Quantitative Program Reasoning with Graded Modal Types

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

In programming, some data acts as a resource (e.g., file handles, channels) subject to usage constraints. This poses a challenge to software correctness as most languages are agnostic to constraints on data. The approach of linear types provides a partial remedy, delineating data into resources to be used but never copied or discarded, and unconstrained values. Bounded Linear Logic provides a more fine-grained approach, quantifying non-linear use via an indexed-family of modalities. Recent work on *coeffect types* generalises this idea to *graded comonads*, providing type systems which can capture various program properties. Here, we propose the umbrella notion of *graded modal types*, encompassing coeffect types and dual notions of type-based effect reasoning via *graded monads*. In combination with linear and indexed types, we show that graded modal types provide an expressive type theory for quantitative program reasoning, advancing the reach of type systems to capture and verify a broader set of program properties. We demonstrate this approach via a type system embodied in a fully-fledged functional language called Granule, exploring various examples.

CCS Concepts: • **Theory of computation → Modal and temporal logics**; **Program specifications**; **Program verification**; *Linear logic*; *Type theory*.

Additional Key Words and Phrases: graded modal types, linear types, coeffects, implementation

ACM Reference Format:
Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (August 2019), 30 pages. https://doi.org/10.1145/3341714

1 INTRODUCTION

Most programming languages treat data as infinitely copiable, arbitrarily discardable, and universally unconstrained. However, this overly abstract view is naïve and can lead to software errors. For example, some data encapsulates resources subject to protocols (e.g., file and device handles, channels); some data has confidentiality requirements and thus should not be copied or communicated arbitrarily. Dually, some programs have non-functional properties (e.g., execution time) dependent on data (e.g., on its size). Thus, the reality is that some data acts *as a resource*, subject to constraints.

In this paper we present Granule, a typed functional language that embeds a notion of *data as a resource* into the type system in a way that can be specialised to different resource and dataflow properties. Granule's type system combines linear types, indexed types (lightweight dependent types), and *graded modal types* to enable novel quantitative reasoning.

Linear types treat data like a physical resource which must be used once, and then never again [Girard 1987; Wadler 1990]. For example, the identity function is linearly typed as it binds a

Authors' addresses: Dominic Orchard, School of Computing, University of Kent, UK; Vilem-Benjamin Liepelt, School of Computing, University of Kent, UK; Harley Eades III, School of Computer and Cyber Sciences, Augusta University, USA.

2475-1421/2019/8-ART110
https://doi.org/10.1145/3341714

variable then uses it, whereas the K combinator $\lambda x.\lambda y.x$ is not linearly typed as y is never used. Non-linear, unconstrained use is instead captured by the modal operator !, giving a binary view: either values are linear (a resource) or non-linear (the traditional view of data). However, in programming, non-linearity rapidly permeates programs. Bounded Linear Logic (BLL) instead provides a more fine-grained view, replacing ! with a family of modal operators indexed by terms quantifying the upper bound on usage [Girard et al. 1992], e.g., $!_2 A$ captures A values that can be used at most twice. The proof rules manipulate these indices, accounting for contraction, weakening, and composition.

Various recent work has generalised BLL, providing a family of modalities whose indices are drawn from an arbitrary semiring, enabling various properties to be tracked by a single system, e.g., bounded reuse, strictness, deconstructor use, sensitivity, and scheduling constraints in hardware synthesis [Brunel et al. 2014; Ghica and Smith 2014; Petricek et al. 2013]. These indices are often described as *coeffects*, capturing how a program consumes its context, dualising the idea of *effects*, which capture how a program changes its environment. Semantically, these semiring-indexed modalities are modelled by *graded exponential comonads* [Gaboardi et al. 2016; Katsumata 2018].

We propose the general terminology of *graded modal types* to capture these notions of semiring-indexed !-modalities as well as *graded monads* [Katsumata 2014; Orchard et al. 2014; Smirnov 2008]. Graded monads generalise monads (the Curry-Howard counterpart to modal possibility [Benton et al. 1998]) to a monoid-indexed form, describing side-effects akin to *effect systems*. In general, a graded modality provides an indexed family of type operators with structure over the indices witnessing proof/typing rules. Through the Curry-Howard lens, graded modal types carry information about the semantic structure of programs, and along with a suitably expressive type system, provide a mechanism for specifying and verifying properties not captured by existing type systems.

We develop this idea, presenting a type system that takes linear 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 dual flavours: graded comonads/necessity and graded monads/possibility. We focus mainly on graded necessity, which is heavily integrated with linearity. Whilst the building blocks of this work have been studied, there have been various limitations. Gaboardi et al. [2016] can only accomodate one graded comonad and graded monad at a time in a simply-typed calculus; De Amorim et al. [2014] integrate indexed typing with one graded modality for sensitivity analysis, but also restricted to a single built-in indexed type; Bernardy et al. [2017] have a form of implicit graded modality in the context of Haskell, with GADTs and pattern matching, but restricted to tracking reuse. Specifically, we make the following contributions:

- We define a type theory, combining graded modalities with linear, polymorphic, and indexed types (§3, §4) for the purposes of type-based reasoning. Our work is novel in allowing multiple different graded modalities to be used at the same time, alongside user-defined indexed types (GADTs) in a linear language. We give an operational model (§6) and meta-theoretic results (§7).
- Based on this theory, we present Granule: a statically-typed, eager functional language. Its reasoning powers are demonstrated through various examples (§2, §8), including tracking fine-grained non-linearity, privacy, stateful protocols (like files and sessions), and their combinations.
- We extend the meaning of coeffects to account for pattern matching, and give new constructions on graded modal types, including usage approximations and the combination of graded modalities.
- We provide a bidirectional type checking algorithm (§5), at the heart of our implementation, which exploits an SMT solver to discharge theorems over the indices of graded modalities.

Section 9 discusses related work in detail and Section 10 further work. The appendix (in the auxiliary material https://doi.org/10.1145/3341714) provides further definitions, collected rules, and proofs.

Graded modalities here are for type-based analysis; we do not consider instantiations of the semantics with particular graded (co)monads, e.g., as in the categorical model of Gaboardi et al.

[2016]. Our graded modalities can thus be considered to be computationally trivial, i.e., not requiring additional underlying semantics, though we provide a graded possibility encapsulating I/O effects.

At the moment, Granule is not designed as a general-use surface-level language. Rather, our aim is to demonstrate the reasoning power provided by combining linear, graded, and indexed types in the context of standard language features like data types, pattern matching, and recursion.

2 A TASTE OF GRANULE

We begin with various example programs in Granule, building from the established concept of linear types up to the graded modalities of this paper. We show how linear and graded modal types allow us to document, discover, and enforce program properties, complementing and extending the reasoning provided by parametric polymorphism and indexed types.

Granule syntactically resembles Haskell. Programs comprise mutually recursive definitions, with functions given by sequences of equations, using pattern matching to distinguish their cases. Top-level definitions must have a type signature (inference and principal types is further work, §10). The TEX source of this section is a literate Granule file; everything here is real code. Ill-typed definitions are marked by ✗. We invite the reader to run the type checker and interpreter themselves.[1]

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
```

```
flip : ∀ {a b c : Type} . (a → b → c) → b → a → c
flip f y x = f x y
```

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 `flip` switches around the order in which a function takes its arguments. From `flip` 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 checker and 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.

Granule's type system is not just an extension of ML-like polymorphic systems. The quantification `∀ {t : Type}` ranges over types which may be resources, subject to consumption constraints, precluding arbitrary copying and dropping. Having strict linearity as the default fits the rule that *the more polymorphic our inputs, the less we can assume about them.* This strictness means linearity can enforce aspects of stateful protocols not readily captured by other type systems, even dependent types, e.g. for file handling [Walker 2005] and networking [Tov and Pucella 2010]. It can further be leveraged to treat pure data as resource-like, constraining the space of possible implementations.

Whilst polymorphic `drop` and `copy` are disallowed, we can define monomorphic versions for data types whose constructors are in scope, e.g. for **`data`** `Bool = False | True`:

```
dropBool : Bool → ()
dropBool False = ();
dropBool True = ()
```

```
copyBool : Bool → (Bool, Bool)
copyBool False = (False, False);
copyBool True  = (True,  True)
```

[1] The `gr` toolchain is available at https://granule-project.github.io. To see type checker output for ill-typed examples, pass `gr` the name of the environment: `--literate-env 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.

Thus, data constructors—as opposed to variables—can be used freely. Values of an abstract type (i.e., with hidden data constructors)[2] are however subject to linearity constraints. For example, Granule has an abstract type of file handles, supporting safe programming with files through the following interface which guarantees that an open handle is always closed and then never used again after:

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

This is a subset of the file-handling operations in Granule. The functions `openHandle` and `closeHandle` are polymorphic on the `HandleType`, an ordinary 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 (indicated by the `R` parameter to `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 the same handle. The `<IO>` type is a modality, written postfix, which captures I/O side effects like Haskell's `IO` monad [Jones 2003]. We explain `<IO>` more later (§2.5) as it approximates a more fine-grained graded modality. The next two programs use Granule's notation for sequencing effectful computations akin to Haskell's "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 lifts a pure value into the `<IO>` type. 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 is rejected with several linearity errors: h_1 is used more than once and h_2 and h_1' are discarded. Thus, as is well-known, a lack of weakening and contraction supports static reasoning about the stateful protocol of file handling [Walker 2005].

2.2 Graded Modalities

Many programs however require discarding and copying of values. Linear logic [Girard 1987] answers this by using a modal type constructor ! to relax linearity and propagate any nonlinearity requirements. We can 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)
```

Similar to linear logic's !, the postfix "box" constructor describes unrestricted use. Our syntax is an allusion to necessity (□) from modal logic. Since the parameters are modal, of type `t []`, we can use an "unboxing" pattern to bind a variable `x` of type `t` which can be discarded or copied freely in the function bodies. A value of type `t []` is itself still subject to linearity: it must be used, which it is here via the unboxing pattern. This modality allows non-linearity, but it gives a coarse-grained view: we cannot distinguish the different forms of non-linearity employed by `drop'` and `copy'` whose parameters have the same type. Instead, we achieve this distinction via a *graded modality*.

[2]Hiding data constructors, e.g. behind an interface, is not yet supported by Granule, but various abstract types are built in.

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

```
drop'' : ∀ {t : Type} . t [0] → ()
drop'' [x] = ()
```

```
copy'' : ∀ {t : Type} . t [2] → (t, t)
copy'' [x] = (x, x)
```

These definitions replay `drop'` and `copy'` but the types now exactly specify the amount of non-linearity: `0` and `2`. The usage is easily determined statically as there is no branching control. We will see various graded modalities in due course, including one for accommodating branching next.

2.3 Analysing Control Flow and Propagating Requirements

Data type definitions in Granule look just like those in Haskell. Consider the following data type:

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

Parameterised data constructors are linear functions as each argument appears once in the result, e.g., `Some` is a linear function `∀ t. t → Maybe t`: if its constructed value is used exactly once, then its argument is used exactly once—it is no more than a wrapper around existing data (similar to Linear Haskell [Bernardy et al. 2017]). From now on we omit unambiguous kinds in quantifications. We 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 as it discards the first parameter in the first equation, violating linearity. The type `∀ t. t → Maybe t → t` is uninhabited in Granule. On the left, `fromMaybe` is the well-typed version from Granule's Standard Library. The first parameter is wrapped in a graded modality using a resource algebra capturing both upper and lower bounds of use as an interval via the `_..._` constructor. Interval grades are useful in the context of control flow, also giving a more fine-grained analysis than just the upper bounds of BLL. We can express linear, affine, and relevant use respectively as `[1..1]`, `[0..1]`, and `[1..∞]` (via a resource algebra of intervals over extended natural numbers). Furthermore, this graded modality allows more permissive affinity, e.g., `[0..n]`, and more restrictive relevance, e.g., `[1..n]` for some `n`.

The type of `fromMaybe` explains that the first parameter is used either 0 or 1 times (affine) depending on control flow, and the second parameter is used linearly, i.e., there is a pattern match for every data constructor and any contained values (e.g. `x`) are also used linearly. This usage information is local to the function's definition; `fromMaybe` need not consider the context in which it is used nor how a partially applied result may be used—this is tracked at application sites.

To apply `fromMaybe`, we need to pass a graded modal value for the first parameter. Such values can be constructed by *promotion*, written `[t]`, which promotes a term `t` to a graded modal type. For example, `fromMaybe [29]` uses promotion to lift a constant integer to the type `Int [0..1]`. This application yields a linear function of type `Maybe Int → Int` which itself must be used once. Promotion propagates constraints to any free variables captured by it. For example, consider a term (`let [f] = [fromMaybe [x]] in e`) where the result of partially applying `fromMaybe` is promoted and bound to `f`. If `e` uses `f` non-linearly *n* times then this information is propagated through the promotion to the type of `x` which must then have the grading `0..n`, i.e., used 0 to n times in `e`. Thus, via promotion we can compose programs, propagating information at the type level about data use.

Note that there is only one total function in Granule inhabiting this type of `fromMaybe`; an erroneous definition always returning the first parameter is ill-typed. Thus, our types capture both extensional and intensional program properties (*what* a program does and *how* it does it).

2.4 Indexed Types, and Putting It All Together

Indexed types enable type-level access to information about data. Granule supports user-defined indexed types in a similar style to Haskell's GADTs [Peyton Jones et al. 2006]. We use here the classic examples of size-indexed lists (`Vec`) and indexed naturals to demonstrate some novel reasoning.

```
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)
```

Some standard functions over `Vec` are already linear and thus have definitions and types in Granule that look like the usual ones from non-linear languages, e.g., for `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 ensure that the length of the output vector is indeed the sum of the length of the inputs. Due to linearity this type guarantees a further property: *every element from the inputs must appear in the output*, which is not guaranteed by this type in a non-linear language.

Functions which are non-linear in the elements look different to their usual counterparts. For example when taking the length of a vector, we do not consume its elements. Either we must discard elements as on the left, or we must reconstruct the vector 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)
```

Both are provided in the standard library. These two types are incomparable, representing distinct kinds of consumption on vectors, e.g., the type of `length'` allows programs which also reorder elements in the output list. Section 8 discusses such design decisions further. A key part of Granule's expressive power is that grades can be computed from type indices. Consider the following (left):

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

```
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 `rep` function takes a number `n` and a value `t`, replicating the value `n`-times to build a vector `Vec n t`. Evidently this function cannot be linear in `t`. Using indexing as lightweight dependent types, we specify that the number of uses depends exactly on the size of the output vector.

On the right, `sub` defines subtraction on indexed naturals, demonstrating Granule's support for preconditions (refinements) in the context of type schemes (to the left of ⇒). These must hold where the function is used. Such predicates are discharged by the external solver. If we include a case "`sub Z (S n') = sub Z n'`" which violates the precondition `m ⩾ n`, then `gr` gives us an error:

> **Impossible pattern match: 3:1:** Pattern match in an equation of `sub` is impossible as it implies the unsatisfiable condition $\exists n_0 : \mathsf{Nat}.(0 \geqslant n_0 + 1)$

Lastly, we put the above functions together to define a function for "left padding" a vector:

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

The type says that given a target length `m` and a vector of length less-than-or-equal to `n`, we consume the padding element of type `t` exactly `m - n` times to produce an output vector of length `m`.

Assuming totality, this type alone implies the correct implementation modulo reordering, via:

(1) *Parametric polymorphism*: ensuring that the implementation cannot depend on the concrete padding items provided or the items of the input vector (hence we use vectors instead of strings);
(2) *Indexed types*: ensuring correct sizes and enabling specification of the padding element's usage;
(3) *Linear and graded modal types*: ensuring that every item in the input vector appears exactly once in the output and that the padding element is used to pad the vector exactly `m - n` times.

The type of `leftPad` is superficially similar to what we could write in GHC Haskell or a dependently-typed language, modulo the graded modality `[m - n]`, a minor syntactic addition here. However the extra guarantees give us properties for free which we would otherwise have to prove separately.

2.5 Other Graded Modalities

We return to the `<IO>` type constructor, which is an effect-capturing modality (the "diamond" syntax alluding to modal possibility), in the spirit of Haskell's `IO` monad. More precise reasoning is possible in Granule via a *graded possibility modality* providing a *graded monad* [Katsumata 2014]. The indices of this graded modality capture side effects via sets of effect labels, forming a lattice (by subset inclusion), for which `IO` aliases the top element. We can give a more precise type for `twoChars` from the end of Section 2.1: `(Char, Char) <{Open,Read,IOExcept,Close}>` which enforces that running it cannot cause any write effects. Note that currently exceptions (`IOExcept`) cannot be caught and will terminate the program, relying on the runtime/OS to reclaim the program's resources.

So far we have seen variations of the `Nat` coeffect for tracking variable reuse. Another analysis is available via the `Level` coeffect, representing a lattice of security levels for enforcing *noninterference*:

```
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 analyses (variable reuse and security levels). 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 calculus first before Section 4 defines the full system.

3 A CORE SIMPLY-TYPED LINEAR CALCULUS WITH A GRADED MODALITY

To aid understanding, we first establish a subset of Granule, called GrMini, which comprises the linear λ-calculus extended with a graded necessity modality (graded exponential comonad), resembling coeffect calculi of Brunel et al. [2014] and Gaboardi et al. [2016]. Section 4 extends GrMini to 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 rules.

Types and terms of GrMini are those of the linear λ-calculus with two additional pieces 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 λ-calculus, with variables x, is extended with the term-former $[t]$ which promotes a term to a graded modality, typed by $\Box_r A$, as shall be seen in the typing rules. The term $\mathbf{let}\,[x] = t_1\ \mathbf{in}\ t_2$ dually provides elimination for graded modal types. The graded 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 proof/typing rules. This semiring parameterises GrMini as a meta-level entity. In contrast, Gr can internally select different resource algebras and thus modalities. We consider here the usual natural numbers semiring for counting exact number of uses as a running example to aid understanding.

Typing judgments are of the form $\Gamma \vdash t : A$ with typing contexts Γ 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. We will denote the domain of a context Γ, the set of variables assigned a type in the context, by $|\Gamma|$.

Typing for the linear λ-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:

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

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 in GR (§4) it does not exactly denote linear use as $x : [A]_1 \vdash t : B$ does not imply $x : A \vdash t : B$ for all semirings once ordering is added to allow approximation. *Promotion* (PR) introduces graded necessity with grade r, propagating this grade to the assumptions via scalar multiplication of the context by r. For tracking number of uses, the rule states that to produce the *capability* to reuse 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 Γ can be multiplied by a semiring element $r \in R$:

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

The (LET) rule provides elimination for the graded modality 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 grades are removed, or collapsed via the singleton semiring, then this system is essentially 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 [Terui 2001]. This technique avoids issues with substitution underneath promotion, providing cut admissibility. See Section 7 for further discussion.

Any term and type derivation in the simply-typed λ-calculus can be translated into GrMini based on Girard's translation of the simply-typed λ-calculus into an intuitionistic linear calculus [Girard 1987]. The idea is to replace every intuitionistic arrow $A \to B$ with $\Box_\omega A \multimap B$ for the singleton semiring $\{\omega\}$ and subsequently unbox via **let** in abstraction and promote when applying, e.g.

$$\begin{aligned}[\![\lambda f.\lambda x.f(fx) : (A \to A) \to A \to A]\!] = \lambda f'.\lambda x'.\mathbf{let}\,[f] = f'\,\mathbf{in}\,(\mathbf{let}\,[x] = x'\,\mathbf{in}\,f\,[f\,[x]]) \\ : \Box_\omega(\Box_\omega A \multimap A) \multimap \Box_\omega A \multimap A\end{aligned}$$

From GrMini to Granule. 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:

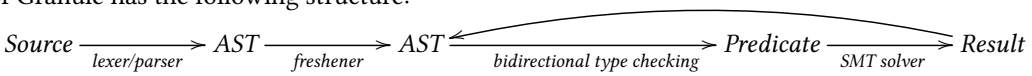
```
k : Int → Int [0] → Int
k = λx → λy' → let [y] = y' in x

foo : Int [3] → Int [6] → Int [3]
foo = λx' → λy' → let [x] = x' in let [y] = y' in [x+y+y]
```

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

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

4 THE GRANULE TYPE SYSTEM

We extend GrMini to Gr, which models Granule with some simplifications. The implementation of Granule has the following structure:



Type checking, if successful, outputs a predicate capturing theorems about grading, which is compiled to the SMT-LIB format [Barrett et al. 2010] and passed to any compatible solver (we use Z3 [De Moura and Bjørner 2008]). Type checking and predicate solving is performed for each top-level definition independently, assuming all other definitions are well-typed by 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:

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

$$p ::= \underbrace{x}_{\text{variables}} \mid \underbrace{\_}_{\text{wildcard}} \mid \underbrace{[p]}_{\text{unbox}} \mid \underbrace{n \mid C\ p_0 \ldots p_n}_{\text{constructors}} \qquad (\textit{patterns})$$

where $\langle t\rangle$ corresponds to `pure t`. Unlike the λ-calculus and GrMini, λ-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:

$$\begin{aligned}
A, B, R, E &::= A \to B \mid K \mid \alpha \mid A\,B \mid A \,\mathsf{op}\, B \mid \Box_c A \mid \Diamond_\varepsilon A && \textit{(types)}\\
\kappa &::= \mathsf{Type} \mid \mathsf{Coeffect} \mid \mathsf{Effect} \mid \mathsf{Predicate} \mid \kappa_1 \to \kappa_2 \mid \uparrow A && \textit{(kinds)}\\
\mathsf{op} &::= + \mid * \mid - \mid \leq \mid < \mid \geq \mid > \mid = \mid \neq \mid \sqcup \mid \sqcap && \textit{(type operators)}\\
K &::= \mathsf{Int} \mid \mathsf{Char} \mid () \mid \times \mid \mathsf{IO} \mid \mathsf{Nat} \mid \mathsf{Level} \mid \mathsf{Ext} \mid \mathsf{Interval} && \textit{(type constructors)}
\end{aligned}$$

Types comprise function types, type constructors K, variables α (and β), application $A\,B$, binary operators $A \,\mathsf{op}\, B$, graded necessity types $\Box_c A$, graded possibility types $\Diamond_\varepsilon A$ where c and ε are coeffect and effect grades defined in Section 4.2, 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 $\to$) as a more familiar concrete syntax. Kinds comprise several constants categorising types, grade kinds (coeffect and effect) and predicates, along with a function space for higher-kinded types and a syntactic construct to denote a type A lifted to a kind, written $\uparrow A$.

Type constructors K comprise various built-in types (with more in the actual implementation, e.g., file handles) and which is extended by user-defined types (§4.3.3). These built-in constructors are grouped above by their kind (defined via the kinding rules, §4.3) with the first three of kind Type, products $\times$ as kind-polymorphic, I/O effect labels of kind Effect, 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 higher-rank quantification (see future work, §10):

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

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) that can be used to express theorems which need to be solved implicitly by the type checker (a kind of refinement), as seen in the sub example earlier (§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 graded necessity modality. Granule instead allows various graded modalities, with different index domains, to be used simultaneously within the same program. The type $\Box_c A$ captures different graded necessity modalities identified by the type of the grade c which is an element of a *resource algebra*: a pre-ordered semiring $(\mathcal{R}, +, 0, \cdot, 1, \sqsubseteq)$ with monotonic multiplication and addition which may be partial. We colour in blue general resource algebra operations, and necessity grades in typing rules.

Within types, necessity grade terms c (which we call *coeffects*) have the following syntax:

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

The first line of syntax exposes the resource algebra operations, including syntax for (possibly undefined) least-upper bounds ($\sqcup$) and greatest-lower bounds ($\sqcap$) derived from the pre-order. Grades can include variables α, enabling grade-polymorphic functions shown later and in Section 8.

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 the 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 §2). Thus, meet and join are only defined on matching inputs.

Definition 4.2. [Security levels] The coeffect type Level provides a way of capturing confidentiality requirements and enforcing noninterference, with a three-point lattice of security levels $\{\mathsf{Irrelevant} \sqsubseteq \mathsf{Private} \sqsubseteq \mathsf{Public}\}$ with $0 = \mathsf{Irrelevant}$, $1 = \mathsf{Private}$, $+ = \sqcup$ (join of the induced lattice), and if $r = \mathsf{Irrelevant}$ or $s = \mathsf{Irrelevant}$ then $r \cdot s = \mathsf{Irrelevant}$ otherwise $r \cdot s = r \sqcup s$.

Multiplication $\cdot$ is such that if a value is used publicly, all of its dependencies must also be public; a private value can depend on public and private values. Recall that $+$ represents contraction (i.e., a split in the dataflow of a value). Therefore, a dependency used publicly must be permitted for public use even if it used elsewhere privately, thus $\mathsf{Public}+\mathsf{Private} = \mathsf{Public}$. Since $0 = \mathsf{Irrelevant}$ (bottom element), we can weaken at any level by approximation. Similarly, since $1 = \mathsf{Private}$, we can essentially derelict at either Private or Public by approximation. The appendix (Lemma A.1) gives the proof that this resource algebra is a preordered semiring.

Definition 4.3. [Intervals] The Interval constructor is unary, of kind $\mathsf{Coeffect} \to \mathsf{Coeffect}$, where $\mathsf{Interval}\ R$ is inhabited by pairs of R elements, giving lower and upper bounds. 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 written with the Granule syntax $c..d$, where the first component is less than the second (according to the preorder on R). Units are $0 = 0_R..0_R$ and $1 = 1_R..1_R$ and the operations and pre-order are defined as in interval arithmetic:

$$c_l..c_u + d_l..d_u = (c_l +_R d_l)..(c_u +_R d_u)$$
$$c_l..c_u \cdot d_l..d_u = (c_l \cdot d_l \sqcap_R c_l \cdot d_u \sqcap_R c_u \cdot d_l \sqcap_R c_u \cdot d_u)..(c_l \cdot d_l \sqcup_R c_l \cdot d_u \sqcup_R c_u \cdot d_l \sqcup_R c_u \cdot d_u)$$
$$c_l..c_u \sqsubseteq d_l..d_u = (d_l \sqsubseteq_R c_l) \wedge (c_u \sqsubseteq_R d_u)$$

For Interval Nat (used in Section 2 to capture lower and upper bounds on usage), multiplication simplifies to $c_l..c_u \cdot d_l..d_u = (c_l \cdot d_l)..(c_u \cdot d_u)$. Whilst Nat is discrete, the implementation uses the $\leq$ natural number ordering to form the Interval Nat resource algebra so that it can properly capture lower and upper bounds on use.

Definition 4.4. [Extended coeffects] For a resource algebra R, applying the unary constructor Ext R extends the resource algebra with an element ∞ (i.e., $\mathsf{Ext}\ R = R \cup \{\infty\}$) with operations:

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

The pre-order for Ext R is that of R, but with $r \sqsubseteq \infty$ for all r. Some Section 2 examples used coeffects of kind Interval (Ext Nat), where $0..\infty$ captures arbitrary ("Cartesian" usage), providing a type analysis akin to the ! modality of linear logic. In Granule, the type "`A []`" is an alias for "`A [0..∞]`".

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

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 that match the structure here.

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—a novel feature. Consider the following example, which takes a value inside two layers of graded modalities, pattern matches on both simultaneously, and then uses the value:

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

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

$$\lambda[[x]].[x] \ : \ \forall\{\alpha : \mathsf{Type}, r : \uparrow R, r' : \uparrow R', s : \uparrow S\} \ . \ \Box_{r'}(\Box_{r}\alpha) \to \Box_{s}\alpha$$

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

Definition 4.6. For the built-in resource algebras, flatten is the symmetric congruence closure of:

$$\begin{array}{ll}
\mathsf{flatten}(r, \mathsf{Ext\ Nat}, s, \mathsf{Ext\ Nat}) = r \cdot s : \mathsf{Ext\ Nat} & \mathsf{flatten}(r, \mathsf{Nat}, s, \mathsf{Ext\ Nat}) = r \cdot s : \mathsf{Ext\ Nat} \\
\mathsf{flatten}(r, R, r_1..r_2, \mathsf{Interval}\ R) = (r \cdot r_1)..(r \cdot r_2) : \mathsf{Interval}\ R & \mathsf{flatten}(r, \mathsf{Nat}, s, \mathsf{Nat}) \quad = r \cdot s : \mathsf{Nat} \\
\mathsf{flatten}(r, R, (r_1, s_1), R \times S) \quad = (r \cdot r_1, s_1) : R \times S & \mathsf{flatten}(r, \mathsf{Level}, s, \mathsf{Level}) \ = r \sqcap s : \mathsf{Level} \\
\mathsf{flatten}(s, S, (r_1, s_1), R \times S) \quad = (r_1, s \cdot s_1) : R \times S & \mathsf{flatten}(r, R, s, S) \ \mid R \neq S = (r, s) : R \times S
\end{array}$$

Thus for Nat we flatten using multiplication, and similarly when combining Nat with an Ext Nat (resolving to the larger type Ext Nat). For levels, we take the meet, i.e., $\Box_{\mathsf{Public}}(\Box_{\mathsf{Private}}\alpha)$ is flattened to $\Box_{\mathsf{Private}}\alpha$, avoiding leakage. For two different resource algebras, flatten forms a product, giving a composite analysis. Note, flatten is a homomorphism with respect to the resource algebra operations. The next section shows how flatten is used in typing.

Whilst the above resource algebras are for graded necessity, Granule also has another flavour of graded modality: *graded possibility*, written $\Diamond_{\varepsilon} A$. Following the literature on graded monads (§9), we provide graded possibility indexed by pre-ordered monoids $(\mathcal{E}, \star, 1, \leq)$ with $1 \leq e$ for all $e \in \mathcal{E}$ and monotonic $\star$. Grade terms ε have syntax capturing these operations. A built-in graded modality for I/O has a lattice of subsets of effect labels $\mathsf{IO} = \mathcal{P}(\{\mathtt{Open}, \mathtt{Read}, \mathtt{IOExcept}, \mathtt{Close}, \mathtt{Write}\})$ as used in Section 2.5, with $(\mathsf{IO}, \cup, \emptyset, \subseteq)$. Other possible graded monads include indexing by natural numbers for cost analysis (as in Danielsson [2008], with $(\mathbb{N}, +, 0, \leq)$). We focus mainly on graded necessity, though the system is easily extended with further graded modalities of both flavours.

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 contexts of top-level definitions (including data constructors), Σ ranges over contexts of type variables and Γ ranges over contexts of term variables. Term contexts Γ 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.

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

$$\frac{}{\Sigma, \alpha : \kappa \vdash \alpha : \kappa}\ \kappa\text{VAR} \qquad \frac{}{\Sigma, \alpha :_\exists \kappa \vdash \alpha : \kappa}\ \kappa\exists\text{VAR} \qquad \frac{A \in \{\text{Int}, \text{Char}, ()\}}{\Sigma \vdash A : \text{Type}}\ \kappa_{\text{Tys}} \qquad \frac{}{\Sigma \vdash \text{IO} : \text{Effect}}\ \kappa\text{IO}$$

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

$$\frac{\text{op} \in \{+, *, \sqcap, \sqcup\} \quad \Sigma \vdash R : \text{Coeff} \quad \Sigma \vdash A : \uparrow R \quad \Sigma \vdash B : \uparrow R}{\Sigma \vdash A\,\text{op}\,B : \uparrow R}\ \kappa_{\text{op1}} \qquad \frac{\text{op} \in \{\leq, =, \neq\} \quad \Sigma \vdash R : \text{(Co)eff} \quad \Sigma \vdash A : \uparrow R \quad \Sigma \vdash B : \uparrow R}{\Sigma \vdash A\,\text{op}\,B : \text{Predicate}}\ \kappa_{\text{op2}}$$

$$\frac{\Sigma \vdash R : \text{Effect} \quad \Sigma \vdash A : \uparrow R \quad \Sigma \vdash B : \uparrow R}{\Sigma \vdash A * B : \uparrow R}\ \kappa_*$$

$$\frac{\Sigma \vdash R : \text{Coeff}}{\Sigma \vdash 0 : \uparrow R}\ \kappa_0 \qquad \frac{\Sigma \vdash R : \text{Coeff}}{\Sigma \vdash 1 : \uparrow R}\ \kappa_1 \qquad \frac{\Sigma \vdash R : \text{Eff}}{\Sigma \vdash 1 : \uparrow R}\ \kappa_1 \qquad \frac{K \in \{\text{Nat}, \text{Level}\}}{\Sigma \vdash K : \text{Coeff}}\ \kappa_{\text{Co}} \qquad \frac{K \in \{\text{Ext}, \text{Interval}\}}{\Sigma \vdash K : \text{Coeff} \to \text{Coeff}}\ \kappa_{\text{Co'}}$$

Fig. 1. Kinding rules (with shorthand Coeff for Coeffect, Eff for Effect, and (Co)eff for either Effect or Coeffect)

Type variable contexts Σ 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 and 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), top-level definitions & GADTs (§4.3.3), typing patterns (§4.3.4), and finally the declarative specification of typing (§4.3.5).

4.3.1 Kinding. Figure 1 defines kinding $\Sigma \vdash A : \kappa$ of a type A as kind κ in a context of type variables Σ. The first two rules are standard, giving kinding of function types ($\kappa\!\to$) and type application (κAPP). We do not have explicit 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 Def. 4.5). Type variables are kinded by (κVAR) and ($\kappa\exists$VAR). The rule ($\kappa\Box$) gives the kinding of graded necessity ($\Box_r A$), where r has a kind which is a coeffect type R lifted to a kind, written $\uparrow R$. Thus, coeffect terms r reside at the type level. Similarly, ($\kappa\Diamond$) gives the kinding of graded possibility ($\Diamond_\varepsilon A$). The next three rules (κ_{op1}, κ_{op2}, and κ_*) give the kinds of type-level binary operators, capturing resource algebra operations and (in)equations over (co)effect terms in the Predicate kind. For Nat, operators $-$, $<$, and $>$ are also available, but their kinding is elided here for brevity. The next three rules give kinding for the resource algebra units and the remaining rules kind the built-in type constructors K, including the types for resource algebras (see §4.2).

The composition of coeffect type constructors has some syntactic restrictions, e.g., Ext (Ext Nat) is disallowed. This is enforced via stratification of the constructors, elided here for brevity as it is not essential to understanding the rest of the system (see appendix, Def. A.1)

Polymorphism in the type of a grade is provided by lifting types to kinds with $\uparrow$. For example:

```
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.

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

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

$$\frac{\Sigma \vdash A \sim A' \rhd \theta_1 \qquad \Sigma \vdash \theta_1 \varepsilon \sim \theta_1 \varepsilon' \rhd \theta_2}{\Sigma \vdash \Diamond_{\varepsilon} A \sim \Diamond_{\varepsilon'} A' \rhd \theta_1 \uplus \theta_2}\, U_{\Diamond} \qquad \frac{\Sigma \vdash A \sim A' \rhd \theta_1 \qquad \Sigma \vdash \theta_1 c \sim \theta_1 c' \rhd \theta_2}{\Sigma \vdash \Box_{c} A \sim \Box_{c'} A' \rhd \theta_1 \uplus \theta_2}\, U_{\Box}$$

Fig. 2. Type unification rules

4.3.2 Unification and Substitutions. Throughout typing we use *type substitutions*, ranged over by θ, which map from type variables α 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 further work, §10).

Substitutions can be applied to types, kinds, grades, contexts Γ, type variable contexts Σ, and substitutions themselves. Substitution application is written θA, i.e. applying the substitution θ to the type A, yielding a type. The appendix (Definition A.2) 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).

Figure 2 defines the type unification relation $\Sigma \vdash A \sim B \rhd \theta$. Unification is essentially a congruence over the structure of types (under a context Σ), creating substitutions from unification variables to types, e.g., ($U_{\text{VAR}\exists}$) for $\alpha \sim A$ (with a symmetric rule for $A \sim \alpha$ elided here). Universally quantified variables can be unified with themselves ($U_{\text{VAR}=}$) and with unification variables via ($U_{\text{VAR}\exists}$). In multi-premise rules, substitutions generated by unifying subterms are applied to types being unified in later premises, e.g., as in ($U_{\to}$). Unification extends to grades, which may contain type variables. We elide the definition as 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 θ is well-formed for use in a particular context. Compatibility is a meta-theoretic property, which follows from our rules. Two substitutions θ_1 and θ_2 may be combined as $\theta_1 \uplus \theta_2$ when they are both compatible with the same type-variable environment Σ (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 θ. 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 substitution are simply concatenated. Composition also computes the transitive closure of the resulting substitution. The appendix (Definition A.3) gives the full definition.

4.3.3 Top-level Definitions and 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, 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 as coercions for indexed type data constructors.

The environment D also holds type schemes of top-level definitions. Type schemes are instantiated to types (without quantification) by the instantiate function, creating fresh unification variables:

Definition 4.7. [Instantiation] Given a type scheme $\forall\{\overrightarrow{\alpha : \kappa}\}\ .\ A$ with an associated set of coercions θ_κ (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 θ maps universal variables to new unification variables, Σ gives kinds to the unification variables, and θ'_κ is a renamed coercion θ_κ. The type scheme is then instantiated by θA.

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. One of Granule's novelties is that patterns can incur consumption which is then captured at the type level by grades. The specification of typing for patterns is similar to other functional languages (e.g., ML [Milner et al. 1997]) but takes into account graded modalities and witnessing of consumption in grades. Patterns are typed by judgments of the form:

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

A pattern p has type A in the context of definitions D and type variables Σ. It produces a variable binding context Δ along with a substitution θ 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 necessity type down to the inner patterns. This information is captured in the typing rules by optional coeffect information:

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

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:

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

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 $\triangleright$). 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 of a box pattern propagates grading information to the typing of the subpattern:

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

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 Δ 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 (Def. 4.6) with the following rule for a box pattern enclosed by another box pattern with grade $r : R$:

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

For example, in the case of $R = R' = \mathsf{Nat}$, flatten computes the multiplication of the two grades: $\mathsf{flatten}(r, \mathsf{Nat}, s, \mathsf{Nat}) = (r \cdot s, \mathsf{Nat})$. Note that double unboxing is distinct from composing two successive unboxing patterns via *let*, e.g., for $t : \Box_r \Box_s A$ then $\mathbf{let}\,[[x]] = t\,\mathbf{in}\,[x] : \Box_{r \cdot s} A$ for any $r : \mathsf{Nat}$ and $s : \mathsf{Nat}$ but $\mathbf{let}\,[x'] = t\,\mathbf{in}\,\mathbf{let}\,[x] = x'\,\mathbf{in}\,[x]$ is only well-typed when $r = 1$ as $x' : \Box_s A$ is used only once. Thus, double unboxing (controlled via flatten) is not derivable, but an important feature of the graded modal analysis here, capturing interaction of two graded necessities.

Depending on flatten, double unboxing and promotion form an isomorphism $\Box_r(\Box_s A) \cong \Box_{(r \cdot s)} A$ for some graded modalities (e.g. for Nat). This is not derivable for all graded modalities, e.g., Level.

The following rules type wildcard and integer patterns. Neither pattern produces bindings, but they have dual notions of consumption (use) which is novelly captured by graded modal 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 as they correspond to weakening, matching any value and discarding rather than consuming it. Thus the enclosing coeffect must be approximable by 0, as stipulated in the premise. Dually, we treat a pattern match against an integer as triggering inspection of a value which counts as a use of the value. Thus if n is enclosed by a box pattern with grade r, then r must approximate 1, e.g. for $R = \mathsf{Nat}$ it must count at least one use.

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 matching on a data constructor C for the data type K, we instantiate its type scheme (Def 4.7) in the first line of premises, getting $\theta(K A_0 \ldots A_m)$ with unification variables Σ' and renamed coercion θ'_κ. We then unify $\theta(K A_0 \ldots A_m)$ with the expected type A yielding a substitution θ'. We now have a way to rewrite the types of the constructor's parameters B_i using the coercion θ'_κ, the instantiation substitution θ, and the substitution θ' from unifying the type of the pattern with the constructor's result type. Thus we check each inner pattern p_i against the type $(\theta'_\kappa \uplus \theta' \uplus \theta) B_i$, yielding bindings Γ_i and substitutions θ_i which are collected 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 λ-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 λ-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 indexed-type data constructors) is applied to the result type B to type the body t. The context Δ 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 Γ, Δ.

Top-level definitions may be polymorphic and thus environments D associate names with type schemes. Type schemes are specialised at their usage site, replacing universal variables with fresh unification variables (see Def 4.7). There are two typings, for constructors and value definitions:

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

On the left, the universally quantified variables are instantiated, providing an environment Σ' of fresh instance variables for each α and a substitution θ mapping the universal variables to their instance counterparts. As described in Section 4.3.3, we also have a coercion θ_κ for indexed types, which gets instantiated to θ'_κ. Thus, the compound substitution $\theta'_\kappa \uplus \theta$ 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 (θB), i.e., the predicate must hold at the usage site. There could be many such predicates stated with the type A, thus this rule generalises in the expected way to a set of predicates. Granule does not yet support predicates in data constructor types, but this is an easy extension.

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

$$\frac{\Sigma \vdash R : \text{Coeff} \quad D; \Sigma; \Gamma \vdash t : A}{D; \Sigma; \Gamma + [\Delta]_{0:R} \vdash t : A}\ \text{WEAK} \qquad \frac{\Sigma \vdash R : \text{Coeff} \quad D; \Sigma; \Gamma, x : A \vdash t : B}{D; \Sigma; \Gamma, x : [A]_{1:R} \vdash t : B}\ \text{DER} \qquad \frac{\Sigma \vdash r : \uparrow R \quad \Sigma \vdash R : \text{Coeff} \quad D; \Sigma; [\Gamma] \vdash t : A}{D; \Sigma; r \cdot \Gamma \vdash [t] : \Box_r A}\ \text{PR}$$

These rules now accommodate the possibility of different graded modalities: weakening, dereliction, and promotion all occur at some coeffect type R, given by a kinding judgement. Similarly to GrMini, promotion requires that all variables Γ in scope of the premise are graded. Differently to GrMini, graded assumptions in the context may not all be graded at the same type R. We account for a context of possibly varying grading types by a redefinition of scalar context multiplication:

Definition 4.8. [Scalar context multiplication for Gr] Given a context of graded assumptions Γ and a semiring element $r \in R$ then Γ can be multiplied by r as follows:

$$r \cdot \emptyset = \emptyset \qquad r \cdot (\Gamma, x : [A]_{s;S}) = (r \cdot \Gamma), x : [A](\iota_1 r \cdot \iota_2 s) \quad \textit{where} \quad R \sqcup S \rhd R'; \iota_1; \iota_2$$

On the right, the graded assumption s may have a different type S to the promoting grade's type R. The function $R \sqcup S \rhd R'; \iota_1; \iota_2$ (see appendix, Def. A.5) computes the least upper-bound type of R and S as R', also generating a pair of injections $\iota_1 : R \to R'$ and $\iota_2 : S \to R'$ to inject the grades into the upper-bound type. For example, $R \sqcup (R \times S) \rhd R \times S; r \mapsto (r, 1); id$, i.e., we can promote an assumption graded by $(r_1, s_1) : R \times S$ with a coeffect $r : R$ by applying the generated injection $r \mapsto (r, 1)$ to r (where 1 is the unit for S) and then multiplying with (r_1, s_1), yielding $(r \cdot r_1, s_1)$.

The typing of monadic metalanguage terms shows Gr's support for capturing dataflow information in an alternate way via graded possibility modalities, graded by Effect resource algebras:

$$\frac{\begin{array}{ll}\Sigma \vdash E : \mathsf{Effect} & \Sigma \vdash \varepsilon_1 : \uparrow E \quad \Sigma \vdash \varepsilon_2 : \uparrow E \\ D; \Sigma; - \vdash p : A \rhd \Delta; \theta & \text{irrefutable } p \\ D; \Sigma; \Gamma_1 \vdash t_1 : \Diamond_{\varepsilon_1} A & D; \Sigma; \Gamma_2, \Delta \vdash t_2 : \Diamond_{\varepsilon_2} \theta B\end{array}}{D; \Sigma; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ \langle p \rangle \leftarrow t_1\ \mathbf{in}\ t_2 : \Diamond_{(\varepsilon_1 \star \varepsilon_2)} B}\ \textsc{let}\Diamond \qquad \frac{\Sigma \vdash E : \mathsf{Effect} \quad D; \Sigma; \Gamma \vdash t : A}{D; \Sigma; \Gamma \vdash \langle t \rangle : \Diamond_{1:E} A}\ \textsc{pure}$$

We allow approximation of grades via their resource algebra's pre-orders, with the following rules:

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

For example, if we have $x : [A]_{\mathsf{Private}} \vdash t : A$ then we can conclude $x : [A]_{\mathsf{Public}} \vdash t : A$ as $\mathsf{Private} \sqsubseteq \mathsf{Public}$ (public information can flow to a private dependency, but not vice versa). Or $x : [A]_{1..2} \vdash t : A$ can be typed as $x : [A]_{0..3} \vdash t : A$ since $(1..2) \sqsubseteq (0..3)$. In practice, we've found that allowing the type system to perform approximation arbitrarily during type checking for graded necessity modalities is often confusing for the programmer. Instead, in the implementation, we allow approximation only at the level of a function definition where approximation is made explicit by the signature. Approximation lifts to types and contexts as a congruence over their structure.

Finally, expressions are contained within the equations of top-level definitions given by one or more function equations, along with a type scheme signature. Each equation of a top-level definition is typed with the following rule (with the same type scheme, coming from the signature):

$$\frac{D; \overrightarrow{\alpha : \kappa}; - \vdash p_i : B_i \rhd \Delta_i; \theta_i \qquad D; \overrightarrow{\alpha : \kappa}; \Delta_0, \ldots, \Delta_n \vdash t : (\theta_0 \uplus \ldots \uplus \theta_n) A}{D \vdash x\ p_0 \ldots p_n = t : \forall \{\overrightarrow{\alpha : \kappa}\}\ .\ (B_0 \to \ldots \to B_n \to A)}\ \textsc{eqn}$$

The first premise generates the binding context for each pattern, in the context of the universally quantified type variables provided by the type scheme. 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 contain a predicate, in which case it is of the form $\forall \overrightarrow{\alpha : \kappa}\ .\ \{A_1, \ldots, A_n\} \Rightarrow B$. 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.

Note that multiple different equations correspond to different control-flow branches and as such may have different gradings. Thus the approximation rules above may be used in order to calculate the least-upper bound gradings across the equations of a top-level definition.

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 as Dunfield and Krishnaswami [2013]. Type checking is defined by mutually 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 kinds Σ, an input context Γ of term variables, and an input term t. Checking also takes a type A as input, whereas synthesis produces A as an output (hence the direction of the double arrow). Both functions, if they succeed, produce an output context Δ, an output type-variable context Σ', output substitution θ, and a 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 Δ 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 top-level definition's equation determines whether the output context matches the specified input context (derived from pattern matching), shown below.

For example, an expression $(x+y)+y$ can be checked in an input context with graded x and y:

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

If this expression is the body of a top-level equation, then we generate a predicate 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).

Our algorithm can generate predicates (first-order formulas) that have the following 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 definition of checking and synthesis is a detailed functionalisation 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 creates 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 predicate 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 λ-terms, via an inferential style. Furthermore, we also provide checking for application (an elimination form). Checking has four core rules:

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

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

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.

In ($\Leftarrow_\lambda$), we see that the predicate generated from pattern matching is used to form an implication, implying the predicate generated from checking the body. Furthermore, any variables generated by the pattern match (that is, $\Sigma_2 \setminus \Sigma_1$) are universally quantified in the scope of this implication.

In ($\Leftarrow_{\text{PR}}$) for promotion, the first premise employs a function which maps contexts into graded contexts by dereliction of linear assumptions (see Def. B.1). This is applied to the subcontext of Γ whose variables appear free in t, yielding the input context of graded assumptions Γ' for checking the subterm t. The output context Δ resulting from checking t is then scalar multiplied by r.

In ($\Leftarrow_{\text{APP}}$) we see that predicates generated from multiple premises are combined by conjunction. The rule ($\Leftarrow_\Rightarrow$) connects checking to synthesis, applying algorithmic unification $\Sigma \vdash A \sim B \rhd \theta'; P$ to check that the synthesised type equals the checked type, generating a further predicate P'.

The type-checking process begins at top-level definitions with the following rule for checking an n-arity function equation against a type scheme (with m type-refinement predicates):

$$\frac{\begin{array}{cc} D; \overrightarrow{\alpha : \kappa}; - \vdash p_i : B_i \rhd \Delta_i; P_i; \theta_i; \Sigma_i & P' = P_0 \wedge .. \wedge P_n \qquad \Sigma' = \overrightarrow{\alpha : \kappa}, \Sigma_0, .., \Sigma_n \\ D; \Sigma'; \Delta_0, .., \Delta_n \vdash t \Leftarrow (\theta_0 \uplus \ldots \uplus \theta_n)A; \Delta; \Sigma''; \theta; P & P'' = (P' \wedge [\![A_0, .., A_m]\!]) \to [\![\Sigma'' \backslash \Sigma']\!].P \end{array}}{D \vdash x\, p_0 \ldots p_n = t \Leftarrow \forall \overrightarrow{\alpha : \kappa}.\{A_0, .., A_m\} \Rightarrow B_0 \to \ldots \to B_n \to A;\ (\forall \Sigma'.P'') \wedge (\exists \Sigma'.P')} \Leftarrow_{\text{EQN}}$$

The generated predicate has two parts. The first conjunct universally quantifies all type variables from the type scheme and unification variables generated by pattern matching (though the predicate-to-SMT compiler strips out those of kind Type, which don't get compiled into SMT-LIB). Under the bindings is the implication P'', whose antecedent is a conjunction of the predicates for the patterns P' and the predicate given by the compilation of the type-refinement predicates $[\![A_0, .., A_m]\!]$. The succedent is then the predicate P from type-checking the body, over which we quantify any remaining type variables $\Sigma'' \setminus \Sigma'$ via an interpretation (existential for unification variables and universal otherwise). The second conjunct of the predicate checks that no pattern match is impossible: there must exist indices making the patterns' predicates true.

The rest of the terms are covered by synthesis. Appendix B shows the rules. We highlight two here. Variables have their type synthesised by looking up from the input context:

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

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

6 OPERATIONAL SEMANTICS

Once type checking shows a program to be well-typed, its AST is interpreted, following a call-by-values semantics. We show a small-step formulation, useful for proving type-preservation (§7). The Granule interpreter applies a big-step version which includes built-in operations elided here (provided by Haskell) such as arithmetic, file handling, and concurrency.

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

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

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

$$\frac{}{(v \rhd \_)t = t}\rhd_{-} \quad \frac{}{(v \rhd x)t = [v/x]t}\rhd_{\text{VAR}} \quad \frac{}{(n \rhd n)t = t}\rhd_{n} \quad \frac{(v \rhd p)t = t'}{([v] \rhd [p])t = t'}\rhd_{\Box} \quad \frac{(v_i \rhd p_i)t_i = t_{i+1}}{(C v_0 .. v_n \rhd C p_0 .. p_n)t_0 = t_{n+1}}\rhd_{C}$$

Call-by-value reduction for terms is then defined by the relation $t \rightsquigarrow t'$ as follows:

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

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

Call-by-value was chosen for simplicity and since the effectful part of Gr is necessarily CBV (to avoid pitfalls of laziness with side effects). Previously, Brunel et al. [2014] provided a CBN abstract machine for BLL-style coeffects where reduction is suspended at promotion, i.e., $[t]$ is a value, and Gaboardi et al. [2016] gave a CBN-based equational theory. This differs to our approach, but our system does not preclude a CBN semantics, which is further work. Such a semantics would trigger reduction when matching non-values against constructor patterns, which would cohere nicely with the consumption constraints on grades induced by pattern matching. The CBV approach, whilst simple, shows that our technique does not force us into the CBN semantics used previously, suggesting that we could add graded modal types to an existing eager language. Note that we do not parameterise our system by denotational models of the graded modalities, i.e, particular graded (co)monads. Exploring this, and its relationship to operational models, is future work.

7 METATHEORY

We have two kinds of substitution lemma for our system, showing that substitution is well-typed when substituting through linear and graded assumptions:

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 $D; \Sigma; \Gamma + r \cdot \Delta + \Gamma' \vdash [t'/x]t : B$.

Remark Some linear type systems with the exponential modality (which we write here as $\Box$) include a promotion rule taking $\Box\Gamma \vdash t : A$ to $\Box\Gamma \vdash [t] : \Box A$ where $\Box\Gamma$ means every hypothesis in the context is modal [Abramsky 1993]. However, in such a system, substitution is not valid (well-typed) [Wadler 1992, 1993] (an issue also noted by Prawitz in a modal context [Prawitz 1965]). Wadler demonstrates this problem with the following example:

$$\textit{not Gr}\ \frac{f : A \to \Box B, x : A \vdash f\ x : \Box B \qquad g : \Box(\Box B \to C), y : \Box B \vdash [\textbf{let}\,[h] = g\ \textbf{in}\ h\ y] : \Box C}{g : \Box(\Box B \to C), f : A \to \Box B, x : A \vdash [\textbf{let}\,[h] = g\ \textbf{in}\ h\,(f\ x)] : \Box C}\ [f\,x\ /\ y]$$

The premises are combined by substituting for y inside a promotion. As Wadler points out, this is not valid since we now have a promotion which closes over linear premises f and x; the resulting judgment cannot be derived. Granule solves this problem via its graded assumptions, which are a graded form of Terui's *discharged assumptions* [Terui 2001] (used also in coeffect work [Brunel et al. 2014; Gaboardi et al. 2016]). Granule's promotion requires the second premise of this example to have g and y as graded assumptions, rather than linear assumptions of graded modal type, i.e., $g : [\Box(\Box B \to C)]_1, y : [\Box B]_1 \vdash [\textbf{let}\,[h] = g\ \textbf{in}\ h\ y] : \Box C$. Subsequently, the linear substitution lemma cannot be applied as y is not linear and the graded substitution lemma cannot be applied as it requires that premises of the substituted term $f\,x$ must be all graded, which they are not here.

Wadler discusses a further issue, exemplified by the following two terms which should be equivalent by substitution (and β-reduction):

$$y : \Box\Box A \vdash [\mathbf{let}\,[z] = y\,\mathbf{in}\,z] : \Box\Box A \qquad y : \Box\Box A \vdash (\lambda x.[x])\,(\mathbf{let}\,[z] = y\,\mathbf{in}\,z) : \Box\Box A$$

Using a comonadic semantics, Wadler shows that these two terms have different denotations, even though substitution makes them appear equal. This problem is avoided in Granule as $\lambda x.[x]$ is not well-typed since the binding of x is linear and therefore promotion of x is disallowed.

Well-typed substitution generalises to arbitrary patterns as follows:

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

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

Theorem 7.1. [Gr type safety] Progress and preservation follow from the above. For all D, Σ, Γ, t, A:

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

where $A' \leq A$ and $\Gamma' \sqsubseteq \Gamma$ lift resource algebra preorders to types and contexts as a congruence (with contravariance in function type parameters). The ordering $\Gamma' \sqsubseteq \Gamma$ means that graded assumptions can become more precise as reduction proceeds. For example, $(0..1) \sqsubseteq (0..\infty)$ (for Interval Nat) thus a reduction from $\Gamma, x : [A]_{0..\infty} \vdash t : A$ to $\Gamma, x : [A]_{0..1} \vdash t' : A$ fits the form of the lemma.

We do not consider I/O exceptions in Gr, though Granule's implementation of IO-graded possibility includes exceptions (e.g., reading an empty file), captured by label `IOExcept`. Such exceptions violate progress, thus disconnecting static notions of linearity from the runtime behaviour. Addressing this, e.g., based on [Iwama et al. 2006], is left as future work.

8 FURTHER EXAMPLES

Interaction with data structures. Graded modalities can be commuted with other data types, for example to pull information about sub-parts of data up to the whole, or dually to push capabilities down to sub-parts of a data type. Such notions are embodied by functions like the following, commuting products with arbitrary graded necessity modalities:

```
push : ∀ {a b : Type, k : Coeffect, c : k}
       . (a, b) [c] → (a [c], b [c])
push [(x, y)] = ([x], [y])
```

```
pull : ∀ {a b : Type, k : Coeffect, c : k}
       . (a [c], b [c]) → (a, b) [c]
pull ([x], [y]) = [(x, y)]
```

A possible intuition for `pull`, when specialised to `Nat` grades, is that making `c` copies of a pair requires `c` copies of each component (a "deep" copy). The notion of copying values can be helpful for thinking about `Nat` grades, but it does not necessarily reflect an actual implementation or execution of the program: no such copying is mandated by the rest of the system.

In practice, combinators such as `push` and `pull` are rarely used as Granule's pattern matching can be used directly to capture the interaction of data types and graded modalities, as in the definitions of `push`/`pull` themselves. Consider the function that returns the first element of a nonempty vector:

```
head : ∀ {a : Type, n : Nat} . (Vec (n+1) a) [0..1] → a
head [Cons x _] = x
```

The input vector has the capability to be consumed `0` or `1` times. Via the unboxing pattern, this capability is pushed down to the vector's sub-parts so that every element and tail must be used `0..1` times. The ability to be used `0`-times is utilised to discard the tail via the inner wildcard pattern.

The Granule standard library[3] provides a variety of data structures including graphs, lists, stacks, vectors. There are often different design decisions for the interaction of data structures and graded modalities. For example, we represent stacks as vectors, with *push* and *pop* as dual linear operations corresponding to `Cons` and uncons respectively, i.e., `pop : ∀ n a. Vec (n+1) a → (a, Vec n a)`. The above `head` function for vectors can then be re-used as the *peek* operation for stacks which, rather than returning a pair, just returns the peeked element. We could define a version of *peek* which emulates the style of `pop`, reusing `head` and returning the original stack in a pair:

```
peekAlt : ∀ {n : Nat, a : Type} . (Vec (n+1) a) [1..2] → (a, Vec (n+1) a)
peekAlt [x] = (head [x], x)
```

The definition divides the capability `1..2` into `0..1` for the head operation and `1..1` for the second component of the pair. The head element is subsequently used twice and the rest of the stack once. In practice, this would not be very useful—it makes more sense to just use `head` directly, and non-linearly use the stack when needed, letting the type system track the usage. A useful alternative to the `head`-based *peek* instead provides a linear interface for the stack but with non-linear elements:

```
peek' : ∀ {n m : Nat, a : Type} . Vec (n+1) (a [m..m+1]) → (a, Vec (n+1) (a [m..m]))
peek' (Cons [x] xs) = (x, Cons [x] xs)
```

The function takes a stack whose elements can be used `m` to `m+1` times. We use this capability to copy the head element, returning a pair of the head and a stack whose elements can be used `m` to `m` times. This form is useful for composing functions which operate on stack elements non-linearly. The `head` operation is more suited to manipulating the whole stack non-linearly, rather than just its elements. Exploring the design space and trade-offs for data structure libraries is further work.

Grade interaction. To illustrate the interaction between different graded necessity modalities, consider a data type for storing patient information of different privacy levels:

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

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, e.g., extracts a sample of cities from a list of patients:

```
import Vec  -- Granule's standard vector library
sampleCities : ∀ n k . N k → (Vec (n+k) Patient) [0..1] → Vec k (String [Public])
sampleCities Z [_] = Nil;
sampleCities (S n) [Cons (Patient [city] [name]) ps] = Cons [city] (sampleCities n [ps])
```

This demonstrates the use of different nested graded modalities. The outer modality declares that the input vector is affine, since we do not necessarily use all its elements, given by an `Interval Nat` modality with `0..1`. The inner modalities provide the security levels of patient information. In the inductive case, we thus get `ps` graded by `0..1` and by flattening `city` and `name` are graded by products `(0..1, Public)` and `(0..1, Private)` respectively. We can thus safely collect the cities and output a list of public city names in our database. Let us see what happens when we try to accumulate the private name fields into a list of public data, e.g.,:

```
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: 3:55: `Private` value cannot be moved to level `Public`

[3] https://github.com/granule-project/granule/blob/icfp19/StdLib

Session types. 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 type primitives. With graded modal types and linearity we can express novel communication properties not supported by existing session type approaches. Granule's builtin library provides channel primitives, where `Session` is a trivial graded possibility modality for capturing communication effects:

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

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 s`. The `fork` 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 → () <Session>
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) <Session>
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` takes a channel which it uses exactly `n` times to send each element of the input vector. Dually, `recvVec` takes a size `n` and a channel which it uses `n` times to receive values of `a`, collecting these into an output vector of size `n`. We can then compose these processes using `fork`:

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

This is not the only approach to session types in a linear calculus. Caires and Pfenning [2010] instead view linear logic propositions as directly representing session types, which would be interesting to explore within the setting of Granule in the future.

9 RELATED WORK

Graded types. Bounded Linear Logic can be considered a graded modal system, generalising linear logic's exponential modality ! to a natural-number graded necessity with polynomial index expressions [Girard et al. 1992], where $!_x A$ means A can be used at most x times. Our graded modalities over Interval Nat provide the same reasoning power but also allow lower and upper bounds on reuse, with general arithmetic expressions over variables. Type checking is undecidable for us in general, but so far this has not 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 program properties by changing or generalising the indices of the modality. For example, Dal Lago and Gaboardi gave 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 is a

special case of the kind of general indexed typing Granule can exploit. 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 our approach is more general, capturing different modalities, polymorphism, and pattern matching.

At the same time as specialised efforts to build on 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 BLL by Ghica and Smith [2014] and Brunel et al. [2014]. Each system has essentially the same structure, with a categorical semantics in terms of a graded exponential comonad. In each, the type system is parameterised by a coeffect semiring capturing how a program depends on its context by tracking a particular notion of variable use and thus dataflow. Brunel et al. directly generalise BLL, replacing natural numbers indices with an arbitrary semiring, providing graded necessity similar to here. In Petricek et al. and Ghica et al., the modalities are implicit, with semiring elements associated to each variable binding (and annotating a function arrow), but the systems have essentially the same expressivity. The categorical foundations of graded exponential comonads have since been studied in more depth [Breuvart and Pagani 2015; Katsumata 2018]. Our work focusses more on program properties captured by graded modal types, when combined with standard programming language features and linearity.

Dual to graded comonads is the notion of *graded monads* which arose in the literature around the same time, generalising monads to an indexed family of functors 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, graded monads capture fine-grained information about side effects. This information can then be used to specialise categorical models. Gaboardi et al. [2016] considered *graded distributive laws* for interacting graded monads and graded comonads. Integrating and extending this in Granule is interesting further work, and may provide a way to resolve interactions between linearity and exceptions (e.g., `IOExcept` in the graded possibility here).

Linear Haskell (LH). Recent work retrofits 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 non-linearity in LH is introduced via a consumption multiplicity on function types, dubbed *linearity on the arrow* (akin to implicit coeffects [Petricek et al. 2014]) instead of a graded modality as in Granule. Without explicit modalities, parametric polymorphism cannot provide the linearity polymorphism we get in Granule. LH works around this in several ways.

Compare 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 (right). As in the LH presentation, we elide the `IOMode` parameter which in Granule also indexes the `Handle` type (§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>
```

A monad type $\texttt{IO}_\texttt{L}$ parameterised by the multiplicity of its result is necessary in LH to propagate linearity information. The results of $\texttt{open}_\texttt{LH}$ and $\texttt{close}_\texttt{LH}$ are linear and unrestricted respectively: multiplicity 1 and ω. By virtue of having one linear function arrow and graded modalities, Granule need not index the monad by linearity information; this can be expressed within its type parameter.

LH includes *multiplicity polymorphism* to parameterise over the degree of nonlinearity in functions (akin to our polymorphic grades). For example `map`, which in LH has the following two incompatible types (on the left), can be given a most general type parameterised over multiplicity `p` (right-top). Contrast this with the type of `map` in Granule (right-bottom). Multiplicity polymorphism is subsumed by standard type polymorphism in Granule since `a` and `b` can be instantiated with graded modal types to capture different modes of (non)linearity in the parameter function.

```
map :: (a ⟶ b) ⟶ [a] ⟶ [b]          map :: forall p a b . (a [p] ⟶ b) ⟶ [a] ⟶ [b]
map :: (a ⊸ b) ⟶ [a] ⊸ [b]          map : forall a b . (a ⟶ b) ⟶ [a] ⟶ [b]
```

```
(a ⊸ b) → List a ⊸ List b
(a → b) → List a → List b
```

Multiplicity monomorphic types of `map` in LH.

```
∀ (p :: Multiplicity). (a →p b) → List a →p List b
```

Multiplicity polymorphic type of `map` in LH.

```
(a → b) [] → List a → List b
```

Parametrically polymorphic type of `map` in Granule.

The Granule type is also more precise in that it expresses that the list spine is consumed linearly. Furthermore, since Granule's grades (multiplicities) are first-class members of the type language, we can combine grading with indexed types to give an even more precise specification for `map` as `∀ a b : Type, n : Nat . (a → b) [n] → Vec n a → Vec n b`. We argue that graded modalities are a useful carrier for (non)linearity information and Bernardy et al. [2017] indeed show an encoding via GADTs of a modality equivalent to our `[]` box, which becomes necessary in LH, e.g. when passing unrestricted values through linear functions. It remains to be seen whether the idioms that LH introduces, some of which resonate with our ideas, will transfer to mainstream Haskell.

Kind-based linearity. One approach to structuring a linear type system is to split types into kinds of linear and unrestricted values [Wadler 1990], where kind polymorphism can be used to make code reusable between kinds [Mazurak et al. 2010; Tov and Pucella 2011]. Such systems tend to make a choice about how much linearity to support in the resource-sensitive world: Mazurak et al. choose a fully linear system for F° while Tov and Pucella adopt affine linearity. In Granule, we can freely choose either affine or standard linearity via our graded modalities.

Since linearity is core to Granule, we compare briefly with Alms [Tov and Pucella 2011], an affine linear language in the style of ML (with side effects). Recall the following from Section 2.3:

```
fromMaybe : ∀ {t : Type} . t [0..1] → Maybe t → t
```

The type explains the *local* usage of inputs: `0..1` for the first parameter and linear for the second. The dataflow information captured in the types is from the perspective of the callee, rather than the call*er*. Promotion then connects demands of a function with capabilities at the call site, e.g., for **`let`** `f = [fromMaybe [x]]` **`in`** `e` where `e` uses the `f` function `n` times, then `x` must have grading `0..n`. Contrast this with the type of the analogous function in Alms [Tov and Pucella 2011, p.9]:

$$\texttt{default} : \forall \hat{\alpha}.\ \hat{\alpha} \xrightarrow{\mathbf{U}} \hat{\alpha}\ \texttt{option} \xrightarrow{\langle \hat{\alpha} \rangle} \hat{\alpha}$$

This type says that given a parameter of type $\hat{\alpha}$, return a function which can be used in an affine or unrestricted manner depending on whether the first parameter is affine or unrestricted. Thus, Alm's kind-based approach views linearity from the caller's perspective, contrasting with Granule's local, callee perspective. Subsequently, the Alms type does not explain/restrict the consumption of the second parameter: the erroneous implementation that always returns the default argument also satisfies this type. In Granule, `fromMaybe` has only one well-typed implementation—the correct one.

One might wonder whether we need multiple Granule implementations/types of `fromMaybe` to account for different cases of non-linearity associated with arguments. However, only one definition is needed (the one given in this paper). The capabilities of any arguments are connected to requirements of a function by promotion at an application site. For example, the following two snippets show `fromMaybe` used in the context of concrete arguments which are unrestricted:

```
mb : ∀ t. t [0..∞] → Maybe t → t
mb [d] m = fromMaybe [d] m
```

```
mb' : ∀ t. t [0..1] → (Maybe t) [0..∞] → t
mb' d [m] = fromMaybe d m
```

The left unboxes an unrestricted capability `0..∞`, reboxing to carve out the capability of `0..1` for `d`. The right takes an unrestricted second argument, from which it carves out the linear use (`1..1`).

Tov and Pucella [2011] state "unlimited values are the common case". With Granule we instead explore a system where linearity is the default, but where we add non-linearity requirements as

needed, and more program properties besides. Furthermore, consumption can depend on other inputs, as captured, e.g., in the type of `rep` (§2), providing strong reasoning principles.

Several other languages have type systems based on ideas from linear logic, though space does not permit a proper comparison. Quill [Morris 2016] leverages quantified types to provide an expressive, kind-polymorphic linear system. ATS [Xi 2003; Zhu and Xi 2005] focusses on combining with dependent types and theorem proving, but avoids polymorphism over linear values. Clean [Brus et al. 1987] on the other hand aims to be more of a general-purpose language, based on uniqueness types. Rust [Matsakis and Klock II 2014] incorporates a static view of *ownership* and *borrowing* for references, matching the notion here that some data should not be copied or propagated arbitrarily. Whilst Granule is linear, our aim was not to just produce a linear language but to explore program reasoning with graded modal types, for which linearity is a useful basis.

10 FURTHER WORK AND CONCLUSIONS

Expressivity and dependent types. A more flexible system could be provided by arbitrary rank quantification rather than the ML-style type schemes that Granule features, e.g. via the recent bidirectional results of Dunfield and Krishnaswami [2019]. Even more attractive is a generalisation of our system to full Martin-Löf-style dependent types. Combining linear and dependent types with full generality has been a long-standing challenge. Various attempts settle 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 [2016], refined by 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 grades but in an implicit style. Usage at the type-level is accounted for by 0 of a semiring and term-level use is tracked similarly to coeffect types. We are investigating a similar approach, extending graded modalities to also track dependent type-level usage. Further work is to leverage dependent types to allow user-defined resource algebras and modalities, providing an internally extendable system.

An orthogonal direction for extending Granule's expressivity is to restrict the structural rule of *exchange*, allowing stronger program properties to be enforced, e.g., that *map* on a list preserves the order of elements. Further work is to investigate controlling exchange via an augmented resource algebra to allow graded modalities to track and control its use.

Usability and implicit grading. Coeffect analyses in Granule are first-class via the graded box modalities. As seen in our examples, this approach requires the programmer to sometimes promote and unbox values. Currently we view Granule more as a language for experimenting with this approach. Implicit-style systems such as those of Bernardy et al. [2017]; Petricek et al. [2014] are superficially more user-friendly since there is no explicit (un)boxing. Further work is to explore whether term-level boxing and unboxing can be inferred and thus omitted from the source language.

Conclusion. There has been a flurry of recent work on graded and quantitative types. Granule aims to take a step forward by taking seriously the role of (indexed) data types, pattern matching, polymorphism, and multi-modalities for real programs. There are still open questions about how to make such programs user-friendly or sufficiently flexible for general-purpose programming. At the very least, a language like Granule is useful for developing parts of a program that need significant verification, for which a trade-off in flexibility is worth taking. Furthermore, Granule could be used as a core language, into which a more user-friendly surface-level language is compiled.

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 Granule will be a useful research vehicle for new ideas in type-based program verification.

ACKNOWLEDGMENTS

We thank Frank Pfenning and the anonymous reviewers for their helpful comments; any remaining errors or infelicities are our own. Thanks to the many people who heard us talk about this work and gave useful feedback, especially the participants of TLLA 2017 where the kernel of the idea for this work was initially presented [Orchard and Liepelt 2017]. The first author was supported in part by EPSRC project EP/M026124/1.

Dominic would like to dedicate this paper in love to his late mother and newborn son, whose last and first breaths respectively bracketed this work.

REFERENCES

Samson Abramsky. 1993. Computational interpretations of linear logic. *Theoretical computer science* 111, 1-2 (1993), 3–57.

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.

Robert Atkey and James Wood. 2018. Context Constrained Computation. In *Workshop on Type-Driven Development, colocated with ICFP 2018*.

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

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

P. Nick 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.

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*. 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

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.

Nils Anders Danielsson. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 133–144.

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.

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 *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, New York, NY, USA, 281–292. https://doi.org/10.1145/964001.964025

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. 2016. 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

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.

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).

Futoshi Iwama, Atsushi Igarashi, and Naoki Kobayashi. 2006. Resource usage analysis for a functional language with exceptions. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM)*. ACM, 38–47.

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 *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/2676726.2676969

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 *6th Conference on Algebra and Coalgebra in Computer Science (CALCO 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

Dominic Orchard and Vilem-Benjamin Liepelt. 2017. Gram: A linear functional language with graded modal (extended abstract). In *Workshop on Trends in Linear Logic and Applications (TLLA)*.

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. *SIGPLAN Not.* 50, 12, 177–188. https://doi.org/10.1145/2887747.2804309

Dag Prawitz. 1965. Natural Deduction: A proof-theoretical study. *Stockholm Studies in Philosophy. Almqvist & Wiksell, Stockholm* 3 (1965).

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.

Jesse A Tov and Riccardo Pucella. 2010. Stateful contracts for affine types. In *European Symposium on Programming*. Springer, 550–569.

Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 447–458. https://doi.org/10.1145/1926385.1926436

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

Philip Wadler. 1992. There's no substitute for linear logic. In *8th International Workshop on the Mathematical Foundations of Programming Semantics*.

Philip Wadler. 1993. A Syntax for Linear Logic. In *Mathematical Foundations of Programming Semantics, 9th International Conference, New Orleans, LA, USA, April 7-10, 1993, Proceedings*. 513–529. https://doi.org/10.1007/3-540-58027-1_24

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.

.