataflow path. The exchange rule, allowing contexts to be re-ordered, is implicit here (though Section 10 discusses alternatives).

The next three rules employ the remaining semiring structure, typing the additional syntax as well as connecting linear assumptions to graded assumptions:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der} \qquad \frac{[\Gamma] \vdash t : A}{r \cdot [\Gamma] \vdash [t] : \Box_r A}\ \textsc{pr} \qquad \frac{\Gamma_1 \vdash t_1 : \Box_r A \quad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\ \mathbf{in}\ t_2 : B}\ \textsc{let}$$

*Dereliction* (der) converts a linear assumption to be graded, marked with 1. Subsequently, the semiring element 1 relates to linearity, though it does not exactly denote linear use as it is not

necessarily the case that $x : [A]_1$ implies $x : A$ for all semirings. *Promotion* (PR) introduces the graded comonadic type with grade $r$, propagating this grade to the assumptions via scalar multiplication of the context by $r$. In the case of tracking usage, the rule states that to produce the *capability* to reuse a result value $t$ of type $A$ exactly $r$-times requires that all the input requirements for $t$ are provided $r$-times over, hence we multiply the context by $r$.

**Definition 3.2.** [Scalar context multiplication] Assuming that a context contains only graded assumptions, denoted $[\Gamma]$ in typing rules, then $\Gamma$ can be multiplied by a semiring element $r \in R$:

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

The (LET) rule provides elimination for the graded comonad via a kind of substitution, where a graded value is "unboxed" and substituted into a graded assumption with matching grades. In the context of reuse, **let** plugs the capability to reuse a value $r$ times into the requirement of using a variable $r$ times. Since (LET) has two subterms, context addition is also employed in the conclusion.

If the grades are removed, or collapsed via a singleton semiring, then this system is that of intuitionistic natural deduction for S4 necessity [Bierman and de Paiva 2000; Pfenning and Davies 2001], but using Terui's technique of delineating modal assumptions via "discharged" (in our case "graded") assumptions to provide cut admissibility [Terui 2001]. We prove this property in Section 7.

Granule incorporates the GrMini syntax and rules. The graded modal operator is written postfix in Granule with the grade inside the box: i.e., $\Box_r A$ is written as `A [r]`. The following are then some simple Granule examples using just the GrMini subset:

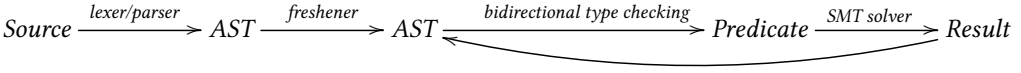
```
k : Int → Int [0] → Int              foo : Int [3] → Int [6] → Int [3]
k = λx → λyb → let [y] = yb in x     foo = λxb → λyb → let [x] = xb in [y] = yb in [x+y+y]
```

In `foo`, the promoted term uses `x` once and `y` twice. Thus, if we promote and require three copies of the result, as specified by the type signature, then we require 3 copies of `x` and 6 copies of `y` by application of the promotion rule, which propagates to the types shown for `xb` and `yb`.

Thus, GrMini provides the core of graded linearity (linearity with comonadic graded modalities) for Granule, capturing dataflow properties of variables (and thus values) via the information of a semiring. Gr develops this idea further to allow multiple different graded modalities within the language, as well as indexed data types, pattern matching, and polymorphism.

## 4 THE GRANULE TYPE SYSTEM

We introduce the core of Granule (Gr) by extending GrMini, reflecting the full power of the implemented language, but with some slight simplifications. The implementation of the Granule type system roughly has the following structure:




Bidirectional type checking, if succesful, outputs a predicate capturing additional theorems about grading, which is compiled to the SMT-LIB format [Barrett et al. 2010] and passed to any compatible solver (we typically use Z3 [De Moura and Bjørner 2008]). Type checking and predicate solving is carried out for each top-level definition independently, assuming that all other definitions are well-typed based on their signatures.

We first define the syntax (§4.1), excluding user-defined data types, then resource algebras and their instances (§4.2), before defining typing declaratively (§4.3) as an expansion of GrMini.

### 4.1 Syntax

A core subset of the surface-level syntax for Granule is given by the following grammar:

$$t ::= \underbrace{x \mid t_1\ t_2 \mid \lambda p.t}_{\lambda\text{-calculus}} \mid \underbrace{[t]}_{\text{box}} \mid \underbrace{C\ t_0 \ldots t_n \mid n}_{\text{constructors}} \mid \underbrace{\textbf{let}\ \langle p \rangle \leftarrow t_1\ \textbf{in}\ t_2 \mid \langle t \rangle}_{\text{monadic metalanguage}} \qquad \textit{(terms)}$$

$$p ::= \underbrace{x}_{\text{variables}} \mid \underbrace{\_}_{\text{wildcard}} \mid \underbrace{[p]}_{\text{unbox}} \mid \underbrace{n \mid C\ p_1 \ldots p_n}_{\text{constructors}} \qquad \textit{(patterns)}$$

where $\langle t \rangle$ is shorthand for (**pure** $t$). Unlike the $\lambda$-calculus and GrMini, $\lambda$ abstraction is over a pattern $p$ rather than just a variable. We include integer constructors $n$ and their corresponding patterns as well as data constructors $C$ with zero or more arguments. A built-in library provides operations on integers and other primitives, but we elide the details since they are routine.

The "boxing" (promotion) construct $[t]$ is dualised by the unboxing pattern $[p]$, replacing the specialised let-binding syntax of GrMini, which is now syntactic sugar:

$$\textbf{let}\ [p] = t_1\ \textbf{in}\ t_2 \ \triangleq\ (\lambda [p].t_2)\ t_1 \qquad \textit{(syntactic sugar)}$$

The syntax of GrMini types is extended and a syntactic category of kinds is also now included:

$$A, B, R ::= A \to B \mid K \mid \alpha \mid A\ B \mid A\ \mathsf{op}\ B \mid \Box_c A \mid \Diamond_\varepsilon A \mid () \qquad \textit{(types)}$$

$$\kappa ::= \mathsf{Type} \mid \mathsf{Coeffect} \mid \mathsf{Effect} \mid \mathsf{Predicate} \mid \kappa_1 \to \kappa_2 \mid \uparrow A \qquad \textit{(kinds)}$$

$$\mathsf{op} ::= + \mid * \mid - \mid < \mid > \mid = \mid \neq \mid \vee \mid \wedge \qquad \textit{(type operators)}$$

Types comprise function types, type constructors $K$, variables $\alpha$ (and $\beta$), application $A\,B$, binary operators $A\ \mathsf{op}\ B$, graded comonadic types $\Box_c A$, graded monadic types $\Diamond_\varepsilon A$, and the unit type $()$. Functions in Gr are linear, though we use the Cartesian function space notation $\to$ rather than the traditional $\multimap$ since we use -> (or $\to$) for a more familiar concrete sytnax. Kinds comprise a number of kind constants categorising types, grades (coeffect and effect) and predicates, along with a function space for higher-kinded types and the ability to lift a type $A$ to a kind, written $\uparrow A$.

We provide some built-in type constructors $K$ in the Granule implementation which is also extended by user-defined types (see Section 4.3.3).

$$K ::= \mathsf{Int} \mid \mathsf{Char} \mid () \mid \times \mid \mathsf{Nat} \mid \mathsf{Level} \mid \mathsf{Ext} \mid \mathsf{Interval} \qquad \textit{(type constructors)}$$

These built-in type constructors are separated by their kind (defined via the kinding judgement in Section 4.3) with the first three of kind Type, products $\times$ as kind-polymorphic, and the last four of kind Coeffect or higher-kinded, producing types of kind Coeffect.

Similarly to ML, we provide polymorphism via *type schemes* allowing type quantification only at the outer level of a type (rather than rank-N quantification, see Section 10 on future work):

$$T ::= \forall \overrightarrow{\alpha : \kappa}\ .\ A \mid \forall \overrightarrow{\alpha : \kappa}\ .\ \{A_1, .., A_n\} \Rightarrow B \qquad \textit{(type schemes)}$$

where $\overrightarrow{\alpha : \kappa}$ represents a comma-separated sequence of type variables and their kinds. The second syntactic form additionally includes a set of one or more predicates (types of the kind Predicate) which can be used to express theorems which need to be solved implicitly by the type checker (a kind of refinement), as seen in the `leftPad` example (Section 2.4).

Finally, top-level definitions provide a type-scheme signature for a definition along with a non-empty sequence of equations headed by patterns:

$$\mathit{Def} ::= x : T; \overrightarrow{x\ p_{i1} \ldots p_{in}\ =\ t_i} \qquad \textit{(definitions)}$$

### 4.2 Grading and resource algebras

GrMini was parameterised at the meta-level by a semiring, providing a system with one particular graded modality. Granule instead allows different graded modalities, with different index domains, to be used simultaneously within the same program. The graded modal type $\Box_c A$ captures different modalities, identified by the type of the grade $c$. In Granule, we describe the structure of $c$ as a *resource algebra*, which is a pre-ordered semiring $(\mathcal{R}, +, 0, \cdot, 1, \sqsubseteq)$ with two differences: the operations may be partial and the standard additive unit axiom $0 + r = r$ is replaced with $0 + r \sqsubseteq r$. An example of where this is useful is seen below for security levels. We color the operations in blue when we are referring to a general resource algebra structure.

The syntax of grading terms $c$ for *coeffects* is given by:

$$\begin{aligned} c ::= \alpha \mid c_1 + c_2 \mid c_1 \cdot c_2 \mid 0 \mid 1 \mid c_1 \sqcup c_2 \mid c_1 \sqcap c_2 \mid \mathsf{flatten}(c_1, R, c_2, S) \\ \mid n \mid \mathsf{Private} \mid \mathsf{Public} \mid c_1..c_2 \mid \infty \mid c_1 \times c_2 \end{aligned} \qquad (\textit{coeffects})$$

The first line of syntax exposes the resource algebra operations, including syntax for (possibly undefined) least-upper bound ($\sqcup$) and greatest-lower bound ($\sqcap$) derived from the pre-order. Grades can include variables $\alpha$, enabling the grade-polymorphic functions seen in Section 2.

Our long-term goal is to allow first-class user-defined resource algebras, with varying axiomatisations (see §10). For now we provide several built-in resource algebras, with syntax provided above in the second line for naturals $n$, security levels, intervals, infinity, and products of coeffects.

**Definition 4.1.** [Exact usage] The coeffect type Nat has resource algebra given by the usual natural numbers semiring $(\mathbb{N}, +, 0, \cdot, 1, \equiv)$, but notably with *discrete ordering* $\equiv$ giving exact usage analysis in Granule (see Section 2). Thus, meet and join are only defined on matching inputs.

**Definition 4.2.** [Security levels] Coeffect type Level captures a resource algebra over a two-point lattice $\{\mathsf{Public} \sqsubseteq \mathsf{Private}\}$ with $+ = \cdot = \wedge$ (meet) of the lattice and $0 = \mathsf{Public}$ and $1 = \mathsf{Private}$. This algebra exploits our weakening of additive units to $0{+}r \sqsubseteq r$ such that $\mathsf{Public}{+}r = \mathsf{Public}{\wedge}r = \mathsf{Public}$. Recall that + represents contraction (i.e., a split in the dataflow of a value). Therefore, if a value is used publicly, then it must be permitted for public use, even if it used elsewhere privately, thus we need $\mathsf{Public} + \mathsf{Private} = \mathsf{Public}$. However, we still want to be able to weaken at any security level, thus $0 = \mathsf{Public}$ for weakening, which can then be approximated to Private to weaken at any level.

**Definition 4.3.** [Intervals] The Interval constructor is a unary constructor of kind $\mathsf{Coeffect} \to \mathsf{Coeffect}$ where $\mathsf{Interval}\ R$ is inhabited by pairs of $R$ values describing a lower and upper bounds, written as $c_l..c_u$ in Granule syntax. Thus, $\mathsf{Interval}\ R$ is the semiring over $\{(c, d) \mid c \in R, \wedge d \in R \wedge c \sqsubseteq_R d\}$, i.e., pairs where the first component is less than the second (according to corresponding pre order on $R$). Units are defined $0 = 0_R \times 0_R$ and $1 = 1_R \times 1_R$ and the operations and pre-order are defined in the usual way for interval arithmetic:

$$\begin{aligned} c_l..c_u + d_l..d_u &= (c_l +_R d_l)..(c_u +_R d_u) \\ c_l..c_u \cdot d_l..d_u &= (c_l \cdot d_l \wedge_R c_l \cdot d_u \wedge_R c_u \cdot d_l \wedge_R c_u \cdot d_u)..(c_l \cdot d_l \vee_R c_l \cdot d_u \vee_R c_u \cdot d_l \vee_R c_u \cdot d_u) \\ c_l..c_u \sqsubseteq d_l..d_u &= d_l \sqsubseteq_R d_u \ \wedge c_l \sqsubseteq_R d \end{aligned}$$

For Interval Nat (which was used in Section 2 for capturing approximate usage), multiplication collapses to $c_l..c_u \cdot d_l..d_u = (c_l \cdot d_l)..(c_u \cdot d_u)$.

**Definition 4.4.** [Extended coeffects] The Ext constructor is a unary constructor on Coeffect: for some coeffect type $R$ then Ext $R$ extends the resource algebra with an element $\infty$ with operations:

$$r + s = \begin{cases} \infty & r = \infty \vee s = \infty \\ r +_R s & \textit{otherwise} \end{cases} \qquad r \cdot s = \begin{cases} 0_R & r = 0_R \vee s = 0_R \\ \infty & r = \infty \vee s = \infty \\ r \cdot_R s & \textit{otherwise} \end{cases}$$

The pre-order for Ext $R$ is the same as that as $R$. In the examples of Section 2, we sometimes used coeffects of kind Interval (Ext Nat), where $0..\infty$ captures arbitrary ("Cartesian" usage), providing the ! modality of modal logic. In Granule, the type "`A []`" is an alias for "`A [0..∞]`".

**Definition 4.5.** [Products] Given two coeffect resource algebras $R$ and $S$, we can form a resource algebra as the $R \times S$ whose operations are the pairwise application of the underlying operations for $R$ and $S$, e.g., $(r, s)+(r', s') = (r +_R r', s +_R s')$. This is useful for composing grades together to capture multiple properties at once.

Other interesting possible coeffect systems are described in the literature, including hardware schedules [Ghica and Smith 2014], monotonicity information [Arntzenius and Krishnaswami 2016; Atkey and Wood 2018], deconstructor usage [Petricek et al. 2013], and sensitivity [de Amorim et al. 2017]. Future work is to make our system user extensible, but it is already straightforward to extend the implementation with further graded modalities.

Finally, an inter-resource algebra operation "flatten" describes how to sequentially compose two levels of grading, which occurs when we have nested pattern matching on nested graded modalities. Consider the following example, which takes a value inside two layers of graded modality, pattern matches on both simultaneously, and then uses the value:

```
unpack : (Int [2]) [3] → Int
unpack [[x]] = x + x + x + x + x + x
```

Here, the unboxing computes the multiplication of the two grades, stating that `x` is used exactly six times. What if two graded modal boxes have information of different coeffect type? The flatten operation is used here, taking two coeffect terms and their types (i.e., $c_1 : R$ and $c_2 : S$), computing a coeffect term describing sequential composition of $c_1$ and $c_2$, resolved to a particular (possibly different) coeffect type. If $\text{flatten}(r, R, s, S) = r' : R'$ then we can type the following:

```
flat : ∀ { a : Type, r : R, s : R } . a [r] [s] → a [r' : R']
flat [[x]] = [x]
```

The flatten operation is defined as follows in Granule (but can be easily extended at a later date):

**Definition 4.6.** [Flatten] For the built-in resource algebras of Gr, we define flatten as follows:

$$\begin{array}{ll} \text{flatten}(r, \text{Nat}, s, \text{Nat}) & = r \cdot s : \text{Nat} \\ \text{flatten}(r, \text{Level}, s, \text{Level}) & = r \vee s : \text{Level} \\ \text{flatten}(r, R, s, S) & = r \cdot s : \text{Ext Nat} \quad \textit{if} \quad R = \text{Ext Nat} \vee S = \text{Ext Nat} \\ \text{flatten}(r, R, s, S) & = (r, s) : R \times S \end{array}$$

Thus for natural numbers we flatten using multiplication, and similarly if we are combining a natural number with an extended natural number (resolving to the larger type Ext Nat). For levels, we take the join, i.e., $\Box_{\text{Public}}(\Box_{\text{Private}}\alpha)$ is flattened to $\Box_{\text{Private}}\alpha$, avoiding leakage.

flatten is a homomorphism with respect to the resource algebra operations.

Whilst the above resource algebras are for graded necessity modalities (graded comonads), Granule also has another flavour of graded modality: *graded monads/possibility*, written $\Diamond_\varepsilon A$. Following the literature (see Section 9), we provide graded possibility indexed by pre-ordered monoids, captured

by $(\mathcal{E}, \star, 1, \leq)$. Here our built-in graded modality for I/O has the lattice of subsets of effect labels $\mathcal{E} = \mathcal{P}(\{\texttt{Open}, \texttt{Read}, \texttt{IOExcept}, \texttt{Close}, \texttt{Write}\})$ as used in Section 2.5.

### 4.3 Typing – declaratively

The declarative specification of the type system has judgments of the form:

$$(\textit{typing}) \quad D; \Sigma; \Gamma \vdash t : A \qquad\qquad (\textit{kinding}) \quad \Sigma \vdash A : \kappa$$

where $D$ ranges over a context of top-level definitions (including data constructors), $\Sigma$ ranges over a context of type variables and $\Gamma$ ranges of contexts of term variables. Term contexts $\Gamma$ are defined as in GrMini but we now include an optional type signature on graded assumptions, written $x : [A]_{r:R}$ (where $R$ is of kind Coeffect), since Granule allows various graded modalities.

Type variable contexts $\Sigma$ are defined as follows:

$$\Sigma ::= \emptyset \mid \Sigma, \alpha : \kappa \mid \Sigma, \alpha :_{\exists} \kappa \qquad (\textit{type-variable contexts})$$

Assumptions $\alpha : \kappa$ denote universally quantified variables whilst assumptions annotated with $\exists$ are unification variables (existentials). Type variable contexts are concatenated by a comma.

The typing rules of GrMini were given in the form $\Gamma \vdash t : A$. Every GrMini rule is a specialisation of the corresponding Gr rule with an empty type variable context, i.e. $\emptyset; \emptyset; \Gamma \vdash t : A$. We first describe the kinding relation (§4.3.1), unification and substitutions (§4.3.2), typing patterns (§4.3.4), and then finally the declarative specification of typing (§4.3.5).

*4.3.1 Kinding.* Kinding is defined in Figure 1, where kinds are assigned to types in an environment of type variables $\Sigma$. The first two rules are standard giving the kinding of function types ($\kappa_{\rightarrow}$) and type application ($\kappa$APP). We do not have explict kind polymorphism, but ($\kappa\times$) is defined for arbitrary kinds. Products are used at the type level in the standard way and for pairing coeffects types (see Definition 4.5). Type variables are kinded by ($\kappa$VAR) and ($\kappa\exists$VAR). Rule ($\kappa\Box$) gives the kinding of the graded modal necessity ($\Box_r A$), where $r$ has a kind which is a promoted coeffect type. Thus, we view coeffect terms $r$ as residing at the type level. Similarly, ($\kappa\Diamond$) gives the kinding for graded modal possibility ($\Diamond_\epsilon A$) for effects. The next three rules ($\kappa_{\text{op1}}$, $\kappa_{\text{op2}}$, and $\kappa_{\text{op}-}$) give the kinds of type-level binary operators, first capturing resource algebra operations then equations and inequations over coeffect terms in the Predicate kind, and cut-off subtraction for natural numbers. The remaining rules kind the built-in type constructors $K$, including the types for resource algebras (see §4.2).

*4.3.2 Unification and substitutions.* Throughout typing we use *type substitutions*, ranged over by $\theta$, which map from type variables $\alpha$ to types $A$, with individual mappings written as $\alpha \mapsto A$. Substitutions are key to polymorphism: similarly to ML, type schemes are instantiated by creating a substitution from the universally quantified variables to fresh instance (unification) variables [Milner et al. 1997]. *Type unification* then provides type equality, generating a type substitution from unification variables to their resolved types. Much of the machinery for substitutions is standard from the literature on polymorphism and indexed types (e.g., [Dunfield and Krishnaswami 2013; Milner et al. 1997]), however we are not concerned with notions like *most general unifier* as we do not provide inference nor consider a notion of principal types (see Section 10).

Substitutions can be applied to types, kinds, grades, contexts $\Gamma$, type variable contexts $\Sigma$, and substitutions themselves. Substitution application is written $\theta A$, i.e. applying the substitution $\theta$ to the type $A$, yielding a type. The appendix (Definition A.1) gives the full definition, which recursively applies a substitution anywhere type variables can occur, rewriting any matching type variables. Substitutions can also be combined, written $\theta_1 \uplus \theta_2$ (discussed below).

Type unification is given by relation $\Sigma \vdash A \sim B \rhd \theta$ in Figure 2. Unification is essentially a congruence over the structure of types (under a context $\Sigma$), creating substitutions from unification

$$\frac{\Sigma \vdash A : \text{Type} \quad \Sigma \vdash B : \text{Type}}{\Sigma \vdash A \to B : \text{Type}}\,\kappa{\to} \qquad \frac{\Sigma \vdash A : \kappa_1 \to \kappa_2 \quad \Sigma \vdash B : \kappa_1}{\Sigma \vdash A\,B : \kappa_2}\,\kappa\text{APP} \qquad \frac{\Sigma \vdash A : \kappa \qquad \Sigma \vdash B : \kappa}{\Sigma \vdash \times A\,B : \kappa}\,\kappa{\times}$$

$$\frac{}{\Sigma, \alpha : \kappa \vdash \alpha : \kappa}\,\kappa\text{VAR} \qquad \frac{}{\Sigma, \alpha :_\exists \kappa \vdash \alpha : \kappa}\,\kappa\exists\text{VAR} \qquad \frac{}{\Sigma \vdash () : \text{Type}}\,\kappa()$$

$$\frac{\Sigma \vdash R : \text{Coeff} \quad \Sigma \vdash r : \uparrow R \quad \Sigma \vdash A : \text{Type}}{\Sigma \vdash \Box_r A : \text{Type}}\,\kappa\Box \qquad \frac{\Sigma \vdash B : \text{Effect} \quad \Sigma \vdash \varepsilon : \uparrow B \quad \Sigma \vdash A : \text{Type}}{\Sigma \vdash \Diamond_\varepsilon A : \text{Type}}\,\kappa\Diamond$$

$$\frac{\text{op} \in \{+, *, \sqcap, \sqcup\} \qquad \Sigma \vdash R : \text{Coeff} \quad \Sigma \vdash A : \uparrow R \quad \Sigma \vdash B : \uparrow R}{\Sigma \vdash A \,\text{op}\, B : \uparrow R}\,\kappa\text{op1} \qquad \frac{\text{op} \in \{<, >, =, \neq\} \qquad \Sigma \vdash R : \text{Coeff} \quad \Sigma \vdash A : \uparrow R \quad \Sigma \vdash B : \uparrow R}{\Sigma \vdash A \,\text{op}\, B : \text{Predicate}}\,\kappa\text{op2}$$

$$\frac{\Sigma \vdash A : \uparrow \text{Nat} \quad \Sigma \vdash B : \uparrow \text{Nat}}{\Sigma \vdash A - B : \uparrow \text{Nat}}\,\kappa\text{op}- \qquad \frac{}{\Sigma \vdash \text{Int} : \text{Type}}\,\kappa\mathbb{Z} \qquad \frac{}{\Sigma \vdash \text{Char} : \text{Type}}\,\kappa\text{C} \qquad \frac{}{\Sigma \vdash \text{Nat} : \text{Coeff}}\,\kappa\mathbb{N}$$

$$\frac{}{\Sigma \vdash \text{Level} : \text{Coeff}}\,\kappa\mathcal{L} \qquad \frac{}{\Sigma \vdash \text{Ext} : \text{Coeff} \to \text{Coeff}}\,\kappa\infty \qquad \frac{}{\Sigma \vdash \text{Interval} : \text{Coeff} \to \text{Coeff}}\,\kappa..$$

Fig. 1. Kinding rules (with abbreviation Coeff for Coeffect)

$$\frac{\Sigma \vdash A' \sim A \rhd \theta_1 \qquad \Sigma \vdash \theta_1 B \sim \theta_1 B' \rhd \theta_2}{\Sigma \vdash A \to B \sim A' \to B' \rhd \theta_1 \uplus \theta_2}\,U{\to} \qquad \frac{\Sigma \vdash A \sim A' \rhd \theta_1 \qquad \Sigma \vdash \theta_1 B \sim \theta_2 B' \rhd \theta_2}{\Sigma \vdash A\,B \sim A'\,B' \rhd \theta_1 \uplus \theta_2}\,U_{\text{APP}}$$

$$\frac{(\alpha : \kappa) \in \Sigma}{\Sigma \vdash \alpha \sim \alpha \rhd \emptyset}\,U_{\text{VAR}=} \qquad \frac{\Sigma \vdash A : \kappa \qquad (\alpha :_\exists \kappa) \in \Sigma}{\Sigma \vdash \alpha \sim A \rhd \alpha \mapsto A}\,U_{\text{VAR}\exists} \qquad \frac{}{\Sigma \vdash K \sim K \rhd \emptyset}\,U_{\text{CON}} \qquad \frac{\Sigma \vdash A : \kappa}{\Sigma \vdash A \sim A \rhd \emptyset}\,U_{=}$$

$$\frac{\Sigma \vdash A \sim A' \rhd \theta \qquad \Sigma \vdash \theta\varepsilon \sim \theta\varepsilon' \rhd \theta'}{\Sigma \vdash \Diamond_\varepsilon A \sim \Diamond_{\varepsilon'} A' \rhd \theta \uplus \theta'}\,U_\Diamond \qquad \frac{\Sigma \vdash A \sim A' \rhd \theta \qquad \Sigma \vdash \theta c \sim \theta c' \rhd \theta'}{\Sigma \vdash \Box_c A \sim \Box_c A' \rhd \theta \uplus \theta'}\,U_\Box$$

Fig. 2. Type unification rules

variables to types, e.g. ($U_{\text{VAR}\exists}$) for $\alpha \sim A$ (which has a symmetric counterpart for $A \sim \alpha$ elided here for brevity). Universally quantified variables can be unified with themselves ($U_{\text{VAR}=}$), and also with unification variables via ($U_{\text{VAR}\exists}$). In multi-premise rules, substitutions generated by unifying subterms are then applied to types being unified in later premises, e.g., as in ($U_{\to}$). Type unification extends to coeffect grade terms, which can also contain type variables. We elide the definition here since it is straightforward and follows a similar scheme to the figure.

Substitutions can be typed by a type-variable environment, $\Sigma \vdash \theta$ (called *compatibility*) which ensures that $\theta$ is well-formend for use in a particular context. Compatibility is a meta-theoretic property, which follows from our rules. Two substitutions $\theta_1$ and $\theta_2$ may be combined as $\theta_1 \uplus \theta_2$ when they are both compatible with the same type-variable environment $\Sigma$ (i.e., $\Sigma \vdash \theta_1$ and $\Sigma \vdash \theta_2$). If $\alpha \mapsto A \in \theta_1$ and $\alpha \mapsto B \in \theta_2$ and if $A$ and $B$ are unifiable $\Sigma \vdash A \sim B \rhd \theta$ then the combined substitution $\theta_1 \uplus \theta_2$ has $\alpha \mapsto \theta A$ and also now includes $\theta$. For example:

$$(\alpha \mapsto (\text{Int} \times \beta)) \uplus (\alpha \mapsto (\gamma \times \text{Char})) = \alpha \mapsto (\text{Int} \times \text{Char}), \beta \mapsto \text{Char}, \gamma \mapsto \text{Int}$$

If two substitutions for the same variable cannot be unified, then context composition is undefined, indicating a type error which is reported to the user in the implementation. Disjoint parts of a subtitution are simply concatenated. Composition also computes the transitive closure of the resulting substitution. The appendix (Definition A.2) gives the full definition.

By lifting types to kinds with ↑, polymorphism in the type of grades is also possible, e.g.

```
poly : ∀ {a : Type, k : Coeffect, c : k} . a [(1+1)*c] → (a, a) [c]
poly [x] = [(x, x)]
```

The grade `(1+1)*c` is for some arbitrary resource algebra `k` of kind Coeffect. Internally, the type signature `c : k` is interpreted as $c :\uparrow k$ (a type variable lifted to a kind). We also promote data types to the kind level, with data constructors lifted to type constructors.

*4.3.3 Top-level definitions & indexed types.* As seen in Section 2, Granule supports algebraic and generalised algebraic data types (providing indexed types) in the style of Haskell [Peyton Jones et al. 2006]. At the start of type checking, all type constructors are kind checked and all data constructors are type checked. In typing relations here, the $D$ environment holds type schemes for data constructors, along with a substitution describing *coercions* from type variables to concrete types, used to implement indexing. For example, the `Cons` data constructor of the `Vec` type in Section 2.4 has the type `Cons : a → Vec n a → Vec (n + 1) a` which is then represented as:

$$\mathsf{Cons} : (\forall(\alpha : \mathsf{Type}, n : \mathsf{Nat}, m : \mathsf{Nat}) \,.\, \alpha \to \mathsf{Vec}\, n\, \alpha \to \mathsf{Vec}\, m\, \alpha,\ \theta_\kappa) \in D \ \ \textit{where}\ \ \theta_\kappa = m \mapsto n + 1$$

We use $\theta_\kappa, \theta'_\kappa$ to range over substitutions used for the coercions, implementing the indices of indexed type data constructors.

The environment $D$ also holds type schemes for top-level function and value definitions. During type checking, type schemes are instantiated to types (without quantification) by creating fresh unification variables. This is provided by the instantiate function:

**Definition 4.7.** [Instantiation] Given a type scheme $\forall\overrightarrow{\alpha : \kappa}\ .\ A$ with an associated set of coercions $\theta_\kappa$ (i.e., from a GADT constructor) then we create an *instantiation* from the binders and coercions:

$$\theta, \Sigma, \theta'_\kappa = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}, \theta_\kappa)$$

where $\theta$ maps from universal variables to new unification variables, $\Sigma$ gives kinds to the unification variables, and $\theta'_\kappa$ is a renamed coercion $\theta_\kappa$. The type scheme is then instantiated by $\theta A$.

For example, with Cons above we get:

$$\begin{aligned}&\mathsf{instantiate}(\{\alpha : \mathsf{Type}, n : \mathsf{Nat}, m : \mathsf{Nat}\}, m \mapsto n + 1)\\ &\quad = (\{\alpha \mapsto \alpha', n \mapsto n', m \mapsto m'\}, \{\alpha' : \mathsf{Type}, n' : \mathsf{Nat}, m' : \mathsf{Nat}\}, \{m' \mapsto n' + 1\})\end{aligned}$$

In the case where there is no coercion, e.g., for top-level definitions which are not data constructors or for algebraic data type constructors without coercions, we simply write: $\theta, \Sigma = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa})$.

*4.3.4 Pattern matching.* Pattern matching, and its interaction with linearity and grading, is one of Granule's novelties. Patterns must be typed, and patterns themselves can incur consumption that is witnessed in types. The declarative specification of typing for patterns is given by judgments:

$$D; \Sigma; r :?R \vdash\ p : A \ \triangleright\ \Delta; \theta \qquad (\textit{pattern typing})$$

A pattern $p$ has type $A$ in the context of definitions $D$ and type variables $\Sigma$. It produces a variable binding context $\Delta$ along with a substitution $\theta$ generated by the pattern match. An additional part of this judgement's context $r :?R$ captures the possibility of having an "enclosing grade", which occurs when we are checking a pattern nested inside an unboxing pattern. Unboxing affects whether further nested patterns bind linear or graded variables. For example, consider the following:

```
copy : ∀ {a : Type} . a [2] → (a, a);    copy [x] = (x, x)
```

The pattern match $[x]$ creates a graded assumption $x : [\alpha]_2$ in the context of the body. Thus, when

checking a box pattern we push the grade of its associated graded modal type down to the inner patterns. This is information is captured in the typing rules by optional coeffect information:

$$r :?R \quad ::= \quad - \mid r : R \qquad \textit{(enclosing coeffect)}$$

where $-$ means we are not inside a box pattern and $r : R$ means we are inside a box pattern with grade $r$ of type $R$. Typing of variable patterns then splits into two rules depending on whether the variable pattern occurs inside a box pattern or not:

$$\frac{\Sigma \vdash A : \mathsf{Type}}{D; \Sigma; - \vdash\ x : A \rhd x : A; \emptyset}\ \textsc{pVar} \qquad \frac{\Sigma \vdash A : \mathsf{Type}}{D; \Sigma; r : R \vdash\ x : A \rhd x : [A]_{r:R}; \emptyset}\ [\textsc{pVar}]$$

On the left, the variable pattern is not inside a box pattern so we can type $x$ at any type $A$ producing a linear binding context $x : A$ (on the right of the $\rhd$). This pattern does not produce any substitutions. On the right, this variable pattern is nested inside a box pattern and subsequently we can check at type $A$ producing a binding graded by the enclosing box's grade $r : R$.

The typing for box patterns shows the propagation of information about the box to the subpatterns in the premise of the typing rule:

$$\frac{D; \Sigma; r : R \vdash\ p : A \rhd \Delta; \theta \qquad \Sigma \vdash r : \uparrow R}{D; \Sigma; - \vdash\ [p] : \Box_r A \rhd \Delta; \theta}\ \textsc{p}\Box$$

Thus, we can type a box pattern $[p]$ as $\Box_r A$ if we can type its subpattern $p$ at $A$, under the context of the grading $r : R$. Subsequently, any variable bindings in $p$ will appear in $\Delta$ graded by $r : R$. Note that this rule itself applies only in a context *not* enclosed by another box pattern.

Nested box patterns lead to interactions between graded modalities, handled by flatten, with the following rule for a box pattern enclosed by another box pattern with grade $r : R$:

$$\frac{D; \Sigma; s : S \vdash\ p : A \rhd \Delta; \theta \qquad \Sigma \vdash r' : \uparrow R' \qquad \mathsf{flatten}(r, R, r', R') = (s, S)}{D; \Sigma; r : R \vdash\ [p] : \Box_{r'} A \rhd \Delta; \theta}\ [\textsc{p}\Box]$$

(see Definition 4.6 for flatten). For example, in the case $R = R' = \mathsf{Nat}$, flatten compute the multiplication of the two grades: $\mathsf{flatten}(r, \mathsf{Nat}, s, \mathsf{Nat}) = (r \cdot s, \mathsf{Nat})$.

The following rules type wildcard and integer patterns (for the built-in atomic integer type). Both wildcards and integer patterns do not produce bindings, but they have dual notions of consumption to each other, which we reflect in the types:

$$\frac{\Sigma \vdash A : \mathsf{Type} \qquad 0 \sqsubseteq r}{D; \Sigma; r : R \vdash\ \_ : A \rhd \emptyset; \emptyset}\ [\textsc{p}\_] \qquad \frac{}{D; \Sigma; - \vdash\ n : \mathsf{Int} \rhd \emptyset; \emptyset}\ \textsc{pInt} \qquad \frac{1 \sqsubseteq r}{D; \Sigma; r : R \vdash\ n : \mathsf{Int} \rhd \emptyset; \emptyset}\ [\textsc{pInt}]$$

Wildcard patterns can only appear inside a box pattern, since they correspond to weakening, matching any value and discarding it rather than consuming it. Thus the enclosing coeffect must be approximatable by 0, as stipulated by the second premise.

Dually, we treat a pattern match against an integer as triggering inspection of a value which counts as a consumption of the integer. Thus if $n$ is enclosed by a box pattern with grade $r$, then $r$ must approximate 1, i.e., it must count at least one usage.

Constructor patterns are more involved as they may instantiate a polymorphic or indexed constructor, essentially performing a *dependent pattern match* [McBride and McKinna 2004]:

$$\frac{\begin{array}{c}(C : (\forall \overrightarrow{\alpha : \kappa}\ .\ B_0 \to \ldots \to B_n \to K A_0 \ldots A_m, \theta_\kappa)) \in D \qquad \theta, \Sigma', \theta'_\kappa = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}, \theta_\kappa) \\ \Sigma, \Sigma' \vdash \theta(K A_0 \ldots A_m) \sim A \rhd \theta' \qquad D; \Sigma, \Sigma'; - \vdash\ p_i : (\theta'_\kappa \uplus \theta' \uplus \theta) B_i \rhd \Gamma_i; \theta_i\end{array}}{D; \Sigma, \Sigma'; - \vdash\ C\ p_0 \,..\, p_n : A \rhd \Gamma_0, \,..\, , \Gamma_n; \theta'_\kappa \uplus \theta' \uplus \theta_0 \uplus \ldots \uplus \theta_n}\ \textsc{pC}$$

Thus for pattern matching on some constructor $C$ for the data type $K$, we instantiate its polymorphic type (see Def 4.7) in the first line of premises, getting $\theta(KA_0 \ldots A_m)$ with unification variables $\Sigma'$ and renamed coercion $\theta'$. We then unify $\theta(KA_0 \ldots A_m)$ with the expected type $A$ providing another substitution $\theta'$. We now have a way to rewrite the types of the constructor's parameters $B_i$ using the coercion $\theta'_\kappa$, the instantiation substitution $\theta'$, and the substitution $\theta$ coming from unifying the type of pattern with the constructor's result type $KA_0 \ldots A_m$. Thus we check each inner pattern $p_i$ against the types $(\theta'_\kappa \uplus \theta' \uplus \theta)B_i$, yielding their own bindings $\Gamma_i$ and substitutions $\theta_i$ which we collect for the result of pattern matching.

Note that this is the version of the rule when the constructor pattern does not occur inside a box pattern (hence $-$ for the enclosing coeffect grade). A variant of the rule inside a box pattern with $r : R$ is exactly the same but also has the premise constraint that $1 \sqsubseteq r$ similarly to [PINT] above since we treat directly matching a constructor as a consumption. We omit the rule here for brevity.

Some patterns are *irrefutable* meaning that they always succeed. Irrefutable patterns are required in binding positions with just one pattern, e.g., let-bindings and $\lambda$-abstractions. The following predicate characterises these patterns and is used later in typing:

$$\frac{}{\text{irrefutable } \_} \qquad \frac{}{\text{irrefutable } x} \qquad \frac{\text{irrefutable } p}{\text{irrefutable } [p]} \qquad \frac{\text{irrefutable } p_i \quad C \in K \quad \text{cardinality } K \equiv 1}{\text{irrefutable } (C\ p_0 \ldots p_n)}$$

The final case, for data constructor $C$, depends on the type $K$ for which $C$ is a constructor: if $K$ has only one possible data constructor (cardinality 1) then a pattern on $C$ is irrefutable.

*4.3.5 Typing.* We describe each core typing rule in turn. Appendix A collects the rules together. The linear $\lambda$-calculus fragment of GR closely resembles that of GRMINI but abstraction is generalised to pattern matching on its parameter rather than just binding one variable, leveraging pattern typing.

$$\frac{\Sigma \vdash A : \text{Type}}{D; \Sigma; x : A \vdash x : A}\ \text{VAR} \qquad \frac{D; \Sigma; - \vdash p : A \rhd \Delta; \theta \quad D; \Sigma; \Gamma, \Delta \vdash t : \theta B \quad \text{irrefutable } p}{D; \Sigma; \Gamma \vdash \lambda p.t : A \to B}\ \text{ABS} \qquad \frac{D; \Sigma; \Gamma_1 \vdash t_1 : A \to B \quad D; \Sigma; \Gamma_2 \vdash t_2 : A}{D; \Sigma; \Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \text{APP}$$

Variables and application are typed as in GRMINI, but with the additional contexts for definitions and type variables. In (ABS), the substitution produced by pattern matching (e.g., from matching against data constructors) is applied to the result type $B$ to type the body $t$. The context $\Delta$ generated from pattern matching is always disjoint from the existing environment (we assume no variable names overlap, which is ensured in the implementation by a freshening phase) thus the function body is typed under the concatenation of disjoint contexts $\Gamma, \Delta$.

Top-level definitions can be polymorphic and thus environments $D$ associate variable names with type schemes. These polymorphic types are specialised at their usage site, replacing universal variables with fresh unification variables (see Def 4.7) (cf. polymorphic $\lambda$-calculus). Top-level definition usages has two typings, for constructors and value definitions:

$$\frac{(C : (\forall \overrightarrow{\alpha : \kappa}\ .\ A, \theta_\kappa)) \in D \qquad \theta, \Sigma', \theta'_\kappa = \text{instantiate}(\overrightarrow{\alpha : \kappa}, \theta_\kappa)}{D; \Sigma, \Sigma'; \emptyset \vdash C : (\theta'_\kappa \uplus \theta)A}\ \text{C} \qquad \frac{(x : \forall \overrightarrow{\alpha : \kappa}\ .\ B \Rightarrow A) \in D \qquad \theta, \Sigma' = \text{instantiate}(\overrightarrow{\alpha : \kappa}) \qquad \Sigma \vdash A : \text{Predicate} \quad (\theta B)}{D; \Sigma, \Sigma'; \emptyset \vdash x : \theta A}\ \text{DEF}$$

On the left, the universally quantified variables are instantiated, providing an environment $\Sigma'$ of fresh instance variables for each $\alpha$ and a substitution $\theta$ mapping the universal variables to their instance counterparts. As described in Section 4.3.3, we also have a coercion $\theta_\kappa$ for indexed types, which gets instantiated to $\theta'_\kappa$. Thus, the compound substituion $\theta'_\kappa \uplus \theta$ is used to instantiate the type $A$ in the conclusion. The right-hand rule follows a similar approach, but value definitions can also include an additional predicate $B$ in the type which is asserted as a premise of the rule ($\theta B$), i.e., the

predicate must hold at the usage site. There could be many such predicates stated with the type, thus this rule generalises in the expected way to a set of predicates $A_1, \ldots, A_n$. Granule does not yet support having predicates in data constructors, but this is an easy extension.

Weakening, dereliction, and promotion rules resemble those in GrMini:

$$\frac{\Sigma \vdash R : \mathsf{Coeff} \quad D; \Sigma; \Gamma \vdash t : A}{D; \Sigma; \Gamma + (0 : R) \cdot [\Delta] \vdash t : A}\ \textsc{weak} \qquad \frac{\Sigma \vdash R : \mathsf{Coeff} \quad D; \Sigma; \Gamma, x : A \vdash t : B}{D; \Sigma; \Gamma, x : [A]_{1:R} \vdash t : B}\ \textsc{der} \qquad \frac{D; \Sigma; \Gamma \vdash t : A \quad \Sigma \vdash [\Gamma]_R \rhd \Gamma';\ \theta \quad \Sigma \vdash r : R}{D; \theta\Sigma; r \cdot \Gamma' \vdash [t] : \Box_r(\theta A)}\ \textsc{pr}$$

These rules now accommodate the possibility of different graded modalities. Weakening and dereliction occur at some coeffect type $R$, given by a kinding judgement. Promotion requires that all variables $\Gamma$ in scope of the premise are graded at the corresponding coeffect type $R$, where the judgment $\Sigma \vdash [\Gamma]_R \rhd \Gamma';\ \theta$ computes an $R$-graded context $\Gamma'$ for the conclusion, defined as follows:

**Definition 4.8.** [Grade a context] Given a context $\Gamma$ we can turn the context into a graded context $\Gamma'$ with grades of type $R$, by a partial operation $\Sigma \vdash [\Gamma]_R \rhd \Gamma';\ \theta$ which also produces a substitution, where $\Sigma \vdash [\emptyset]_R \rhd \emptyset;\ \emptyset$ for empty contexts, and:

$$\frac{\Sigma \vdash [\Gamma]_R \rhd \Gamma';\ \theta}{\Sigma \vdash [\Gamma, x : A]_R \rhd \Gamma', x : [A]_{1:R};\ \theta}\ [\textsc{lin}] \qquad \frac{\Sigma \vdash [\Gamma]_R \rhd \Gamma';\ \theta' \quad \Sigma \vdash r : R' \quad \Sigma \vdash R \vee R' \rhd S;\theta}{\Sigma \vdash [\Gamma, x : [A]_r]_R \rhd \theta(\Gamma', x : [A]_{r:S});\ \theta \uplus \theta'}\ [\textsc{gr}]$$

On the left, a linear assumption is turned into an assumption graded by 1 of type $R$. On the right, a graded assumption with $r : R'$ can be turned into an assumption graded at $r : S$ if and only if $S$ is the least-upper bound coeffect type of $R$ and $R'$. For example, if $x : [A]_{r:\mathsf{Nat}}$ and we are trying to grade at Ext Nat then we get $x : [A]_{r:\text{Ext Nat}}$ since Nat $\subseteq$ Ext Nat. The judgement $\Sigma \vdash R \vee R' \rhd S;\theta$ (given in Appendix A.2) also serves to unify any type variables in coeffect types and thus produces a substitution which must be combined with the premise's substitution in the output.

The typing of the graded monadic terms shows the facilities in Granule for capturing dataflow information about programs in an alternate way, via the graded monadic composition:

$$\frac{D; \Sigma; - \vdash p : A \rhd \Delta; \theta \quad \text{irrefutable } p \quad D; \Sigma; \Gamma_1 \vdash t_1 : \Diamond_{\varepsilon_1} A \quad D; \Sigma; \Gamma_2, \Delta \vdash t_2 : \Diamond_{\varepsilon_2} \theta B}{D; \Sigma; \Gamma_1 + \Gamma_2 \vdash \textbf{let } \langle p \rangle \leftarrow t_1 \textbf{ in } t_2 : \Diamond_{(\varepsilon_1 \star \varepsilon_2)} B}\ \textsc{let}\Diamond \qquad \frac{D; \Sigma; \Gamma \vdash t : A}{D; \Sigma; \Gamma \vdash \langle t \rangle : \Diamond_1 A}\ \textsc{pure}$$

*Approximation and top-level definitions.* We allow approximation of grades for graded modalities via their resource algebra's pre-orders, with the rules:

$$\frac{D; \Sigma; \Gamma, x : [A]_s, \Gamma' \vdash t : B \quad r \sqsubseteq s}{D; \Sigma; \Gamma, x : [A]_r, \Gamma' \vdash t : B}\ \sqsubseteq \qquad \frac{D; \Sigma; \Gamma \vdash t : \Diamond_\varepsilon B \quad \varepsilon \leq \varepsilon'}{D; \Sigma; \Gamma \vdash t : \Diamond_{\varepsilon'} B}\ \leq$$

For example, for the security levels modality, if we have $x : [A]_{\mathsf{Private}} \vdash t : A$ then we can conclude $x : [A]_{\mathsf{Public}} \vdash t : A$ since Public $\sqsubseteq$ Private. In practice, we've found that allowing the type system to perform approximation arbitrarily for graded necessity modalities during type checking is often confusing for the programmer. Instead, in the implementation of Granule, we allow approximation only at the level of a function definition when the approximation is made explicit by the signature.

Finally, expressions are contained within the equations of top-level definitions given by one or more function equations, along with a polymorphic type scheme signature, with rule:

$$\frac{D; \overrightarrow{\alpha : \kappa}; - \vdash p_i : B_i \rhd \Delta_i; \theta_i \quad D; \overrightarrow{\alpha : \kappa}; \Delta_1, \ldots, \Delta_n \vdash t : (\theta_1 \uplus \ldots \uplus \theta_n) A}{D \vdash x\, p_1 \ldots p_n = t : \forall \overrightarrow{\alpha : \kappa}\ .\ (B_1 \to \ldots \to B_n \to A)}\ \textsc{eqn}$$

The first premise generates the binding context for each pattern, in the context of the universally quantified type variables provided by the type sycheme. The body of the equation $t$ is then typed as

$A$ in the context of the bindings generated from the patterns and with the substitutions generated from pattern matching applied (which could incur dependent pattern matches, with coercions specialising $A$). A type scheme can also include a predicate, i.e., of the form $\forall \overrightarrow{\alpha : \kappa} \,.\, \{A_1, .., A_n\} \Rightarrow B$. We show the typing of equations with such predicates in the algorithmic definition of the type system next, where we explicitly represent predicates and theorems generated by type checking.

## 5 BIDIRECTIONAL TYPE CHECKING ALGORITHM

The previous section gave a declarative definition of the Granule type system, but for an implementation we need an algorithm. The type checking algorithm is based on a *bidirectional approach*, following a similar scheme to that of Dunfield and Krishnaswami [2013]. Type checking is defined by mutual recursive functions for *checking* and *synthesis* of types, of the form:

$$(\textit{checking}) \quad D; \Sigma; \Gamma \vdash t \Leftarrow A; \Delta; \Sigma'; \theta; P \qquad (\textit{synthesis}) \quad D; \Sigma; \Gamma \vdash t \Rightarrow A; \Delta; \Sigma'; \theta; P$$

Checking and synthesis both take as inputs the usual contexts from our declarative definition: top-level definitions and constructors $D$, type-variable kinding $\Sigma$, and an input context $\Gamma$ of term variables, as well as an input term $t$. Checking also takes a type $A$ as input, where as synthesis produces this as output $A$ (hence the direction of the double arrow). Both functions, if they succeed, produce an output context $\Delta$, an output type-variable context $\Sigma'$, output substitution $\theta$, and a theorem (predicate) $P$. The input and output type variable contexts act as state for the known set of unification variables, with the property that the output type-variable context is always a superset of the input type-variable context. The output context $\Delta$ records exactly the variables that were used in $t$ and their computed grades, following a similar approach to Hodas [1994]; Polakow [2015]. A check for each equation determines whether the output context matches the specified input context (derived from pattern matching), which we see below for the algorithmic checking on equations.

For example, the expression $(x + y) + y$ can be checked with an input context where $x$ and $y$ are graded, giving the following:

$$D; \Sigma; x : [\mathsf{Int}]_{r:\mathsf{Nat}}, y : [\mathsf{Int}]_{s:\mathsf{Nat}} \vdash (x + y) + y \Leftarrow \mathsf{Int}; x : [\mathsf{Int}]_{1:\mathsf{Nat}}, y : [\mathsf{Int}]_{2:\mathsf{Nat}}; \Sigma; \emptyset; \top \qquad (1)$$

If this expression is the body of a top-level equation, then we generate a theorem that $r = 1 \wedge s = 2$ to be discharged by the solver (e.g., $r$ and $s$ might represent terms coming from a type signature).

Predicates that can be generated from our algorithm are of the form:

$$P ::= \; P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \rightarrow P_2 \mid \top \mid \forall \Sigma \,.\, P \mid \exists \Sigma \,.\, P \mid \mathbf{t}_1 \equiv \mathbf{t}_2 \mid \mathbf{t}_1 \sqsubseteq \mathbf{t}_2 \qquad \text{(predicates)}$$

Predicates comprise propositional formulae, universal and existential quantification (over type variables, usually of a coeffect/effect kinded type), and equality and inequality over compilable terms (grades and types used in predicates): $\mathbf{t} ::= c \mid \varepsilon \mid A$. Predicates are compiled into the SMT-LIB format [Barrett et al. 2010] and passed to a compatible SMT solver.

Much of the definitions of checking and synthesis are a detailed functionalisations of the relations in the previous section. We highlight just a few details for a subset of the rules.

*Pattern matching.* Pattern matches occur in positions where the type is known, thus we can check the type of a pattern (rather than synthesise), and doing so generates a context of free-variable assumptions which create a local scope. We define pattern type checking via the judgment:

$$D; \Sigma; r :? R \vdash p : A \rhd \Gamma; P; \theta; \Sigma'$$

Algorithmic pattern checking is largely the same as the declarative definition, which already had an algorithmic style, but we now generate a theorem $P$ which encapsulates the consumption constraints generated by patterns which were previously premises in the declarative definition.

*Expressions.* A minimal approach to bidirectional type checking provides checking just for introduction forms and synthesis just for elimination forms [Dunfield and Pfenning 2004]. However, this minimality ends up costing the programmer by way of extra type annotation. Following Dunfield and Krishnaswami [2013], we take a more liberal view providing checking and synthesis for introduction forms of graded modalities and $\lambda$ terms, via an inferential style. Furthermore, we also provide checking for application (an elimination form). Checking has four core rules:

$$\frac{\begin{array}{l}\text{irrefutable } p \\ D; \Sigma_1; - \vdash p : A \rhd \Delta; P; \theta; \Sigma_2 \\ D; \Sigma_2; \Gamma, \Delta \vdash t \Leftarrow \theta B; \Delta'; \Sigma_3; \theta'; P'\end{array}}{D; \Sigma_1; \Gamma \vdash \lambda p.t \Leftarrow A \to B; \Delta' \setminus \Delta; \Sigma_3; \theta \uplus \theta'; P \wedge P'} \Leftarrow_\lambda \qquad \frac{\begin{array}{l}\Sigma_1 \vdash [\Gamma \cap \mathsf{FV}(t)]_R \rhd \Gamma'; \ \theta' \\ \Sigma_1 \vdash r : R \\ D; \Sigma_1; \Gamma' \vdash t \Leftarrow A; \Delta; \Sigma_2; \theta; P\end{array}}{D; \Sigma_1; \Gamma \vdash [t] \Leftarrow \Box_r A; r \cdot \Delta; \Sigma_2; \theta \uplus \theta'; P} \Leftarrow_{\text{PR}}$$

$$\frac{\begin{array}{l}D; \Sigma_1; \Gamma \vdash t_2 \Rightarrow A; \Delta_2; \Sigma_2; \theta_2; P \\ D; \Sigma_2; \Gamma \vdash t_1 \Leftarrow A \to B; \Delta_1; \Sigma_3; \theta_1; P'\end{array}}{D; \Sigma_1; \Gamma \vdash t_1\ t_2 \Leftarrow B; \Delta_1 + \Delta_2; \Sigma_3; \theta_1 \uplus \theta_2; P \wedge P'} \Leftarrow_{\text{APP}} \qquad \frac{\begin{array}{l}D; \Sigma; \Gamma \vdash t \Rightarrow B; \Delta; \Sigma'; \theta; P \\ \Sigma \vdash A \sim B \rhd \theta'; P'\end{array}}{D; \Sigma; \Gamma \vdash t \Leftarrow A; \Delta; \Sigma'; \theta \uplus \theta'; P \wedge P'} \Leftarrow_{\Rightarrow}$$

In each rule, type variable contexts are threaded through judgements as state. Substitutions from premises are combined in the conclusion resulting in type errors if substitutions conflict.

Promotion checking ($\Leftarrow_{\text{PR}}$) reuses the previous function for generating a graded version of a context (Def 4.8), taking as input the subcontext of $\Gamma$ whose variables appear free in $t$, giving the input context of graded assumptions $\Gamma'$ for checking the subterm $t$. The resulting output context $\Delta$ from checking $t$ is then scalar multiplied by the grade $r$.

Predicates generated from multiple premises are combined by conjunction. Usually, the theorem generated from pattern matching is used to form an implication, implying the theorem generated from checking the body (see ($\Leftarrow_{\text{EQN}}$) below). However, for $\lambda$-abstraction we have only one possible pattern (due to irrefutability) and furthermore we perform an additional check in Granule that no pattern match is impossible. Thus, in the ($\Leftarrow_\lambda$), the resulting theorem is really $(P \to P') \wedge \neg\neg P$ which (since predicates are classic) is simplified to $P \wedge P'$. We saw this impossibility check in action in Section 2.4, where an impossible guard for sub is reported as a type error. The rule ($\Leftarrow_\Rightarrow$) connects checking to synthesis, applying algorithmic type unification/equality $\Sigma \vdash A \sim B \rhd \theta'; P$ to check that the synthesised type equals the checked type, which also generates a theorem $P'$.

Top-level definitions start the type checking process, with the rule:

$$\frac{\begin{array}{c}D; \overrightarrow{\alpha : \kappa}; - \vdash p_i : B_i \rhd \Delta_i; P_i; \theta_i; \Sigma_i \\ D; \overrightarrow{\alpha : \kappa}, \Sigma_1, \ldots, \Sigma_n; \Delta_1, \ldots, \Delta_n \vdash t \Leftarrow (\theta_1 \uplus \ldots \uplus \theta_n)A; \Delta; \Sigma'; \theta; P\end{array}}{D \vdash x\ p_1 \ldots p_n = t \Leftarrow \forall \overrightarrow{\alpha : \kappa}\ .\ (B_1 \to \ldots \to B_n \to A); \forall \overrightarrow{\alpha : \kappa}\ .\ \exists \Sigma'\ .\ (P_1 \wedge \ldots \wedge P_n) \to P} \Leftarrow_{\text{EQN}}$$

In the conclusion of the rule, we generate a theorem as an implication from all the predicates for the patterns to the predicate for the body. We then universally quantify all the type variables from the type scheme (though the predicate-to-SMT compiler strips out those of kind Type which don't get compiled into SMT-LIB) and existential quantifying any unification variables generated by checking (again, excluding those which need not be compiled to SMT, e.g. of kind Type).

The rest of the terms are covered by synthesis. Appendix A.3 shows all the rules. We highlight a few here. Variables have their type synthesised by looking up from the input context:

$$\frac{(x : A) \in \Gamma}{D; \Sigma; \Gamma \vdash x \Rightarrow A; x : A; \Sigma; \emptyset; \top} \Rightarrow_{\text{LIN}} \qquad \frac{\Sigma \vdash r : R \quad \Sigma \vdash R : \text{Coeff} \quad (x : [A]_r) \in \Gamma}{D; \Sigma; \Gamma \vdash x \Rightarrow A; x : [A]_{1:R}; \Sigma; \emptyset; \top} \Rightarrow_{\text{GR}}$$

For variables which are graded in the input context, we also perform dereliction as part of synthesis, with grade $1 : R$ for $x$ in the output context.

## 6 OPERATIONAL SEMANTICS

Once the type checker ascertains that a program is well-typed and well-resourced, we pass the AST to an interpreter based on the following operational semantics. We show a small-step semantics here which is useful for proving type-preservation in Section 7. The Granule interpreter however applies a big-step version which includes various built-in operations (provided by Haskell underneath), including arithmetic, file handling, communication and concurrency, which are elided here.

We first define the syntactic category of values $v$ as a subset of the Gr terms:

$$v ::= x \mid n \mid C\ v_0\ ..\ v_n \mid \langle t \rangle \mid [v] \mid \lambda p.t \qquad (values)$$

During reduction, values can be matched against patterns given by a partial function $(v \triangleright p)t = t'$ meaning value $v$ is matched against pattern $p$, substituting values into $t$ to yield $t'$, defined:

$$\frac{}{(v \triangleright \_)t = t}\ \triangleright_{-} \qquad \frac{}{(v \triangleright x)t = [v/x]t}\ \triangleright_{\text{VAR}} \qquad \frac{}{(n \triangleright n)t = t}\ \triangleright_{\text{INT}} \qquad \frac{(v \triangleright p)t = t'}{([v] \triangleright [p])t = t'}\ \triangleright_{\square}$$

$$\frac{(v_i \triangleright p_i)t_i = t_{i+1}}{(C\ v_0\ ..\ v_n \triangleright C\ p_0\ ..\ p_n)t_0 = t_{n+1}}\ \triangleright_{C}$$

Granule has a call-by-value semantics, with reductions $t \rightsquigarrow t'$ defined between terms as follows:

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{APP}_L \qquad \frac{t_2 \rightsquigarrow t_2'}{v\ t_2 \rightsquigarrow v\ t_2'}\ \text{APP}_R \qquad \frac{}{(\lambda p.t)\ v \rightsquigarrow (v \triangleright p)t}\ \text{P}\beta \qquad \frac{}{\textbf{let}\ \langle p \rangle \leftarrow \langle v \rangle\ \textbf{in}\ t_2 \rightsquigarrow (v \triangleright p)t_2}\ \text{LET}\beta$$

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let}\ \langle p \rangle \leftarrow t_1\ \textbf{in}\ t_2 \rightsquigarrow \textbf{let}\ \langle p \rangle \leftarrow t_1'\ \textbf{in}\ t_2}\ \text{LET}_1 \qquad \frac{t \rightsquigarrow t'}{\textbf{let}\ \langle p \rangle \leftarrow \langle t \rangle\ \textbf{in}\ t_2 \rightsquigarrow \textbf{let}\ \langle p \rangle \leftarrow \langle t' \rangle\ \textbf{in}\ t_2}\ \text{LET}_2 \qquad \frac{t \rightsquigarrow t'}{[t] \rightsquigarrow [t']}\ \square$$

## 7 METATHEORY

**Lemma 7.1.** [Well-typed linear substitution] Given $D; \Sigma; \Delta \vdash t' : A$ and $D; \Sigma; \Gamma, x : A, \Gamma' \vdash t : B$ then $D; \Sigma; \Gamma + \Delta + \Gamma' \vdash [t'/x]t : B$.

**Lemma 7.2.** [Well-typed graded substitution] Given $D; \Sigma; [\Delta] \vdash t' : A$ and $D; \Sigma; \Gamma, x : [A]_r, \Gamma' \vdash t : B$ then $D; \Sigma; \Gamma + r \cdot \Delta + \Gamma' \vdash [t'/x]t : B$.

**Lemma 7.3.** [Linear pattern type safety] For all patterns $p$ where irrefutable $p$ and $D; \Sigma; - \vdash p : A \triangleright \Gamma; \theta$ and for values $v$ with $D; \Sigma; \Gamma_2 \vdash v : A$ and terms $t$ depending on the bindings of $p$ such that $D; \Sigma; \Gamma_1, \Gamma \vdash t : \theta B$ then there exists a term $t'$ such that $(v \triangleright p)t = t'$ (progress) and $D; \Sigma; \Gamma_1 + \Gamma_2 \vdash t' : B$ (preservation).

**Lemma 7.4.** [Graded pattern type safety] For all patterns $p$ where irrefutable $p$ with $D; \Sigma; r : R \vdash p : A \triangleright \Gamma; \theta$ and for values $v$ with $D; \Sigma; [\Gamma_2] \vdash v : A$ and terms $t$ depending on the bindings of $p$ such that $D; \Sigma; \Gamma_1, \Gamma \vdash t : \theta B$ then there exists a term $t'$ such that $(v \triangleright p)t = t'$ (progress) and $D; \Sigma; \Gamma_1 + r \cdot \Gamma_2 \vdash t' : B$ (preservation).

**Theorem 7.1.** [Type preservation] For all $D, \Sigma, \Gamma, t, A$ then:

$$D; \Sigma; \Gamma \vdash t : A \implies (\text{value } t) \vee (\exists t', \Gamma'.\ t \rightsquigarrow t' \ \wedge\ D; \Sigma; \Gamma' \vdash t' : A' \ \wedge\ \Gamma' \sqsubseteq \Gamma \ \wedge\ A' \leq A)$$

where $A' \leq A$ and $\Gamma' \sqsubseteq \Gamma$ lift resource algebra preorders to types and contexts as a congruence (with contravariance in premise of a function type for $\leq$).

## 8 FURTHER EXAMPLES

Having seen the details of the type system, we provide some final examples. To illustrate the interaction between different modalities, consider a data type for storing patient information of different privacy levels:

```
data Patient = Patient (String [Public]) (String [Private])
```

where the first field gives the city for a patient (public information) and the second field gives their name (private). We can then define a function that extracts a list of cities from a list of patients:

```
import Vec  -- Granule's standard vector library
getCities : ∀ n. Vec n (Patient [0..1])  → Vec n (String [Public])
getCities Nil = Nil;
getCities (Cons [Patient [city] [name]] ps) = Cons [city] (getCities ps)
```

Since we do not use all of the information in the patient record, we declare the input as affine using an `Interval Nat` modality with `0..1`. The base case (`Nil`) is simple to understand, but in the inductive case we see that the head of the input is `[Patient [city] [name]]`. Let us remind ourselves of the meaning of these nested boxes by looking at the types: the elements of the list are of type `Patient [0..1]` so the outer box is tracking variable usage via $\mathbb{N}$ intervals; the elements of the `Patient` value constructor are of type `String [Public]` and `String [Private]` respectively, so both inner boxes are tracking security level information via the lattice algebra we have seen before. We can safely collect the cities and output a list of public city names in out database. Let us see what happens when we try to accumulate the private name fields into a list of public data:

```
getCitiesBad : ∀ n. Vec n (Patient [0..1])  → Vec n (String [Public])
getCitiesBad Nil = Nil;
getCitiesBad (Cons [Patient [city] [name]] ps) = Cons [name] (getCitiesBad ps)
```

The Granule interpreter gives the following type error:

```
Grading error: 3:54:
Private value cannot be moved to level Public.
```

As a final example, Granule supports *session types* [Yoshida and Vasconcelos 2007] in the style of the GV calculus [Gay and Vasconcelos 2010], leveraging linear types to embed session types primitives. When combined with graded linearity, we can express novel communication properties not supported existing session type approaches. Granule's builtin library provides channel primitives, where `Com` is a trivial graded possibility modality for capturing communication effects:

```
data Protocol = Recv Type Protocol | Send Type Protocol | ...
send : ∀ { a : Type, s : Protocol } . Chan (Send a s) → a → (Chan s) <Com>
recv : ∀ { a : Type, s : Protocol } . Chan (Recv a s) → (a, Chan s) <Com>
forkC : ∀ { s : Protocol, k : Coeffect, c : k } . ((Chan s) [c] → () <Com>)
                                                  → ((Chan (Dual s)) [c]) <Com>
```

where `Dual : Protocol → Protocol` computes the dual of a protocol. Thus, `send` takes a channel on which an `a` can be sent, returning a channel on which behaviour `s` can then be carried out. Similarly, `recv` takes a channel on which one can receive an `a` value, getting back (in a pair) the continuation channel `Chan n`. The `forkC` primitive is higher-order, taking a function that uses a channel in a way captured by some graded modality with grade `c`, producing a session computation. A channel with dual capabilities is returned, that can also be used in a way captured by the grade `c`.

We can use these primitives to capture precisely-bounded replication in protocols:

```
sendVec : ∀ n a .
        (Chan (Send a End)) [n]
    → Vec n a → () <Com>
sendVec [c] Nil = pure ();
sendVec [c] (Cons x xs) =
  let c'  ← send c x;
      ()  ← close c'
  in sendVec [c] xs
```

```
recvVec : ∀ n a .
  N n → (Chan (Recv a End)) [n] → (Vec n a) <Com>
recvVec Z [c] = pure Nil;
recvVec (S n) [c] =
  let (x, c') ← recv c;
      ()      ← close c';
      xs      ← recvVec n [c]
  in pure (Cons x xs)
```

On the left, `sendVec` has a channel which it can use exactly `n` times to send values of type `a`, taking a vector and sending each element on the channel. Dually, `recvVec` takes a size `n` and a channel which it uses `n` times to receive values of `a`, collecting the results into an output vector. We can then put these two processes together using `forkC`:

```
example : ∀ {n : Nat, a : Type} . N n → Vec n a → (Vec n a) <Com>
example n list = let c ← forkC (λc → sendVec c list) in recvVec n c
```

## 9 RELATED WORK

*Graded modalities and coeffects.* Bounded Linear Logic can be considered one of the first graded modal systems, augmenting linear logic with a family of modalities indexed by bounds on the amount of reuse allowed [Girard et al. 1992]. Girard et al. first generalise linear logic's exponential modality ! into a family indexed by natural numbers where $!_x A$ meaning that a value $A$ can only be used at most $x$ times. BLL then generalises this further to "resource polynomial" indices which essentially capture equations over variable terms. Our graded modalities over Interval Nat provide the same as power as BLL and more, allowing lower and upper bounds on reuse, with arbitrary arithmetic expressions over variables. Naturally, type checking is then undecidable for us in general, but so far this has no proved to be a limitation: our standard library replicates many standard functional programming ideas, with decidable types.

In the 2010s, Bounded Linear Logic was subject to various generalisations, capturing other properties of programs via changing or generalising the indices of the modality. For example, Dal Lago and Gaboardi presented a linear PCF with modalities indexed by usage bounds, whose indices could depend on natural number values [Dal Lago and Gaboardi 2011; Gaboardi et al. 2013]. This a specialisation of the kind of general indexed typing we can exploit in Granule. De Amorim et al. [2014] used linear types with natural-number indexed modalities to capture fine-grained analyses for differential privacy. Our declarative type system has a similar shape to theirs, but we generalise our approach considerably to captures different modalities, polymorphism, and pattern matching.

At the same time as specialised efforts to build off BLL, the notion of *coeffects* arose in literature almost simultaneously from three independent sources: as a dualisation of effect systems by Petricek et al. [2013, 2014], and as a generalisation of Bounded Linear Logic by Ghica and Smith [2014] and Brunel et al. [2014]. Each system has essentially the same structure, with a categorical semantics captured by a graded exponential comonad. Each system tracks quantitative properties by instantiating a type system with a semiring algebra. In all of these approaches, coeffects capture how a program depends on its context, capturing how variables are used and subsequently how data flows through a program. The approach of Brunel et al., gives a direct generalisation of BLL, replacing the family of modalities indexed by natural numbers with any family of modalities indexed by an arbitrary semiring. In the approach of Petricek et al. and Ghica et al., the modalities are made implicit, with semiring elements associated to each variable binding (annotating a function

arrow), but the systems have essentially the same structure and power. The categorical foundations of graded exponential comonads have since been studied in more depth [Breuvart and Pagani 2015; Katsumata 2018]. Our approach here was to focus more on program properties captures by graded comonads, when combined with standard programming language features and linearity. We followed the explicit approach of Brunel et al. [2014]; De Amorim et al. [2014].

*Graded monads.* Dual to graded comonads is the notion of *graded monads* which arose naturally in the literature around the same time, generalising monads to an indexed family of functors (type constructors) with monoidal structure on the indices [Fujii et al. 2016; Katsumata 2014; Milius et al. 2015; Mycroft et al. 2016; Orchard et al. 2014; Smirnov 2008]. Whilst graded comonads are employed to capture how programs depend on their context and use variables, the indices of a graded monad can be used to give more fine-grained information about side effects. This information can then be used to specialise the categorical models. The notion of graded monads generalises other earlier indexed generalisations of monads, such as by Wadler and Thiemann [2003].

Gaboardi et al. [Gaboardi et al. 2016a] brought graded monads and graded (exponential) comonads together into a single calculus, exploring interactions between the two structures via *graded distributive laws*. The coming together of these two dual flavours of modalities is also provided by Granule, but as of yet we have not explored internalising notions of graded distributive law, though it is possible to express such structures directly already in our language.

*Linearity and quantitative type theories.* We previously mentioned work that splits types into kinds of linearity and non-linearity [Mazurak et al. 2010; Wadler 1990]. Data types are however a major pain point in such systems because data types are then either applicable only in one context (linear/non-linear), or require parameterisation on their kind. The language Quill leverages quantified types to provide a system with expressive, kind-polymorphic, and usage constrained programs [Morris 2016]. Our approach is to make linearity the default and use graded when we want to weaken that restriction. As discussed in Section 2, standard parametric polymorphism combined with explicit non-linearity via graded modal types provides a very flexible system.

*Linear Haskell (LH).* Recent work attempts to retrofit linearity onto GHC Haskell [Bernardy et al. 2017], modifying the Core language to support a variant of linearity that is somewhat related to Granule. One major difference is that in LH, nonlinearity is introduced via a consumption multiplicity on function types, dubbed *linearity on the arrow* (akin to implicit coeffect systems [Petricek et al. 2014]) instead of a graded modality like in Granule. While in a language with GADTs the two approaches are isomorphic [Bernardy et al. 2017], we have found that linearity on the arrow is not able to succinctly express intuitions about resource usage. Consider the interface for the safe interaction with files which Bernardy et al. [2017] present, an excerpt of which we reproduce below (left), with the equivalent in Granule on the right. As in the LH presentation, we elide here the `IOMode` parameter, which in the case of Granule also indexes the type of `Handle` (§2).

```
open_LH  :: String → IO_L 1 File        open_Gr  : String → Handle <Open,IOExcept>
close_LH :: File ⊸ IO_L ω ()            close_Gr : Handle → () <Close,IOExcept>
```

In LH, a new $\texttt{IO}_\texttt{L}$ monad parameterised over the multiplicity of its result is necessary to propagate linearity information. The result of $\texttt{open}_\texttt{LH}$ is linear, shown by the `1` multiplicity and the result of $\texttt{close}_\texttt{LH}$ is unrestricted, as indicated by $\omega$. By virtue of having a single, linear, function arrow, the interface provided by Granule is cleaner and easier to understand.

LH makes use of *multiplicity polymorphism* to parameterise over the degree of nonlinearity in functions (akin to our polymorphic grades), such as for example `foldl`, which in LH has the following four incompatible types (on the left) followed by the most general type parameterised over multiplicities `p` and `q` (right-top). Contrast this with the type of `foldl` in Granule (right-bottom).

Multiplicity polymorphism here is subsumed by standard parametric polymorphism in Granule since `a` and `b` can be instantiated with any type of arbitrary linearity.

$$(b \multimap a \multimap b) \to b \multimap \texttt{List}\ a \multimap b$$
$$(b \multimap a \to b) \to b \multimap \texttt{List}\ a \to b$$
$$(b \to a \multimap b) \to b \to \texttt{List}\ a \multimap b$$
$$\underbrace{(b \to a \to b) \to b \to \texttt{List}\ a \to b}$$

Multiplicity monomorphic types of `foldl` in LH.

$$\underbrace{\forall pq.(b \to_p a \to_q b) \to b \to_p \texttt{List}\ a \to_q b}$$

Multiplicity polymorphic type of `foldl` in LH.

$$\underbrace{(b \to a \to b)\ [\,] \to b \to \texttt{List}\ a \to b}$$

Parametrically polymorphic type of `foldl` in Granule.

Furthermore, in Granule since grades (multiplicities) are first-class members of the type-language, we can smoothly combine grading with indexed types to capture an even more precise specification for `foldl: ∀ n a b . (b → a → b) [n] → b → Vec n a → b`.

Bernardy et al. [2017] show why a system based on linear logic, where linearity is tracked in type constructors and not in the kind system leads to better code reuse. Girard was far-sighted to present linear logic in the style with an explicit modality, which we adopt and generalise in Granule. By adding parametric polymorphism to his approach, we can directly abstract over non-linearity, hence obviating the need for multiplicity polymorphism required in other systems.

There are a number of other languages that have built their type systems around some linear-like notion, such as ATS [Xi 2003; Zhu and Xi 2005]. ATS has a strong focus on combining with dependent types and theorem proving, but avoiding polymorphism over linear values. Clean [Brus et al. 1987] on the other hand aimed to be more of a general purpose language, based on uniqueness types. Our work here is a continuation of their themes, but exploring the depths of linearity as originally imagined by Girard, with the additional power of grading which has arisen recently.

## 10 FURTHER WORK AND CONCLUSIONS

*Restricting exchange.* Whilst we have restricted weakening and contraction here, the exchange rule remains in our system. However, if a system could temporarily "switch-off" the exchange rule, then various stronger program properties could be proved, for example, that *map* on a list preserves the order of elements. A possible way to provide such flexibility would be to represent the exchange rule via an operation in a resource algebra. Then graded modalities could track the use of exchange, and thus programs could control its use by way of type signatures, e.g. banning the use of exchange to define the *map* function. Exploring this is further work.

*Rank-N quantification and dependent types.* A much more flexible system can be provided by rank-N quantification (i.e. arbitrary nested quantifications) rather than ML-style type schemes. Future work is to adapt our type system to the rank-N approach, perhaps reusing the recent bidirectional results of Dunfield and Krishnaswami [2019].

Even more attractive is to obviate the need for rank-N types by generalising our system to full Martin-Löf-style dependent types. However, combining linear and dependent types well has been a long-standing difficulty. The linear types mantra of "use at exactly once" interferes with dependent types, where values can be used to form types and terms, providing two phases of use. Various attempts have settled on the compromise that types can depend only on non-linear values [Barber and Plotkin 1996; Cervesato and Pfenning 2002; Krishnaswami et al. 2015]. Recent work by McBride [McBride 2016], refined by Atkey [Atkey 2018], however resolves the interaction of linear and dependent types by augmenting a linear system with usage annotations capturing the number of times a variable is used computationally, akin to our grades here or coeffects. The essential insight is that usage at the type-level is accounted for by 0 of a semiring structure, and

term-level use is tracked in much the same way as BLL. This approach leverages an implicit style of grades, differing to our explicit graded modalities.

Further work for us is to adapt our graded modal approach to accomodate full dependent types, in a similar way to McBride and Atkey. However, graded modalities offer a further attractive proposition: what if we also tracked usage in a fine-grained way at the type-level as well as the term-level (rather than just reducing all type-level use to 0)? This is a topic we have already made significant progress on. Our hope is that such an approach would also enable a system in which graded modalities could be user-defined, internal to the language.

*Usability and implicit grading.* Type-based coeffect analyses in Granule are made explicit via the graded box modalities. As we have seen with our examples, the explicit approach requires the programmer to "box" and "unbox" values rather frequently. Currently we view Granule more as a core language for experimenting with this new paradigm. An implicit approach (e.g., akin to [Bernardy et al. 2017; Ghica and Smith 2014; Petricek et al. 2014]) is attractive as it is superficially more user-friendly (no explicit boxing/unboxing). However, as we have argued, the implicit approach can be much more cumbersome as it does not capture generic operations well. Instead, a compromise may be to have a system in which boxing/unboxing can be inferred and implicitly inserted by the type system, but still made explicit if desired.

*Conclusion.* There has been a flurry of recent work on graded and quantitative types. Granule is a big step forwards, applying these ideas in the context of a real language, taking seriously the role of data types, pattern matching, polymorphism, and multi-modalities for real programs. Despite three decades of linearity, there is still much to yield out of its fertile ground. We have barely scratched the surface of what can be expressed by treating data as a resource, and building fine-grained, quantitative, extensible type systems to capture its properties. Our hope is that, now that Granule has wheels, it will be a useful research vehicle for new ideas in type-based program verification.

## REFERENCES


Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: a functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 214–227. https://doi.org/10.1145/2951913.2951948

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

Robert Atkey and James Wood. 2018. Context Constrained Computation. (2018).

Andrew Barber and Gordon Plotkin. 1996. *Dual intuitionistic linear logic.* University of Edinburgh, Department of Computer Science, Laboratory for ....

Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The STM-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, Vol. 13. 14.

P. N. Benton, Gavin M. Bierman, and Valeria de Paiva. 1998. Computational Types from a Logical Perspective. *J. Funct. Program.* 8, 2 (1998), 177–193. http://journals.cambridge.org/action/displayAbstract?aid=44159

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

Gavin M. Bierman and Valeria CV de Paiva. 2000. On an intuitionistic modal logic. *Studia Logica* 65, 3 (2000), 383–416.

Flavien Breuvart and Michele Pagani. 2015. Modelling coeffects in the relational semantics of linear logic. In *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

TH Brus, Marko CJD van Eekelen, MO Van Leer, and Marinus J Plasmeijer. 1987. Clean—a language for functional graph rewriting. In *Conference on Functional Programming Languages and Computer Architecture*. Springer, 364–384.

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

Ugo Dal Lago and Marco Gaboardi. 2011. Linear dependent types and relative completeness. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*. IEEE, 133–142.

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

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. 2017. A semantic account of metric preservation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 545–556. http://dl.acm.org/citation.cfm?id=3009890

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Ewen Denney. 1998. Refinement types for specification. In *Programming Concepts and Methods PROCOMET'98*. Springer, 148–166.

Joshua Dunfield and Neelakantan R Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 429–442.

Joshua Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *PACMPL* 3, POPL (2019), 9:1–9:28. https://dl.acm.org/citation.cfm?id=3290322

Joshua Dunfield and Frank Pfenning. 2004. Tridirectional typechecking. In *POPL*, Vol. 39. ACM.

Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*. 268–277. https://doi.org/10.1145/113445.113468

Soichiro Fujii, Shinya Katsumata, and Paul-André Melliès. 2016. Towards a formal theory of graded monads. In *FOSSACS*. Springer, 513–530.

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

Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016a. Combining effects and coeffects via grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 476–489. https://doi.org/10.1145/2951913.2951939

Marco Gaboardi, Shin-ya Katsumata, Dominic A Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016b. Combining Effects and Coeffects via Grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ACM, 476–489.

Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear type theory for asynchronous session types. *J. Funct. Program.* 20, 1 (2010), 19–50. https://doi.org/10.1017/S0956796809990268

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

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

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

F. Keith Hanna, Neil Daeche, and Mark Longley. 1990. Specification and verification using dependent types. *IEEE Transactions on Software Engineering* 16, 9 (1990), 949–964.

Joshua S Hodas. 1994. Logic Programming in Intutionistic Linear Logic: Theory, Design and Implementation. *PhD Thesis, University of Pennsylvania, Department of Computer and Information Science* (1994).

Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Computing Surveys (CSUR)* 41, 4 (2009), 21.

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Shin-ya Katsumata. 2014. Parametric effect monads and semantics of effect systems. In *Proceedings of POPL 2014*. ACM, 633–645.

Shin-ya Katsumata. 2018. A Double Category Theoretic Analysis of Graded Linear Exponential Comonads. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 110–127.

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

Per Martin-Löf and Giovanni Sambin. 1984. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples.

Nicholas D Matsakis and Felix S Klock II. 2014. The Rust Language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight linear types in System F°. In *Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010*. 77–88. https://doi.org/10.1145/1708016.1708027

Conor McBride. 2016. I got Plenty o'Nuttin'. In *A List of Successes That Can Change the World*. Springer, 207–233.

Conor McBride and James McKinna. 2004. The view from the left. *Journal of functional programming* 14, 1 (2004), 69–111.

Stefan Milius, Dirk Pattinson, and Lutz Schröder. 2015. Generic Trace Semantics and Graded Monads. In *CALCO*.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML: revised*. MIT press.

J Garrett Morris. 2016. The best of both worlds: linear functional programming without compromise. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 448–461.

Alan Mycroft, Dominic Orchard, and Tomas Petricek. 2016. Effect systems revisited – control-flow algebra and semantics. In *Semantics, Logics, and Calculi: Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*. Springer, 1–32. https://doi.org/10.1007/978-3-319-27810-0_1

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *Journal of Functional Programming* 18, 5-6 (2008), 865–911.

Peter W O'Hearn. 2018. Continuous Reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, 13–25.

Dominic A. Orchard, Tomas Petricek, and Alan Mycroft. 2014. The semantic marriage of monads and effects. *CoRR* abs/1401.5391 (2014). http://arxiv.org/abs/1401.5391

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2013. Coeffects: Unified Static Analysis of Context-Dependence. In *ICALP (2)*. 385–397.

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

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 50–61.

Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. *Mathematical structures in computer science* 11, 4 (2001), 511–540.

Jeff Polakow. 2015. Embedding a full linear Lambda calculus in Haskell.. In *Haskell*, Vol. 15. 177–188.

A.L. Smirnov. 2008. Graded monads and rings of polynomials. *Journal of Mathematical Sciences* 151, 3 (2008), 3032–3051. https://doi.org/10.1007/s10958-008-9013-7

K. Terui. 2001. Light Affine Lambda Calculus and Polytime Strong Normalization. In LICS '01. IEEE Computer Society, 209–220.

Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 269–282.

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

Philip Wadler and Peter Thiemann. 2003. The marriage of effects and monads. *ACM Trans. Comput. Logic* 4 (January 2003), 1–32. Issue 1.

David Walker. 2005. Substructural type systems. *Advanced Topics in Types and Programming Languages* (2005), 3–44.

Hongwei Xi. 2003. Applied type system. In *International Workshop on Types for Proofs and Programs*. Springer, 394–408.

Nobuko Yoshida and Vasco Thudichum Vasconcelos. 2007. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. *Electr. Notes Theor. Comput. Sci.* 171, 4 (2007), 73–93. https://doi.org/10.1016/j.entcs.2007.02.056

Dengping Zhu and Hongwei Xi. 2005. Safe programming with pointers through stateful views. In *International Workshop on Practical Aspects of Declarative Languages*. Springer, 83–97.