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

**ACM Reference Format:**
Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2018. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 55 pages.

## 1 INTRODUCTION

Most programming languages (and type systems) take the view that data is unrestricted: it can be replicated, discarded, and used without constraint. However, this view is naïve and can thus lead to software errors. For example, some data is subject to confidentiality requirements and therefore should not be copied arbitrarily. Some data acts a proxy for an external resource (e.g., a socket, hardware device, or file) and therefore is sensitive to the order of operations applied to it. Dually, some programs are sensitive to the size of their inputs, with non-functional aspects (e.g., execution time) dependent on data. Thus, the reality is that some data act *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 notions of resource and dataflow property. Granule is based on a combination of linear types, indexed types (lightweight dependent types), and the recent notion of *graded modal types* to enable novel quantitative reasoning.

Linear types, in their strictest sense, treat data like a physical resource which must be used once, and then never again [Girard 1987; Wadler 1990]. For example, we can type the identity function as it binds a variable, then uses it, whereas the K combinator $\lambda x.\lambda y.x$ is not linearly typed as $y$ is never used. To overcome this restriction, linear logic provides a modal operator ! which captures and tracks non-linear, unconstrained values. This provides a binary view: either values are linear (like a resource) or non-linear (like the traditional view of data). However, in programming, non-linearity rapidly permeates programs. Bounded Linear Logic (BLL) instead provides a more fine-grained view, replacing ! with an family of modal operators indexed by terms capturing 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 then manipulate these indices, accounting for contraction, weakening, and composition.

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.

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

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

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 a possibility modality [Benton et al. 1998]) to a monoid-indexed form, describing side-effects similarly to *effect systems.* In general, a graded modality provides an indexed family of operators with structure over the indices witnessing proof/type rules. Through the Curry-Howard lens, graded modal types carry information about the structure of programs, and in concert with a suitably expressive type system, provide a mechanism for specifying and verifying properties of programs 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 flavours: graded comonads/necessity and graded monads/possibility. We focus mainly on the graded comonadic part, 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. In particular, we make the following contributions:

- We define a novel type theory, combining graded modal types with linear, polymorphic, and indexed types (§3, §4). Our work is the first to allow multiple different graded modalities to be used at the same time along with user-defined indexed types in a linear language. We provide an operational model (§6) and meta-theoretic results (§7).
- Based on this theory, we present Granule: a statically-typed, functional language, syntactically resembling Haskell. Its novel reasoning powers are demonstrated through various examples (§2, §8). We show graded modalities for tracking fine-grained non-linearity, privacy, stateful protocols (including files and sessions), and their combinations.
- We give a notion of coeffects for pattern matching and novel constructions on graded modal types, including for approximations and combining graded modalities.
- We provide a bidirectional type checking algorithm for our system (§5) which is at the heart of our implementation. This involves interaction between standard type-checking techniques and an SMT solver, discharging theorems from equations over the indices of graded modalities.

The concluding sections of this paper discuss related work in detail and next steps for taking this work forward to provide a new generation of resource-aware programming languages. At the moment, Granule is not designed as a general-purpose 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 (generalised) algebraic data types and pattern matching.

## 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 examples where 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      flip : ∀ {a b c : Type} . (a → b → c) → b → a → c
id x = x                       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 → ()      ✗ copy : ∀ {t : Type} . t → (t, t)
  drop x = ()                         copy x = (x, x)
```

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

```
Linearity error: 2:1:                Linearity error: 2:10:
Linear variable x is never used.     Linear variable x is used more than once.
```

Thus we see that Granule's type system is not just an extension of standard polymorphic systems like Haskell or OCaml. The quantification `∀ {t : Type}` ranges over types which may be resources that are subject to consumption constraints: we cannot simply assume that any `t` is droppable or copiable. Having strict linearity as the default fits the rule that *the more polymorphic our inputs, the less we can assume about them.* Linearity can be employed to enforce some stateful protocols, e.g. for file handling [Walker 2005] and networking primitives [Tov and Pucella 2010], but also to treat pure data as a resource, affording us new ways of reasoning about code.

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

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

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>
```

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

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 akin to Haskell's `IO` monad [Jones 2003]. We explain `<IO>` more later (§2.5) as it approximates a more fine-grained graded modality. We now give two programs using Granule's notation for sequencing effectful computations akin to Haskell "do" notation:

```
twoChars : (Char, Char) <IO>                  bad : Char <IO>
twoChars = let                                bad = let
  h ← openHandle ReadMode "somefile";           h1 ← openHandle ReadMode "somefile";
  (h, c1) ← readChar h;                     ✗   h2 ← openHandle ReadMode "another";
  (h, c2) ← readChar h;                         () ← closeHandle h1;
  () ← closeHandle h                            (h1', c) ← readChar h1
  in pure (c1, c2)                              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: `h1` is used more than once and `h2` and `h1'` 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. Linear logic [Girard 1987] answers this by using a modal type constructor ! to relax linearity and propagate the 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 [] → ()        copy' : ∀ {t : Type} . t [] → (t, t)
drop' [x] = ()                          copy' [x] = (x, x)
```

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

To track fine-grained resource information, modalities in Granule are *graded* by elements of a *resource algebra* whose 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] → ()       copy'' : ∀ {t : Type} . t [2] → (t, t)
drop'' [x] = ()                          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. Parameterised data constructors respect linearity because each argument appears exactly once in the result. From now on we omit unambiguous kinds in quantifications. Consider the following data type.

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

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

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

```
fromMaybe : ∀ t. t [0..1] → Maybe t → t          fromMaybe' : ∀ t. t → Maybe t → t
fromMaybe [_] (Some x) = x;                     ✗ fromMaybe' _ (Some x) = x;
fromMaybe [d] None     = d                        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 (affine) and lower (relevant) 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.

The type of `fromMaybe` explains that the first parameter is used either 0 or 1 times (affine linearity) depending on control flow, and the second parameter is used linearly—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 maybe 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          data N (n : Nat) where
  Nil : Vec 0 a;                                Z : N 0;
  Cons : a → Vec n a → Vec (n + 1) a            S : N n → N (n + 1)
```

When defining operations over `Vec`, we notice that some functions have the usual type signatures we know from non-linear languages, such as append:

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

Indexed types ensure that the length of the output list is indeed the sum of the length of the inputs. Due to linearity our type guarantees more: *every element from the inputs must appear in the output*, which this type does not guarantee in a non-linear language.

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

```
length : ∀ t, n. Vec n (t [0]) → N n      length' : ∀ t, n. Vec n t → (N n, Vec n t)
length Nil = Z;                            length' Nil = (Z, Nil);
length (Cons [_] xs) = S (length xs)       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        sub : ∀ m n. {m ⩾ n} ⇒ N m → N n → N (m - n)
rep Z [t] = Nil;                          sub m Z = m;
rep (S n) [t] = Cons t (rep n [t])        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 an output vector `Vec n t`. Subsequently, 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 list.

On the right, `sub` defines subtraction on indexed naturals, demonstrating Granule's support for preconditions (refinements) in the context of type schemes (before ⇒). These must hold where the function is used. Such predicates are discharged by the external solver. If we include 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 guard for equation of sub is impossible.
Its condition 0 > n_{0.11} + 1 is unsatisfiable.
```

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

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

The type says that for a target length `m` and an input list with a lesser or equal length `n`, we consume the padding element of type `t` exactly `m - n` times to produce an output list 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 list (hence we use lists 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 list appears exactly once in the output and that the padding element is used to pad exactly `m - n` times.

The type of `leftPad` in Granule is superficially similar to what we could write in GHC Haskell or a fully dependently-typed language, except for the non-linearity modality `[m - n]` on the padding element, a minor syntactic addition. However the extra guarantees here mean we get properties for free which we would otherwise have to prove ourselves.

## 2.5 Other graded modalities

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

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

```
secret : Int [Private]                          ✗ main : Int [Public]
secret = [1234]                                   main = hash secret

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

Section 8 shows more examples, including combining security levels and variable usage. Now that we have a taste for Granule, we set out the type system that enables all of these examples. Section 3 describes a core simply-typed 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 the Granule language, called GrMini, which comprises the linear $\lambda$-calculus extended with a graded comonad, resembling the coeffect calculi of Brunel et al. [2014] and Gaboardi et al. [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 typing rules.

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

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

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

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

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

Contexts are either empty $\emptyset$, or can be extended with a linear variable assumption $x : A$ or a *graded assumption* $x : [A]_r$. For a graded assumption, $x$ can behave non-linearly, with substructural behaviour captured by the semiring element $r$, which describes $x$'s use in a term. We will denote the domain of a context $\Gamma$, the set of variables assigned a type in the context, by $|\Gamma|$.

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

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

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

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

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

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:

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

*Dereliction* (DER) converts a linear assumption to be graded, marked with $1$. Subsequently, the semiring element $1$ relates to linearity, though it does not exactly denote linear use as it is not necessarily the case that $x : [A]_1 \vdash t : B$ implies $x : A \vdash t : B$ for all semirings. *Promotion* (PR) introduces the graded comonadic type with grade $r$, propagating this grade to the assumptions via scalar multiplication of the context by $r$. In the case of tracking usage, the rule states that to produce the *capability* to reuse a result value $t$ of type $A$ exactly $r$ times requires that all the input requirements for $t$ are provided $r$ times over, hence we multiply the context by $r$.

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

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

The (LET) rule provides elimination for the graded 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 $\lambda$-calculus can be translated into GrMini based on Girard's translation of the simply-typed $\lambda$-calculus into an intuitionistic linear calculus [Girard

1987]. The idea is to replace every intuitionistic arrow $A \rightarrow B$ with $\Box_{\omega} A \multimap B$ for the singleton semiring $\{\omega\}$ and subsequently unbox via **let** in abstraction and promote when applying, e.g.

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

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

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

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

## 4 THE GRANULE TYPE SYSTEM

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

$$\begin{tikzcd} \mathit{Source} \arrow[r, "\mathit{lexer/parser}"] & \mathit{AST} \arrow[r, "\mathit{freshener}"] & \mathit{AST} \arrow[r, "\mathit{bidirectional\ type\ checking}"] & \mathit{Predicate} \arrow[r, "\mathit{SMT\ solver}"] & \mathit{Result} \arrow[lll, bend left] \end{tikzcd}$$

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

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

### 4.1 Syntax

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

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

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

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

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

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

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

$$
\begin{aligned}
A, B, R &::= A \rightarrow B \mid K \mid \alpha \mid A\,B \mid A \,\mathsf{op}\, B \mid \Box_{c} A \mid \Diamond_{\varepsilon} A \mid () && (\textit{types})\\
\kappa &::= \mathsf{Type} \mid \mathsf{Coeffect} \mid \mathsf{Effect} \mid \mathsf{Predicate} \mid \kappa_1 \rightarrow \kappa_2 \mid \uparrow A && (\textit{kinds})\\
\mathsf{op} &::= + \mid * \mid - \mid < \mid > \mid = \mid \neq \mid \sqcup \mid \sqcap && (\textit{type operators})
\end{aligned}
$$

Types comprise function types, type constructors $K$, variables $\alpha$ (and $\beta$), application $A\,B$, binary operators $A \,\mathsf{op}\, B$, graded comonadic types $\Box_c A$, graded monadic types $\Diamond_\varepsilon A$ where $c$ and $\varepsilon$ 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 $\rightarrow$ rather than the traditional $\multimap$ since we use -> (or $\rightarrow$) as a more familiar concrete syntax. Kinds comprise a number of constants categorising types, grades (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 $\uparrow A$.

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

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

These built-in type constructors are separated by their kind (defined via the kinding judgement in Section 4.3) with the first three of kind Type, products $\times$ as kind-polymorphic, 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 Section 10 on future work):

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

where $\overrightarrow{\alpha : \kappa}$ represents a comma-separated sequence of type variables and their kinds. The second syntactic form additionally includes a set of one or more predicates (types of the kind Predicate) which can be used to express theorems which need to be solved implicitly by the type checker (a kind of refinement), as seen in the sub example (§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 modality. Granule instead allows different graded modalities, with different index domains, to be used simultaneously within the same program. The graded modal type $\Box_c A$ captures different modalities, identified by the type of the grade $c$ which is an element of a *resource algebra*: a pre-ordered semiring $(\mathcal{R}, +, 0, \cdot, 1, \sqsubseteq)$ with monotonic multiplication and addition but two differences: the operations may be partial and the additive unit axiom $0 + r = r$ is replaced with $0 + r \sqsubseteq r$. An example of where this is useful is seen below for security levels. We colour operations in blue when referring to a general resource algebra structure. The syntax of grading terms $c$ is given by:

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

The first line of syntax exposes the resource algebra operations, including syntax for (possibly undefined) least-upper bound ($\sqcup$) and greatest-lower bound ($\sqcap$) derived from the pre-order. Grades can include variables $\alpha$, enabling the 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 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] Coeffect type Level captures a resource algebra for a two-point lattice $\{\mathsf{Private} \sqsubseteq \mathsf{Public}\}$ with $+ = \cdot = \sqcup$ (join) of the lattice and $0 = \mathsf{Public}$ and $1 = \mathsf{Private}$. This algebra exploits our weakening of additive units to $0 + r \sqsubseteq r$ as $(\mathsf{Public}{+}r) = (\mathsf{Public} \sqcup r) = \mathsf{Public}$. Recall that + represents contraction (i.e., a split in the dataflow of a value). Therefore, if a value is used publicly, then it must be permitted for public use, even if it used elsewhere privately, thus we have $\mathsf{Public}{+}\mathsf{Private} = \mathsf{Public}$. However, we still want to be able to weaken at any security level, thus $0 = \mathsf{Public}$ for weakening, which can then be approximated to Private to weaken at any level.

**Definition 4.3.** [Intervals] The Interval constructor is a unary constructor of kind Coeffect $\rightarrow$ Coeffect where Interval $R$ is inhabited by pairs of $R$ values giving lower and upper bounds respectively. Thus, 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 corresponding preorder on $R$). Units are defined $0 = 0_R..0_R$ and $1 = 1_R..1_R$ and the operations and pre-order are defined in the usual way for interval arithmetic:

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

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

**Definition 4.4.** [Extended coeffects] For coeffect type $R$, applying the unary constructor Ext $R$ extends the resource algebra with an element $\infty$ (i.e., 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 the same as that of $R$. In the examples of Section 2, we sometimes used coeffects of kind Interval (Ext Nat), where $0..\infty$ captures arbitrary ("Cartesian" usage), providing the ! modality of modal logic. In Granule, the type "`A []`" is an alias for "`A [0..∞]`".

**Definition 4.5.** [Products] Given two 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., $(r, s){+}(r', s') = (r +_R r', s +_R s')$. 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.

Finally, an inter-resource algebra operation "flatten" describes how to sequentially compose two levels of grading, which occurs when we have nested pattern matching on nested graded modalities.

Consider the following example, which takes a value inside two layers of graded modality, pattern matches on both simultaneously, and then uses the value:

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

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

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

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

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

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

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

Whilst the above resource algebras are for graded comonads/necessity, Granule also has another flavour of graded modality: *graded monads/possibility*, written $\Diamond_{\varepsilon} A$. Following the literature (§9), we provide graded possibility indexed by pre-ordered monoids, captured by $(\mathcal{E}, \star, 1, \leq)$ with monotonic $\star$ and analogous syntax to define $\varepsilon$. 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. Other possible graded monads include indexing by natural numbers for cost analysis (as in Danielsson [2008]). We focus here on coeffect modalities, though the system is easily extended with further modalities.

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

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

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

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

The typing rules of GrMini were given in the form $\Gamma \vdash t : A$. Every GrMini rule is a specialisation of the corresponding Gr rule with an empty type variable context, i.e. $\emptyset; \emptyset; \Gamma \vdash t : A$. We first describe the kinding relation (§4.3.1), unification and substitutions (§4.3.2), top-level definitions & GADTs (§4.3.3), typing patterns (§4.3.4), and finally the declarative specification of typing (§4.3.5).

$$\frac{\Sigma \vdash A : \mathsf{Type} \quad \Sigma \vdash B : \mathsf{Type}}{\Sigma \vdash A \to B : \mathsf{Type}}\ \kappa_{\to} \qquad \frac{\Sigma \vdash A : \kappa_1 \to \kappa_2 \quad \Sigma \vdash B : \kappa_1}{\Sigma \vdash A\,B : \kappa_2}\ \kappa\textsc{app} \qquad \frac{\Sigma \vdash A : \kappa \quad \Sigma \vdash B : \kappa}{\Sigma \vdash A \times B : \kappa}\ \kappa\times$$

$$\frac{}{\Sigma, \alpha : \kappa \vdash \alpha : \kappa}\ \kappa\textsc{var} \qquad \frac{}{\Sigma, \alpha :_{\exists} \kappa \vdash \alpha : \kappa}\ \kappa\exists\textsc{var} \qquad \frac{}{\Sigma \vdash () : \mathsf{Type}}\ \kappa() \qquad \frac{}{\Sigma \vdash \mathsf{IO} : \mathsf{Effect}}\ \kappa\textsc{io}$$

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

$$\frac{\begin{array}{c}op \in \{+, *, \sqcap, \sqcup\}\\ \Sigma \vdash R : \mathsf{Coeff} \quad \Sigma \vdash A : \uparrow R \quad \Sigma \vdash B : \uparrow R\end{array}}{\Sigma \vdash A\,op\,B : \uparrow R}\ \kappa_{\mathrm{op1}} \qquad \frac{\begin{array}{c}op \in \{<, >, =, \neq\}\\ \Sigma \vdash R : \mathsf{Coeff} \quad \Sigma \vdash A : \uparrow R \quad \Sigma \vdash B : \uparrow R\end{array}}{\Sigma \vdash A\,op\,B : \mathsf{Predicate}}\ \kappa_{\mathrm{op2}}$$

$$\frac{\Sigma \vdash A : \uparrow\mathsf{Nat} \quad \Sigma \vdash B : \uparrow\mathsf{Nat}}{\Sigma \vdash A - B : \uparrow\mathsf{Nat}}\ \kappa_{\mathrm{op}-} \qquad \frac{}{\Sigma \vdash \mathsf{Int} : \mathsf{Type}}\ \kappa\mathbb{Z} \qquad \frac{}{\Sigma \vdash \mathsf{Char} : \mathsf{Type}}\ \kappa\textsc{c} \qquad \frac{}{\Sigma \vdash \mathsf{Nat} : \mathsf{Coeff}}\ \kappa\mathbb{N}$$

$$\frac{}{\Sigma \vdash \mathsf{Level} : \mathsf{Coeff}}\ \kappa_{\mathcal{L}} \qquad \frac{}{\Sigma \vdash \mathsf{Ext} : \mathsf{Coeff} \to \mathsf{Coeff}}\ \kappa_{\infty} \qquad \frac{}{\Sigma \vdash \mathsf{Interval} : \mathsf{Coeff} \to \mathsf{Coeff}}\ \kappa_{..}$$

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

*4.3.1 Kinding.* Kinding is defined in Figure 1, where kinds are assigned to types in an environment of type variables $\Sigma$. The first two rules are standard, giving kinding of function types ($\kappa_{\to}$) and type application ($\kappa\textsc{app}$). 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 Definition 4.5). Type variables are kinded by ($\kappa\textsc{var}$) and ($\kappa\exists\textsc{var}$). Rule ($\kappa\Box$) gives the kinding of the graded modal necessity ($\Box_r A$), where $r$ has a kind which is a promoted coeffect type. Thus, we view coeffect terms $r$ as residing at the type level. Similarly, ($\kappa\Diamond$) gives the kinding for graded modal possibility ($\Diamond_\varepsilon A$) for effects. The next three rules ($\kappa_{\mathrm{op1}}$, $\kappa_{\mathrm{op2}}$, and $\kappa_{\mathrm{op}-}$) give the kinds of type-level binary operators, first capturing resource algebra operations then equations and inequations over coeffect terms in the Predicate kind, and cut-off subtraction for natural numbers. The remaining rules kind the built-in type constructors $K$, including the types for resource algebras (see §4.2). We actually place some restrictions on how constructors can be composed, e.g., Ext (Ext Nat) is not allowed. This requires some stratification, which is elided here for brevity as it is not essential to understanding the rest of the system.

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

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

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

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

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

Fig. 2. Type unification rules

Type unification is given by relation $\Sigma \vdash A \sim B \rhd \theta$ in Figure 2. Unification is essentially a congruence over the structure of types (under a context $\Sigma$), creating substitutions from unification variables to types, e.g. ($U_{\text{VAR}\exists}$) for $\alpha \sim A$ (which has a symmetric counterpart for $A \sim \alpha$ elided here for brevity). Universally quantified variables can be unified with themselves ($U_{\text{VAR}=}$), and also with unification variables via ($U_{\text{VAR}\exists}$). In multi-premise rules, substitutions generated by unifying subterms are then applied to types being unified in later premises, e.g., as in ($U_{\rightarrow}$). Type unification extends to grading terms, which can also contain type variables. We elide the definition here since it is straightforward and follows a similar scheme to the figure.

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

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

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.2) gives the full definition.

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

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

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

#### 4.3.3 *Top-level definitions & indexed types.*

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

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

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

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

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

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

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

For example, with Cons above we get:

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

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

*4.3.4 Pattern matching.* Pattern matching, and its interaction with linearity and grading, is one of Granule's novelties. Patterns must be typed, and patterns themselves can incur consumption that is witnessed in types. The declarative specification of typing for patterns is similar to other functional languages (e.g., ML [Milner et al. 1997]) but takes into account grading and graded modalities. Patterns are typed by judgments of the form:

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

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

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

The pattern match $[x]$ creates a graded assumption $x : [\alpha]_2$ in the context of the body. Thus, when checking a box pattern we push the grade of its associated graded modal type down to the inner patterns. This information is captured in the typing rules by optional coeffect information:

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

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

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

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

The typing of box patterns 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 $\Delta$ graded by $r : R$. Note that this rule itself applies only in a context *not* enclosed by another box pattern.

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

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

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

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

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

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

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

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

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

Thus for pattern matching on some constructor $C$ for the data type $K$, we instantiate its polymorphic type (see Def 4.7) in the first line of premises, getting $\theta(K A_0 \ldots A_m)$ with unification variables $\Sigma'$ and renamed coercion $\theta'$. We then unify $\theta(K A_0 \ldots A_m)$ with the expected type $A$ providing another substitution $\theta'$. We now have a way to rewrite the types of the constructor's parameters $B_i$ using the coercion $\theta'_\kappa$, the instantiation substitution $\theta'$, and the substitution $\theta$ coming from unifying the type of pattern with the constructor's result type $K A_0 \ldots A_m$. Thus we check each inner pattern $p_i$ against the types $(\theta'_\kappa \uplus \theta' \uplus \theta)B_i$, yielding their own bindings $\Gamma_i$ and substitutions $\theta_i$ which we collect for the result of pattern matching.

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

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

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

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

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

$$\frac{\Sigma \vdash A : \mathsf{Type}}{D; \Sigma; x : A \vdash x : A}\ \textsc{var} \qquad \frac{D; \Sigma; - \vdash p : A \rhd \Delta; \theta \qquad D; \Sigma; \Gamma, \Delta \vdash t : \theta B \qquad \mathsf{irrefutable}\ p}{D; \Sigma; \Gamma \vdash \lambda p.t : A \to B}\ \textsc{abs} \qquad \frac{D; \Sigma; \Gamma_1 \vdash t_1 : A \to B \qquad D; \Sigma; \Gamma_2 \vdash t_2 : A}{D; \Sigma; \Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \textsc{app}$$

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

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

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

On the left, the universally quantified variables are instantiated, providing an environment $\Sigma'$ of fresh instance variables for each $\alpha$ and a substitution $\theta$ mapping the universal variables to their instance counterparts. As described in Section 4.3.3, we also have a coercion $\theta_\kappa$ for indexed types, which gets instantiated to $\theta'_\kappa$. Thus, the compound 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 $(\theta B)$, i.e., the predicate must hold at the usage site. There could be many such predicates stated with the type, thus this rule generalises in the expected way to a set of predicates $A_1, \ldots, A_n$. Granule does not yet support having predicates in data constructors, but this is an easy extension.

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

$$\frac{\Sigma \vdash R : \mathsf{Coeff} \qquad D; \Sigma; \Gamma \vdash t : A}{D; \Sigma; \Gamma + [\Delta]_{0:R} \vdash t : A}\ \textsc{weak} \qquad \frac{\Sigma \vdash R : \mathsf{Coeff} \qquad D; \Sigma; \Gamma, x : A \vdash t : B}{D; \Sigma; \Gamma, x : [A]_{1:R} \vdash t : B}\ \textsc{der} \qquad \frac{\Sigma \vdash r : \uparrow R \qquad \Sigma \vdash R : \mathsf{Coeff} \qquad D; \Sigma; [\Gamma] \vdash t : A}{D; \Sigma; r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}$$

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 $\Gamma$ 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 graded context $\Gamma$ and a semiring element $r \in R$ then $\Gamma$ can be multiplied by $r$ as follows:

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

On the right, the graded assumption $s : S$ may have a different type $S$ to the promoting grade's type $R$. The function $R \sqcup S \rhd R'; \iota_1; \iota_2$ (given in Appendix A.2, Def. A.4) computes the least upper-bound type of $R$ and $S$ which is $R'$, generating also 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 $r$ and then multiplying with $(r_1, s_1)$, yielding $(r \cdot r_1, s_1)$.

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

$$\frac{\begin{array}{ll} D;\Sigma;- \vdash p : A \rhd \Delta;\theta & \text{irrefutable } p \\ D;\Sigma;\Gamma_1 \vdash t_1 : \Diamond_{\varepsilon_1} A & D;\Sigma;\Gamma_2,\Delta \vdash t_2 : \Diamond_{\varepsilon_2} \theta B \end{array}}{D;\Sigma;\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ \langle p \rangle \leftarrow t_1\ \mathbf{in}\ t_2 : \Diamond_{(\varepsilon_1 \star \varepsilon_2)} B}\ \text{LET}\Diamond \qquad \frac{D;\Sigma;\Gamma \vdash t : A}{D;\Sigma;\Gamma \vdash \langle t \rangle : \Diamond_1 A}\ \text{PURE}$$

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

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

For example, for the security levels modality, if we have $x : [A]_{\mathsf{Private}} \vdash t : A$ then we can conclude $x : [A]_{\mathsf{Public}} \vdash t : A$ since $\mathsf{Private} \sqsubseteq \mathsf{Public}$. 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 for graded necessity modalities during type checking is often confusing for the programmer. Instead, in the implementation of Granule, we allow approximation only at the level of a function definition where any approximation is made explicit by the signature.

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

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

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 also include a predicate, i.e., of the form $\forall \overrightarrow{\alpha : \kappa} . \{A_1, .., A_n\} \Rightarrow B$. We show the typing of equations with such predicates in the algorithmic definition of the type system next, where we explicitly represent predicates and theorems generated by type checking.

Note that multiple different equations correspond to different control-flow branches and as such may have different gradings. Thus the approximation rules may be applied in order to calculate the least-upper bound gradings across the equations.

## 5 BIDIRECTIONAL TYPE CHECKING ALGORITHM

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

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

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

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

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

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

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

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

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

Much of the definitions of checking and synthesis are a detailed 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 create a local scope. We define pattern type checking via the judgment:

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

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

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

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

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

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

The first premise of promotion ($\Leftarrow_{\text{PR}}$) employs a function mapping contexts into graded contexts by dereliction of linear assumptions (see Def. A.5). This is applied to the subcontext of $\Gamma$ whose variables appear free in $t$, giving the input context of graded assumptions $\Gamma'$ for checking the subterm $t$. The context $\Delta$ resulting from checking $t$ is then scalar multiplied by $r$.

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

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

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

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

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

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

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

## 6 OPERATIONAL SEMANTICS

Once type checking shows a program to be well-typed, its AST is interpreted, following a call-by-values semantics. CBV was chosen for simplicity, though exploring other choices (e.g., call-by-push-value) is interesting further work. 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 (provided by Haskell) such as arithmetic, file handling, and concurrency, elided here.

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

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

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:

$$\frac{}{(v \rhd \_)t = t} \rhd_{\_} \quad \frac{}{(v \rhd x)t = [v/x]t} \rhd_{\text{VAR}} \quad \frac{}{(n \rhd n)t = t} \rhd_{\text{INT}} \quad \frac{(v \rhd p)t = t'}{([v] \rhd [p])t = t'} \rhd_{\Box}$$

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

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

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \textsc{App}_L \qquad \frac{t_2 \rightsquigarrow t_2'}{v\ t_2 \rightsquigarrow v\ t_2'}\ \textsc{App}_R \qquad \frac{}{(\lambda p.t)\ v \rightsquigarrow (v \rhd p)t}\ \textsc{P}\beta \qquad \frac{}{\mathbf{let}\ \langle p\rangle \leftarrow \langle v\rangle\ \mathbf{in}\ t_2 \rightsquigarrow (v \rhd p)t_2}\ \textsc{Let}\beta$$

$$\frac{t_1 \rightsquigarrow t_1'}{\mathbf{let}\ \langle p\rangle \leftarrow t_1\ \mathbf{in}\ t_2 \rightsquigarrow \mathbf{let}\ \langle p\rangle \leftarrow t_1'\ \mathbf{in}\ t_2}\ \textsc{Let}_1 \qquad \frac{t \rightsquigarrow t'}{\mathbf{let}\ \langle p\rangle \leftarrow \langle t\rangle\ \mathbf{in}\ t_2 \rightsquigarrow \mathbf{let}\ \langle p\rangle \leftarrow \langle t'\rangle\ \mathbf{in}\ t_2}\ \textsc{Let}_2 \qquad \frac{t \rightsquigarrow t'}{[t] \rightsquigarrow [t']}\ \Box$$

## 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 Pravitz in a modal context [Prawitz 1965]). Wadler demonstrates this problem with the following example:

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

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 type, i.e., $g : [\Box(\Box B \to C)]_1, y : [\Box B]_1 \vdash [\mathbf{let}\ [h] = g\ \mathbf{in}\ h\ y] : \Box C$. 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 $\beta$-reduction):

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

Via 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.

Note, some graded modalities (e.g. for Nat) permit an isomorphism $\Box_r(\Box_s A) \cong \Box_{(r \cdot s)}A$ enabled by double unboxing, and thus depending on flatten; this is not derivable for all graded modalities.

Well-typed substitution generalises to arbitrary patterns in Granule as follows:

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

**Lemma 7.4.** [Graded pattern type safety] For all patterns $p$ where irrefutable $p$ with $D; \Sigma; r : R \vdash p : A \rhd \Gamma; \theta$ and for values $v$ with $D; \Sigma; [\Gamma_2] \vdash v : A$ and terms $t$ depending on the bindings of

$p$ such that $D; \Sigma; \Gamma_1, \Gamma \vdash t : \theta B$ then there exists a term $t'$ such that $(v \triangleright p)t = t'$ (progress) and $D; \Sigma; \Gamma_1 + r \cdot \Gamma_2 \vdash t' : \theta B$ (preservation).

Combining these results, we get progress and type preservation:

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

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

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

The ordering $\Gamma' \sqsubseteq \Gamma$ means that as reduction proceeds, grading in the context can become more precise. For example, with interval grades over Nat, we have that $(0..1) \sqsubseteq (0..\infty)$. Subsequently, a reduction from $\Gamma, x : [A]_{0..\infty} \vdash t : A$ to $\Gamma, x : [A]_{0..1} \vdash t' : A$ would fit the form of the lemma.

## 8 FURTHER EXAMPLES

*Interaction with data structures.* Graded modalities can be commuted with other data types, for example to pull information about consumption of sub-parts of data up to the whole, or dually to push capabilities for consumption 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 operational intuition for `pull`, when specialised to `Nat` grades, is that making `c` copies of a pair requires `c` copies of each component (a "deep" copying). 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. For example, the safe head function on vectors can be defined as:

```
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.

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

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 modalities, consider a data type for storing patient information of different privacy levels:

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

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

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

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

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

The Granule interpreter gives the following type error:

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

*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 Com is a trivial graded possibility modality for capturing communication effects:

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

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

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

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

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

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

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

## 9 RELATED WORK

*Grading and coeffects.* Bounded Linear Logic can be considered an early graded modal system, generalising linear logic's exponential modality ! into a family of modalities indexed by natural-number polynomials [Girard et al. 1992]. In BLL, $!_x A$ then means that $A$ can be used at most $x$ times. Our graded modalities over Interval Nat provide the same power as BLL and more, allowing lower and upper bounds on reuse, with arbitrary polynomial 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 we generalise our approach considerably to capture 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 Bounded Linear Logic by Ghica and Smith [2014] and Brunel et al. [2014]. Each system has essentially the same structure, with a categorical semantics captured by a graded exponential comonad. Each system tracks quantitative properties by instantiating a type system with a semiring. In all of these approaches, coeffects capture how a program depends on its context by tracking a particular notion of variable usage and therefore dataflow. Brunel et al. directly generalise BLL, replacing natural numbers indices with an arbitrary semiring. In the approach of Petricek et al. and Ghica et al., the modalities are made implicit, with semiring elements associated to each variable binding (and annotating a function arrow), but the systems have essentially the same structure. The categorical foundations of graded exponential comonads have since been studied in more depth [Breuvart and Pagani 2015; Katsumata 2018]. Our approach with Granule is to focus more on program properties captured by graded comonads, when combined with standard programming language features and linearity. We follow the explicit indexed-modal approach of Brunel et al. [2014]; De Amorim et al. [2014].

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

Gaboardi et al. [2016] brought graded monads and graded (exponential) comonads together into a single calculus, exploring interactions between the two structures via *graded distributive laws*. The coming together of these two dual flavours of modalities is also provided by Granule, but exploring their graded distributive laws is further work.

*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).

$$\begin{array}{llll}
\texttt{open}_{\texttt{LH}} & \texttt{:: String} \rightarrow \texttt{IO}_{\texttt{L}}\ \texttt{1 File} & \texttt{open}_{\texttt{Gr}} & \texttt{: String} \rightarrow \texttt{Handle <Open,IOExcept>} \\
\texttt{close}_{\texttt{LH}} & \texttt{:: File} \multimap \texttt{IO}_{\texttt{L}}\ \omega\ \texttt{()} & \texttt{close}_{\texttt{Gr}} & \texttt{: Handle} \rightarrow \texttt{() <Close,IOExcept>}
\end{array}$$

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 $\omega$. 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. The Granule type is also more precise in that it expresses that the list spine itself is consumed linearly.

$$\underbrace{\begin{array}{l}\texttt{(a} \multimap \texttt{b)} \rightarrow \texttt{List a} \multimap \texttt{List b} \\ \texttt{(a} \rightarrow \texttt{b)} \rightarrow \texttt{List a} \rightarrow \texttt{List b}\end{array}}_{\text{Multiplicity monomorphic types of map in LH.}}$$

$$\underbrace{\forall\ \texttt{(p :: Multiplicity). (a} \rightarrow_{\texttt{p}} \texttt{b)} \rightarrow \texttt{List a} \rightarrow_{\texttt{p}} \texttt{List b}}_{\text{Multiplicity polymorphic type of map in LH.}}$$

$$\underbrace{\texttt{(a} \rightarrow \texttt{b) [] } \rightarrow \texttt{List a} \rightarrow \texttt{List b}}_{\text{Parametrically polymorphic type of map in Granule.}}$$

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 argued 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 linear type systems splits 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 $F^\circ$ while Tov and Pucella adopt affine linearity. In Granule, we can freely choose either affine or standard linearity via our graded modalities.

We compare briefly with Alms [Tov and Pucella 2011]. 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 caller. 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' : ∀ t. t [0..1] → (Maybe t) [0..∞] → t
mb [d] m = fromMaybe [d] m             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.

There are several other languages with type systems based on ideas from linear logic. For space reasons we can only name a few here, without providing further comparisons. Quill [Morris 2016] leverages quantified types to provide a system with expressive, kind-polymorphic, and usage constrained programs. ATS [Xi 2003; Zhu and Xi 2005] has a strong focus on combining with dependent types and theorem proving, but avoiding polymorphism over linear values. Clean [Brus et al. 1987] on the other hand aimed to be more of a general purpose language, based on uniqueness types. Rust [Matsakis and Klock II 2014] incorporates a static view of *ownership* and *borrowing* for references, thus acknowledging that some data should not just be copied and propagated arbitrarily. Whilst linearity is a big part of Granule, our aim was not to just produce a linear language but to explore program reasoning with graded modal types, for which linearity provides a useful basis.

## 10 FURTHER WORK AND CONCLUSIONS

*Restricting exchange.* Granule restricts weakening and contraction, but *exchange* remains. However, a system restricting exchange would allow stronger program properties to be proved, e.g., that *map* on a list preserves the order of elements. Further work is to explore controlling exchange via an augmented resource algebra, allowing graded modalities to track the use of exchange.

*Rank-N quantification 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 well has been a long-standing difficulty. Various attempts have settled on the compromise that types can depend only on non-linear values [Barber and Plotkin 1996; Cervesato and Pfenning 2002; Krishnaswami et al. 2015]. Recent work by McBride [McBride 2016], refined by Atkey [Atkey 2018], however resolves the interaction of linear and dependent types by augmenting a linear system with usage annotations capturing the number of times a variable is used computationally, akin to our grades here, but in an implicit style. Usage at the type-level is accounted for by 0 of a semiring structure, and term-level use is tracked in much the same way as BLL. We have made significant progress on a dependent system that extends the graded modalities presented here to also track type-level usage. In this system, graded modalities become first-class and can hence be user-defined. Further work is to leverage dependent types to allow user-defined resource algebras and modalities, providing an internally extendable system.

*Usability and implicit grading.* Type-based coeffect analyses in Granule are made explicit via the graded box modalities. As we have seen with our examples, the explicit approach requires the programmer to "box" and "unbox" values rather frequently. Currently we view Granule more as a language for experimenting with this approach. Implicit-style systems such as those of Bernardy et al. [2017]; Ghica and Smith [2014]; Petricek et al. [2014] are superficially more user-friendly since there is no explicit boxing/unboxing. We will 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.

## 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. (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 STM-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.

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 *POPL*, Vol. 39. ACM.

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).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 *POPL*. 447–458.

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*. Citeseer.

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

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

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

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

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

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

## A COLLECTED TYPING RULES

### A.1 Declarative system

*Pattern typing (declarative)* $\boxed{D;\Sigma;r\,{:}?R \vdash\ p : A \rhd \Gamma;\theta}$

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

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

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

$$\frac{\begin{array}{c} 0 \sqsubseteq r \\ \Sigma \vdash A : \mathsf{Type} \end{array}}{D;\Sigma;r : R \vdash\ \_ : A \rhd \emptyset;\emptyset}\ [\textsc{p\_}]$$

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

$$\frac{\begin{array}{c} (C : (\forall \overrightarrow{\alpha : \kappa}\ .\ B_0 \to \ldots \to B_n \to K A_0 \ldots A_m, \theta_\kappa)) \in D \\ \theta, \Sigma', \theta'_\kappa = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}, \theta_\kappa) \qquad \Sigma, \Sigma' \vdash \theta(K A_0 \ldots A_m) \sim A \rhd \theta' \\ 1 \sqsubseteq r \qquad D;\Sigma,\Sigma';r : R \vdash\ p_i : (\theta'_\kappa \uplus \theta' \uplus \theta)B_i \rhd \Gamma_i;\theta_i \end{array}}{D;\Sigma,\Sigma';r : R \vdash\ C\, p_0 \mathinner{..} p_n : A \rhd \Gamma_0, \mathinner{..}, \Gamma_n; \theta'_\kappa \uplus \theta' \uplus \theta_0 \uplus \ldots \uplus \theta_n}\ [\textsc{pC}]$$

*Typing equations (declarative)* $\boxed{D \vdash Eqn : T}$

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

$$\frac{\begin{array}{c} D;\overrightarrow{\alpha : \kappa};- \vdash\ p_i : B_i \rhd \Delta_i;\theta_i \\ A_1 \land \ldots \land A_n \implies D;\overrightarrow{\alpha : \kappa};\Delta_1, \mathinner{..}, \Delta_n \vdash t : (\theta_1 \uplus \ldots \uplus \theta_n)A \end{array}}{D \vdash x\, p_1 \mathinner{..} p_n = t : \forall \overrightarrow{\alpha : \kappa}\ .\ \{A_1, \mathinner{..}, A_n\} \Rightarrow (B_1 \to \ldots \to B_n \to A)}\ \textsc{eqnP}$$

*Typing expressions (declarative)* $\boxed{D;\Sigma;\Gamma \vdash t : A}$

$$\frac{\Sigma \vdash A : \mathsf{Type}}{D;\Sigma;x : A \vdash x : A}\ \textsc{var} \qquad \frac{\begin{array}{c} D;\Sigma;- \vdash\ p : A \rhd \Delta;\theta \\ D;\Sigma;\Gamma,\Delta \vdash t : \theta B \\ \mathsf{irrefutable}\ p \end{array}}{D;\Sigma;\Gamma \vdash \lambda p.t : A \longrightarrow B}\ \textsc{abs} \qquad \frac{\begin{array}{c} D;\Sigma;\Gamma_1 \vdash t_1 : A \longrightarrow B \\ D;\Sigma;\Gamma_2 \vdash t_2 : A \end{array}}{D;\Sigma;\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \textsc{app}$$

$$\frac{\begin{array}{c} (C : (\forall \overrightarrow{\alpha : \kappa}\ .\ A, \theta_\kappa)) \in D \\ \theta, \Sigma', \theta'_\kappa = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}, \theta_\kappa) \end{array}}{D;\Sigma,\Sigma';\emptyset \vdash C : (\theta'_\kappa \uplus \theta)A}\ \mathrm{C} \qquad \frac{}{D;\Sigma;\emptyset \vdash n : \mathsf{Int}}\ \textsc{int}$$

$$\frac{\begin{array}{c} (x : \forall \overrightarrow{\alpha : \kappa}\ .\ A \Rightarrow B) \in D \\ \theta, \Sigma' = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}) \quad \Sigma \vdash A : \mathsf{Predicate} \quad (\theta A) \end{array}}{D;\Sigma,\Sigma';\emptyset \vdash x : \theta B}\ \textsc{def}$$

$$\frac{\begin{array}{c} \Sigma \vdash R : \mathsf{Coeff} \\ D;\Sigma;\Gamma \vdash t : A \end{array}}{D;\Sigma;\Gamma + [\Delta]_{0:R} \vdash t : A}\ \textsc{weak} \qquad \frac{\begin{array}{c} \Sigma \vdash R : \mathsf{Coeff} \\ D;\Sigma;\Gamma, x : A \vdash t : B \end{array}}{D;\Sigma;\Gamma, x : [A]_{1:R} \vdash t : B}\ \textsc{der} \qquad \frac{\begin{array}{c} \Sigma \vdash r : \uparrow R \qquad \Sigma \vdash R : \mathsf{Coeff} \\ D;\Sigma;[\Gamma] \vdash t : A \end{array}}{D;\Sigma;r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}$$

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

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

## A.2 Auxiliary definitions

**Definition A.1.** [Type substitution] Substitution $\theta$ applied to types $A$:

$$\begin{aligned} \theta(A \longrightarrow B) &= (\theta A) \longrightarrow (\theta B) \\ \theta K &= K \\ \theta\alpha &= \begin{cases} A & \theta(\alpha) = A \\ a & \textit{otherwise} \end{cases} \\ \theta(A\,B) &= (\theta A)\,(\theta B) \\ \theta(A\ \mathsf{op}\ B) &= (\theta A)\ \mathsf{op}\ (\theta B) \\ \theta(\Box_c A) &= \Box_{(\theta c)}(\theta A) \\ \theta(\Box_{c:R} A) &= \Box_{(\theta c):\theta R}(\theta A) \\ \theta(\Diamond_\varepsilon A) &= \Diamond_\varepsilon(\theta A) \end{aligned}$$

Substitution $\theta$ applied to kinds $\kappa$:

$$\begin{aligned} \theta(\uparrow A) &= \uparrow(\theta A) \\ \theta(\kappa_1 \longrightarrow \kappa_2) &= (\theta\kappa_1) \longrightarrow (\theta\kappa_2) \\ \theta\kappa &= \kappa \end{aligned}$$

Substitution $\theta$ applied to the contexts $\Gamma$:

$$\begin{aligned}
\theta\emptyset &= \emptyset \\
\theta(\Gamma, x : A) &= \theta\Gamma, x : \theta A \\
\theta(\Gamma, x : [A]_{c}) &= \theta\Gamma, x : [\theta A]_{(\theta c)} \\
\theta(\Gamma, x : [A]_{c:R}) &= \theta\Gamma, x : [\theta A]_{\theta c:\theta R}
\end{aligned}$$

Substitution $\theta$ applied to type variable environments $\Sigma$:

$$\begin{aligned}
\theta\emptyset &= \emptyset \\
\theta(\Sigma, \alpha : \kappa) &= \theta\Sigma, \alpha : \theta\kappa \\
\theta(\Sigma, \alpha :_{\exists} \kappa) &= \theta\Sigma, \alpha :_{\exists} \theta\kappa
\end{aligned}$$

Substitution $\theta$ applied to substitution $\theta'$:

$$\begin{aligned}
\theta\emptyset &= \emptyset \\
\theta(\theta' \uplus x \mapsto A) &= (\theta\theta') \uplus x \mapsto (\theta A)
\end{aligned}$$

Substitution $\theta$ applied to coeffect term $c$ similarly follows the recursive structure of the coeffect terms, until it hits variables.

**Definition A.2.** [Substitution composition] Given two substitutions $\theta_1$ and $\theta_2$ and a type variable context $\Sigma$ such that $\Sigma \vdash \theta_1$ and $\Sigma \vdash \theta_2$ then we define the context composition $\theta_1 \uplus \theta_2$ as follows, by induction on $\theta_1$:

$$\emptyset \uplus \theta_2 = \theta_2$$

$$(\theta_1, \alpha \mapsto A) \uplus \theta_2 \begin{cases} (\theta_1 \uplus (\theta_2 \setminus x) \uplus \theta), \alpha \mapsto \theta A & \theta_2(\alpha) \;=\; B \wedge \Sigma \vdash A \sim B \triangleright \theta \\ (\theta_1 \uplus \theta_2), \alpha \mapsto A & x \notin \mathrm{dom}(\theta_2) \end{cases}$$

This is a partial operation which may fail if both substitutions contain a substitution for variable $\alpha$ but the two substitutees are not unifiable.

**Definition A.3.** [Substitution compatibility] Given a type-variable context $\Sigma$ and a substitution $\theta$ then we say a substitution is *compatible* with the context if $\Sigma \vdash \theta$, defined:

$$\frac{}{\Sigma \vdash \emptyset}\ \textsc{empty} \qquad \frac{\Sigma \vdash \theta \qquad (\alpha : \kappa) \in \Sigma \qquad \Sigma \setminus \alpha : \kappa \vdash A : \kappa}{\Sigma \vdash \theta \uplus (\alpha \mapsto A)}\ \textsc{ext}$$

Essentially compatibility says that any substitutions are well kinded (i.e., don't change the kind). The premise for calculating well-kindedness for a substituted type $A$ includes that it is well kinded in the context without $\alpha$. This prevents recursive kinds.

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

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

**Definition A.4.** [Upper-bound on coeffect types] Given two coeffect types $R$ and $S$, we can compute their upper-bound by the following function, written $R \sqcup R' \triangleright S; \iota_1; \iota_2$ meaning that the upper bound of $R$ and $R'$ is the coeffect type $S$ with injections $\iota_1 : R \rightarrow S$ and $\iota_2 : R' \rightarrow S$. This function is defined:

$$\begin{array}{ll}
R \sqcup R & \triangleright R; id; id \\
\mathsf{Interval}\, R \sqcup R & \triangleright \mathsf{Interval}\, R; id; r \mapsto r..r \\
R \sqcup \mathsf{Interval}\, R & \triangleright \mathsf{Interval}\, R; r \mapsto r..r; id \\
\mathsf{Nat} \sqcup \mathsf{Ext\,Nat} & \triangleright \mathsf{Ext\,Nat}; \iota_{\subseteq}; id \\
\mathsf{Ext\,Nat} \sqcup \mathsf{Nat} & \triangleright \mathsf{Ext\,Nat}; id; \iota_{\subseteq} \\
(R \times S) \sqcup R & \triangleright R \times S; id; r \mapsto (r, 1) \\
(R \times S) \sqcup S & \triangleright R \times S; id; s \mapsto (1, s) \\
R \sqcup (R \times S) & \triangleright R \times S; r \mapsto (r, 1); id \\
S \sqcup (R \times S) & \triangleright R \times S; s \mapsto (1, s); id \\
R \sqcup S & \triangleright R \times S; r \mapsto (r, 1); s \mapsto (1, s) \qquad \textit{where } R \neq S
\end{array}$$

Note that this is defined as a function to make the ordering between cases clear.

**Lemma A.1.** [Flatten is compatible with $\vee$] For all coeffect terms $r, r'$ and types $R, R', S$ then:

$$(\exists s.\mathsf{flatten}(r, R, r', R') = (s, S)) \Leftrightarrow (\exists \iota_1, \iota_2.R \vee R' \triangleright S; \iota_1; \iota_2)$$

## A.3 Bidirectional checking

### *A.3.1 Checking.*

*Algorithmic type checking* $\boxed{D; \Sigma; \Gamma \vdash t \Leftarrow A; \Delta; \Sigma'; \theta; P}$

$$\frac{\begin{array}{l} D; \Sigma_1; - \vdash p : A \;\triangleright\; \Delta; P; \theta;\; \Sigma_2 \\ \text{irrefutable } p \\ D; \Sigma_2; \Gamma, \Delta \vdash t \Leftarrow \theta B; \Delta'; \Sigma_3; \theta'; P' \end{array}}{D; \Sigma_1; \Gamma \vdash \lambda p.t \Leftarrow A \rightarrow B; \Delta' \backslash \Delta; \Sigma_3; \theta \uplus \theta'; P \wedge P'} \Leftarrow_{\lambda}$$

$$\frac{\begin{array}{l} D; \Sigma_1; \Gamma \vdash t_2 \Rightarrow A; \Delta_2; \Sigma_2; \theta_2; P \\ D; \Sigma_2; \Gamma \vdash t_1 \Leftarrow A \rightarrow B; \Delta_1; \Sigma_3; \theta_1; P' \end{array}}{D; \Sigma_1; \Gamma \vdash t_1\; t_2 \Leftarrow B; \Delta_1 + \Delta_2; \Sigma_3; \theta_1 \uplus \theta_2; P \wedge P'} \Leftarrow_{\text{APP}}$$

$$\frac{\begin{array}{l} \Sigma_1 \vdash [\Gamma \cap \mathsf{FV}(t)]_R \triangleright \Gamma';\; \theta' \\ \Sigma_1 \vdash r : R \\ D; \Sigma_1; \Gamma' \vdash t \Leftarrow A; \Delta; \Sigma_2; \theta; P \end{array}}{D; \Sigma_1; \Gamma \vdash [t] \Leftarrow \Box_r A; r \cdot \Delta; \Sigma_2; \theta \uplus \theta'; P} \Leftarrow_{\text{PR}} \qquad \frac{\begin{array}{l} D; \Sigma; \Gamma \vdash t \Rightarrow B; \Delta; \Sigma'; \theta; P \\ \Sigma \vdash A \sim B \triangleright \theta'; P' \end{array}}{D; \Sigma; \Gamma \vdash t \Leftarrow A; \Delta; \Sigma'; \theta \uplus \theta'; P \wedge P'} \Leftarrow_{\Rightarrow}$$

**Definition A.5.** [Grade a context] Given a context $\Gamma$ we can turn the context into a graded context $\Gamma'$ with grades of type $R$, by a partial operation $\Sigma \vdash [\Gamma]_R \triangleright \Gamma';\; \theta$ which also produces a substitution, where $\Sigma \vdash [\emptyset]_R \triangleright \emptyset;\; \emptyset$ for empty contexts, and:

$$\frac{\Sigma \vdash [\Gamma]_R \triangleright \Gamma';\; \theta}{\Sigma \vdash [\Gamma, x : A]_R \triangleright \Gamma', x : [A]_{1:R};\; \theta} [\text{LIN}] \qquad \frac{\Sigma \vdash [\Gamma]_R \triangleright \Gamma';\; \theta' \qquad \Sigma \vdash s : \uparrow S \qquad \Sigma \vdash R \sqcup S \triangleright R'; \iota_1; \iota_2; \theta}{\Sigma \vdash [\Gamma, x : [A]_s]_R \triangleright \theta(\Gamma', x : [A]_{\iota_2 s});\; \theta \uplus \theta'} [\text{GR}]$$

On the left, a linear assumption is turned into an assumption graded by 1 of type $R$. On the right, a graded assumption with $r : R'$ can be turned into an assumption graded at $r : S$ if and only if $S$ is the least-upper bound coeffect type of $R$ and $R'$. For example, if $x : [A]_{r:\mathsf{Nat}}$ and we are trying to grade at $\mathsf{Ext\,Nat}$ then we get $x : [A]_{r:\mathsf{Ext\,Nat}}$ since $\mathsf{Nat} \subseteq \mathsf{Ext\,Nat}$. The judgement $\Sigma \vdash R \sqcup R' \triangleright S; \iota_1; \iota_2; \theta$ (see Appendix A.2) generates injections into the upper bound type and also serves to unify any type variables in coeffect types and thus produces a substitution which must be combined with the premise's substitution in the output.

*A.3.2 Synthesis.*

*Algorithmic type synthesis* $\boxed{D; \Sigma; \Gamma \vdash t \Rightarrow A; \Delta; \Sigma'; \theta; P}$

$$\frac{}{D; \Sigma; \Gamma \vdash n \Rightarrow \mathsf{Int}; \emptyset; \Sigma; \emptyset; \top} \Rightarrow_n \qquad \frac{\begin{array}{l}(C : (\forall \overrightarrow{\alpha : \kappa} \,.\, A, \theta_\kappa)) \in D \\ \theta, \Sigma', \theta'_\kappa = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}, \theta_\kappa)\end{array}}{D; \Sigma; \Gamma \vdash C \Rightarrow (\theta'_\kappa \uplus \theta)A; \emptyset; \Sigma, \Sigma'; \emptyset; [\![\theta'_\kappa]\!]} \Rightarrow_{\mathrm{C}}$$

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

$$\frac{\begin{array}{l}(x : \forall \overrightarrow{\alpha : \kappa} \,.\, \{A_1, \ldots, A_n\} \Rightarrow B) \in D \\ \theta, \Sigma' = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa})\end{array}}{D; \Sigma; \Gamma \vdash x \Rightarrow \theta B; \emptyset; \Sigma, \Sigma'; \emptyset; [\![(A_1, \ldots, A_n)]\!]} \Rightarrow_{\text{DEF}}$$

$$\frac{\begin{array}{l}D; \Sigma_1; \Gamma \vdash t_1 \Rightarrow \Diamond_\varepsilon A; \Delta_1; \Sigma_2; \theta_1; P_1 \\ D; \Sigma_2; - \vdash p : A \rhd \Delta; P; \theta; \Sigma_3 \\ \mathsf{irrefutable}\ p \\ D; \Sigma_3; \Gamma, \Delta \vdash t_2 \Rightarrow \Diamond_{\varepsilon'} B; \Delta_2; \Sigma_4; \theta_2; P_2 \\ \Gamma' \equiv (\Delta_2 \cap \Delta)\end{array}}{D; \Sigma_1; \Gamma \vdash \mathbf{let}\,\langle p \rangle \leftarrow t_1\ \mathbf{in}\ t_2 \Rightarrow (\Diamond_{(\varepsilon \star \varepsilon')} B); \Delta_1 + (\Delta_2 \backslash \Delta); \Sigma_4; \theta \uplus \theta_1 \uplus \theta_2; P \wedge P_1 \wedge P_2} \Rightarrow_{\text{LET}}$$

$$\frac{\begin{array}{l}D; \Sigma_1; \Gamma \vdash t_1 \Rightarrow A \rightarrow B; \Delta_1; \Sigma_2; \theta_1; P \\ D; \Sigma_2; \Gamma \vdash t_2 \Leftarrow A; \Delta_2; \Sigma_3; \theta_2; P'\end{array}}{D; \Sigma_1; \Gamma \vdash t_1\ t_2 \Rightarrow \theta_2 B; \Delta_1 + \Delta_2; \Sigma_3; \theta_1 \uplus \theta_2; P \wedge P'} \Rightarrow_{\text{APP}}$$

$$\frac{\begin{array}{l}\Sigma_1 \vdash [\Gamma \cap \mathsf{FV}(t)]_\beta \rhd \Gamma'; \theta' \\ D; \Sigma_1; \Gamma' \vdash t \Rightarrow A; \Delta; \Sigma_2; \theta; P\end{array}}{D; \Sigma_1; \Gamma \vdash [t] \Rightarrow \Box_\alpha A; \alpha \cdot \Delta; \Sigma_2, \beta :_\exists \mathsf{Coeffect}, \alpha :_\exists \uparrow \beta; \theta \uplus \theta'; P} \Rightarrow_{\text{PR}}$$

$$\frac{\begin{array}{l}D; \Sigma_1; - \vdash p : \alpha \rhd \Gamma'; P; \theta; \Sigma_2 \\ D; \Sigma_2; \Gamma, \Gamma' \vdash t \Rightarrow B; \Delta; \Sigma_3; \theta'; P' \\ \mathsf{irrefutable}\ p \\ \Gamma' \equiv \Delta \cap \Gamma'\end{array}}{D; \Sigma_1; \Gamma \vdash \lambda p.t \Rightarrow \theta(\alpha \rightarrow B); \Delta \backslash \Gamma'; \Sigma_3, \alpha :_\exists \mathsf{Type}; \theta \uplus \theta'; P \wedge P'} \Rightarrow_{\text{ABS}}$$

## B PROOFS

### B.1 Term substitution and type preservation

**Lemma B.1.** [Restriction Collapse] For contexts $\Gamma_1$ and $\Gamma_2$:

$$\Gamma_{1\,|\Gamma_2} + \Gamma_{1\,|\overline{\Gamma_2}} = \Gamma_1$$

**Lemma B.2.** [Context Shuffle 1] For typing contexts $\Gamma_1$, $\Gamma_1'$, and $\Gamma_2$, variable $x$ and type $A$:

$$(\Gamma_1, x : A, \Gamma_1') + \Gamma_2 = (\Gamma_1 + \Gamma_{2|\Gamma_1}), x : A, (\Gamma_1' + \Gamma_{2|\overline{\Gamma_1}})$$

**Lemma B.3.** [Context Shuffle 2] For typing contexts $\Gamma_1$, $\Gamma_2$, and $\Gamma_2'$, variable $x$ and type $A$:

$$\Gamma_1 + (\Gamma_2, x : A, \Gamma_2') = (\Gamma_{1|\overline{\Gamma_2'}} + \Gamma_2), x : A, (\Gamma_{1|\Gamma_2'} + \Gamma_2')$$

**Lemma B.4.** [Context Shuffle 3] For typing contexts $\Gamma_1$, $\Gamma_2$, $\Gamma_3$, and $\Gamma_4$:

$$((\Gamma_1, \Gamma_2) + (\Gamma_3, \Gamma_4)) = ((\Gamma_1 + \Gamma_{3|\Gamma_1} + \Gamma_{4|\Gamma_1}), (\Gamma_2 + \Gamma_{3|\overline{\Gamma_1}} + \Gamma_{4|\overline{\Gamma_1}}))$$

**Lemma B.5.** [Distribution of Scalar Multiplication over Context Addition] For contexts $\Gamma$, and $r_1, r_2 \in R$:

$$(r_1 \cdot \Gamma) + (r_2 \cdot \Gamma) = (r_1 + r_2) \cdot \Gamma$$

**Lemma B.6.** [Declarative Discharge Inversion] Suppose $\Sigma \vdash [\Gamma_1]_R \triangleright (\Gamma_2', x : [A]_{r''}, \Gamma_2'');\ \theta$. Then one of the following is true:

- $r'' = 1$, $\Gamma_1 = (\Gamma_1', x : A, \Gamma_1'')$, and $\Sigma \vdash [(\Gamma_1', \Gamma_1'')]_R \triangleright (\Gamma_2', \Gamma_2'');\ \theta$
- $\Gamma_1 = (\Gamma_1', x : [A]_{r''}, \Gamma_1'')$, and $\Sigma \vdash [(\Gamma_1', \Gamma_1'')]_R \triangleright (\Gamma_2', \Gamma_2'');\ \theta$

**Lemma B.7.** [Discharger Extension] If $\Sigma \vdash [(\Gamma_1', \Gamma_1'')]_R \triangleright (\Gamma_2', \Gamma_2'');\ \theta$ and $[\Delta]_R$, then $\Sigma \vdash [(\Gamma_1', \Delta, \Gamma_1'')]_R \triangleright (\Gamma_2', \Delta, \Gamma_2'');\ \theta$.

Proof. Since $\Delta$ is fully discharged in $R$, then the only rule that will apply when the derivation gets to $\Delta$ is [gr], and thus, preserve $\Delta$. □

**Lemma B.8.** [Disjoint Collapse] For contexts $\Gamma_1$, $\Delta$ and $\Gamma_2$:

$$(\Gamma_1 + \Delta + \Gamma_2) = (\Gamma_1 + \Delta_{|\Gamma_1}), \Delta_{|\overline{(\Gamma_1, \Gamma_2)}}, (\Gamma_2 + \Delta_{|\Gamma_2})$$

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

Proof. This proof is by induction on the structure of $D; \Sigma; \Gamma, x : A, \Gamma' \vdash t : B$.

$$\text{Case.}\quad \frac{}{D; \Sigma; \emptyset \vdash n : \mathsf{Int}}\ \textsc{int}$$

This case holds trivially, because the typing context is empty.

$$\text{Case.}\quad \frac{\Sigma \vdash B : \mathsf{Type}}{D; \Sigma; y : B \vdash y : B}\ \textsc{var}$$

In this case we know that:
- $\Gamma = \Gamma' = \emptyset$,
- $x = t = y$, and
- $B = A$.

Thus, it suffices to show that the following holds:

$$D; \Sigma; \Delta \vdash [t'/y]y : B$$

However, $[t'/y]y = t'$, and the previous judgment holds by assumption.

$$\text{Case.}\quad \dfrac{\begin{array}{l}(y : \forall \overrightarrow{\alpha : \kappa}\ .\ B) \in D \\ \theta, \Sigma' = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa})\end{array}}{D; \Sigma, \Sigma'; \emptyset \vdash y : \theta B}\ \textsc{def}$$

This case holds trivially, because the typing context is empty.

$$\text{Case.}\quad \dfrac{\begin{array}{l}(C : (\forall \overrightarrow{\alpha : \kappa}\ .\ B, \theta_\kappa)) \in D \\ \theta, \Sigma', \theta'_\kappa = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}, \theta_\kappa)\end{array}}{D; \Sigma, \Sigma'; \emptyset \vdash C : (\theta'_\kappa \uplus \theta)B}\ C$$

This case holds trivially because the typing context is empty.

$$\text{Case.}\quad \dfrac{\begin{array}{l}\Sigma \vdash B_1 : \mathsf{Predicate} \\ (\theta B_1) \\ \theta, \Sigma' = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}) \\ (y : \forall \overrightarrow{\alpha : \kappa}\ .\ B_1 \Rightarrow B_2) \in D\end{array}}{D; \Sigma, \Sigma'; \emptyset \vdash y : \theta B_2}\ \textsc{def}$$

This case holds trivially because the typing context is empty.

$$\text{Case.}\quad \dfrac{D; \Sigma; \Gamma, x : A, \Gamma', y : B_1 \vdash t : B_2}{D; \Sigma; \Gamma, x : A, \Gamma' \vdash \lambda y.t : B_1 \to B_2}\ \textsc{Ty\_abs}$$

By the induction hypothesis we know the following:

$$D; \Sigma; \Gamma + \Delta + (\Gamma', y : B_1) \vdash [t'/x]t : B_2$$

However, we know by the definition of context addition that $(\Gamma + \Delta + (\Gamma', y : B_1)) = ((\Gamma + \Delta + \Gamma'), y : B_1)$, because $y$ is linear. Thus, the previous judgment is equivalent to the following:

$$D; \Sigma; \Gamma + \Delta + \Gamma', y : B_1 \vdash [t'/x]t : B_2$$

Therefore, by reapplying the rule we know the following:

$$D; \Sigma; \Gamma + \Delta + \Gamma' \vdash \lambda y.[t'/x]t : B_2$$

By the definition of substitution $\lambda y.[t'/x]t = [t'/x](\lambda y.t)$, and we obtain our result.

$$\text{Case.}\quad \dfrac{\begin{array}{l}D; \Sigma; - \vdash\ p : B_1 \rhd \Delta'; \theta \\ D; \Sigma; \Gamma, x : A, \Gamma', \Delta' \vdash t : \theta B_2 \\ \mathsf{irrefutable}\ p\end{array}}{D; \Sigma; \Gamma, x : A, \Gamma' \vdash \lambda p.t : B_1 \to B_2}\ \textsc{abs}$$

By the induction hypothesis we know the following:

$$D; \Sigma; \Gamma + \Delta + (\Gamma', \Delta') \vdash [t'/x]t : \theta B_2$$

In this case, $\Delta'$ consists of bound variables in $p$, and thus, must be disjoint from $\Gamma$, $\Delta$, and $\Gamma'$. Thus, $(\Gamma + \Delta + (\Gamma', \Delta')) = ((\Gamma + \Delta + \Gamma'), \Delta')$, and we know the following:

$$D; \Sigma; (\Gamma + \Delta + \Gamma'), \Delta' \vdash [t'/x]t : \theta B_2$$

Therefore, by reapplying the rule we know the following:

$$D; \Sigma; \Gamma + \Delta + \Gamma' \vdash \lambda p.[t'/x]t : B_1 \to B_2$$

By the definition of substitution $\lambda p.[t'/x]t = [t'/x](\lambda p.t)$, and we obtain our result.

$$\text{Case.}\quad \dfrac{\begin{array}{l} D;\Sigma;\Gamma_1 \vdash t_1 : B_1 \longrightarrow B_2 \\ D;\Sigma;\Gamma_2 \vdash t_2 : B_1 \end{array}}{D;\Sigma;\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B_2}\ \textsc{app}$$

In this case we know the following:
- $t = t_1\ t_2$,
- $B = B_2$,
- $(\Gamma, x : A, \Gamma') = (\Gamma_1 + \Gamma_2)$.

Now by the definition of context addition and the fact that $x$ is linear means that $(x : A) \in \Gamma_1$ or $(x : A) \in \Gamma_2$, but not both. Thus, we have the following two cases to consider:
- Suppose $(x : A) \in \Gamma_1$. Then we are in the following situation:
$$\dfrac{\begin{array}{l} D;\Sigma;\Gamma_1', x : A, \Gamma_1'' \vdash t_1 : B_1 \longrightarrow B_2 \\ D;\Sigma;\Gamma_2 \vdash t_2 : B_1 \end{array}}{D;\Sigma;(\Gamma_1', x : A, \Gamma_1'') + \Gamma_2 \vdash t_1\ t_2 : B_2}\ \textsc{app}$$
In order to ensure we obtain our expected result we must determine what $\Gamma$ and $\Gamma'$ are in this case. We know that:
$$\begin{array}{lll} (\Gamma, x : A, \Gamma') & = & (\Gamma_1 + \Gamma_2) \\ & = & (\Gamma_1', x : A, \Gamma_1'') + \Gamma_2 \\ & = & (\Gamma_1' + \Gamma_{2|\Gamma_1}), x : A, (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1}}) \end{array}$$
The last equation holds by Lemma B.2. The previous reasoning implies that $\Gamma = (\Gamma_1' + \Gamma_{2|\Gamma_1})$ and $\Gamma' = (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1}})$.
By the induction hypothesis we know the following:
$$D;\Sigma;\Gamma_1' + \Delta + \Gamma_1'' \vdash [t/x]t_1 : B_1 \longrightarrow B_2$$
Hence, pair this with the second premise above and we can reapply the rule to obtain:
$$D;\Sigma;(\Gamma_1' + \Delta + \Gamma_1'') + \Gamma_2 \vdash t_1\ t_2 : B_2$$
Finally, we must show that $((\Gamma_1' + \Delta + \Gamma_1'') + \Gamma_2) = (\Gamma + \Delta + \Gamma')$, but this follows from the following reasoning:
$$\begin{array}{lll} (\Gamma + \Delta + \Gamma') & = & ((\Gamma_1' + \Gamma_{2|\Gamma_1}) + \Delta + (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1}})) \\ & = & (\Gamma_1' + \Delta + \Gamma_1'' + \Gamma_{2|\Gamma_1} + \Gamma_{2|\overline{\Gamma_1}}) \\ & = & (\Gamma_1' + \Delta + \Gamma_1'' + \Gamma_2) \\ & = & ((\Gamma_1' + \Delta + \Gamma_1'') + \Gamma_2) \end{array}$$
The only non-trivial equation used in the previous reasoning is the third equation, but that holds by Lemma B.1, all others hold by commutativity and associativity of context addition. Thus, we obtain our result.
- Suppose $(x : A) \in \Gamma_2$. Then we are in the following situation:
$$\dfrac{\begin{array}{l} D;\Sigma;\Gamma_1 \vdash t_1 : B_1 \longrightarrow B_2 \\ D;\Sigma;\Gamma_2', x : A, \Gamma_2'' \vdash t_2 : B_1 \end{array}}{D;\Sigma;\Gamma_1 + (\Gamma_2', x : A, \Gamma_2'') \vdash t_1\ t_2 : B_2}\ \textsc{app}$$

This case is similar to the previous case, but using Lemma B.3.

$$\text{Case.}\quad \dfrac{\begin{array}{l} D;\Sigma;\Gamma, x : A, \Gamma', y : B_1 \vdash t : B_2 \\ \Sigma \vdash R : \mathsf{Coeffect} \end{array}}{D;\Sigma;\Gamma, x : A, \Gamma', y : [B_1]_{1:R} \vdash t : B_2}\ \textsc{der}$$

In this case $x : A$ cannot be $y : [B_1]_{1:R}$, because $x$ is linear. Furthermore, we know that $\Gamma' = (\Gamma'', y : [B_1]_{1:R})$. This case then follows from first applying the induction hypothesis to obtain:

$$D;\Sigma;\Gamma + \Delta + (\Gamma'', y : B_1) \vdash t : B_2$$

Then noting that $(\Gamma + \Delta + (\Gamma'', y : B_1)) = (\Gamma + \Delta + \Gamma', y : B_1)$, because $y$ is linear, and thus, the previous judgment is equivalent to the following:

$$D;\Sigma;\Gamma + \Delta + \Gamma'', y : B_1 \vdash t : B_2$$

Thus, we obtain our result by applying the rule to obtain:

$$D;\Sigma;\Gamma + \Delta + \Gamma'', y : [B_1]_{1:R} \vdash t : B_2$$

Finally, because we know that $y$ is linear we know that this is equivalent to the following

$$D;\Sigma;\Gamma + \Delta + (\Gamma'', y : [B_1]_{1:R}) \vdash t : B_2$$

which is our required result.

$$\text{Case.}\quad \frac{\Sigma \vdash r : \uparrow R \qquad \Sigma \vdash R : \mathsf{Coeff} \qquad D;\Sigma;[\Gamma] \vdash t : B}{D;\Sigma; r \cdot \Gamma \vdash [t] : \Box_r B}\;\textsc{pr}$$

This case holds trivially, because $\Gamma$ contains only graded assumptions (by the premise), and thus does not contain any linear variables.

$$\text{Case.}\quad \frac{D;\Sigma;\Gamma_1, x : A, \Gamma_2 \vdash t : B \qquad \Sigma \vdash R : \mathsf{Coeffect}}{D;\Sigma;(\Gamma_1, x : A, \Gamma_2) + [\Delta']_{0:R} \vdash t : B}\;\textsc{weak}$$

In this case $x : A$ cannot be in $\Delta'$, because it is fully discharged. By Lemma B.2 we know the following:

$$((\Gamma_1, x : A, \Gamma_2) + [\Delta']_{0:R}) \quad = \quad ((\Gamma_1 + ([\Delta']_{0:R})_{|\Gamma_1}), x : A, (\Gamma_2 + ([\Delta']_{0:R})_{|\overline{\Gamma_1}}))$$

Thus, it must be the case that $\Gamma = (\Gamma_1 + ([\Delta']_{0:R})_{|\Gamma_1})$ and $\Gamma' = (\Gamma_2 + ([\Delta']_{0:R})_{|\overline{\Gamma_1}})$. By the induction hypothesis we know the following:

$$D;\Sigma;\Gamma_1 + \Delta + \Gamma_2 \vdash [t'/x]t : B$$

and after reapplying the rule we know the following:

$$D;\Sigma;(\Gamma_1 + \Delta + \Gamma_2) + [\Delta']_{0:R} \vdash [t'/x]t : B$$

Finally, we must show that $((\Gamma_1 + \Delta + \Gamma_2) + [\Delta']_{0:R}) = (\Gamma + \Delta + \Gamma')$, but this follows from the following reasoning:

$$\begin{array}{lll}
(\Gamma + \Delta + \Gamma') & = & ((\Gamma_1 + [\Delta']_{0:R|\Gamma_1}) + \Delta + (\Gamma_2 + [\Delta']_{0:R|\overline{\Gamma_1}})) \\
 & = & (\Gamma_1 + [\Delta']_{0:R|\Gamma_1} + \Delta + \Gamma_2 + [\Delta']_{0:R|\overline{\Gamma_1}}) \\
 & = & (\Gamma_1 + \Delta + \Gamma_2 + ([\Delta']_{0:R|\Gamma_1} + [\Delta']_{0:R|\overline{\Gamma_1}})) \\
 & = & (\Gamma_1 + \Delta + \Gamma_2 + [\Delta']_{0:R}) \\
 & = & ((\Gamma_1 + \Delta + \Gamma_2) + [\Delta']_{0:R})
\end{array}$$

The previous reasoning holds by associativity and commutativity of context addition, and Lemma B.1. Thus, we obtain our result.

$$\text{Case.} \quad \frac{\begin{array}{l} D; \Sigma; \Gamma_1 \vdash t_1 : \Diamond_{\varepsilon_1} B_1 \\ D; \Sigma; - \vdash p : B_1 \rhd \Delta; \theta \\ D; \Sigma; \Gamma_2, \Delta \vdash t_2 : \Diamond_{\varepsilon_2} \theta B_2 \\ \text{irrefutable } p \end{array}}{D; \Sigma; \Gamma_1 + \Gamma_2 \vdash \textbf{let} \langle p \rangle \leftarrow t_1 \textbf{ in } t_2 : \Diamond_{(\varepsilon_1 \star \varepsilon_2)} B_2} \text{ LET}\Diamond$$

This case is similar to the case for APP above.

$$\text{Case.} \quad \frac{D; \Sigma; \Gamma, x : A, \Gamma' \vdash t : B'}{D; \Sigma; \Gamma, x : A, \Gamma' \vdash t : \Diamond_1 B'} \text{ PURE}$$

This case follows straightforwardly from the induction hypothesis and reapplying the rule.

$$\text{Case.} \quad \frac{\begin{array}{l} D; \Sigma; \Gamma_1, y : [B']_r, \Gamma_2 \vdash t : B \\ r \sqsubseteq s \end{array}}{D; \Sigma; \Gamma_1, y : [B']_s, \Gamma_2 \vdash t : B}$$

Either $(x : A) \in \Gamma_1$ or $(x : A) \in \Gamma_2$, but not both. In either case, we can apply the induction hypothesis, straightforwardly reorganise the context, and then reapply the rule.

□

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

Proof. This is a proof by induction on the form of $D; \Sigma; \Gamma, x : [A]_r, \Gamma' \vdash t : B$.

$$\text{Case.} \quad \frac{}{D; \Sigma; \emptyset \vdash n : \mathsf{Int}} \text{ INT}$$

This case holds trivially, because the typing context is empty.

$$\text{Case.} \quad \frac{\Sigma \vdash B : \mathsf{Type}}{D; \Sigma; y : B \vdash y : B} \text{ VAR}$$

It cannot be the case that $x = y$ since the premise for substitution states that the variable $x$ is marked as a graded assumption. Subsequently, substitution cannot be applied here since the rest of the context is empty and so there cannot be any graded variable to substitute through.

$$\text{Case.} \quad \frac{\begin{array}{l} (y : \forall \overrightarrow{\alpha : \kappa} \, . \, B) \in D \\ \theta, \Sigma' = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}) \end{array}}{D; \Sigma, \Sigma'; \emptyset \vdash y : \theta B} \text{ DEF}$$

This case holds trivially, because the typing context is empty.

$$\text{Case.} \quad \frac{\begin{array}{l} (C : (\forall \overrightarrow{\alpha : \kappa} \, . \, B, \theta_\kappa)) \in D \\ \theta, \Sigma', \theta'_\kappa = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}, \theta_\kappa) \end{array}}{D; \Sigma, \Sigma'; \emptyset \vdash C : (\theta'_\kappa \uplus \theta) B} \; C$$

This case holds trivially, because the typing context is empty.

$$\text{Case.}\quad \frac{\begin{array}{l}\Sigma \vdash B_1 : \mathsf{Predicate} \\ (\theta B_1) \\ \theta, \Sigma' = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}) \\ (y : \forall \overrightarrow{\alpha : \kappa} \,.\, B_1 \Rightarrow B_2) \in D\end{array}}{D; \Sigma, \Sigma'; \emptyset \vdash y : \theta B_2}\ \textsc{def}$$

This case holds trivially, because the typing context is empty.

$$\text{Case.}\quad \frac{D; \Sigma; \Gamma, x : [A]_r, \Gamma', y : B_1 \vdash t : B_2}{D; \Sigma; \Gamma, x : [A]_r, \Gamma' \vdash \lambda y.t : B_1 \to B_2}\ \textsc{absVar}$$

By the induction hypothesis we know the following:

$$D; \Sigma; \Gamma + r \cdot \Delta + (\Gamma', y : B_1) \vdash [t/x]t : B_2$$

However, we know that $y$ is linear, and thus, by the definition of context addition $\Gamma + r \cdot \Delta + (\Gamma', y : B_1) = (\Gamma + r \cdot \Delta + \Gamma'), y : B_1$. Thus, the previous judgment is equivalent to the following:

$$D; \Sigma; (\Gamma + r \cdot \Delta + \Gamma'), y : B_1 \vdash [t'/x]t : B_2$$

Thus, we obtain our result by reapplying the rule to obtain:

$$D; \Sigma; \Gamma + r \cdot \Delta + \Gamma' \vdash \lambda y.[t'/x]t : B_1 \to B_2$$

By the definition of substitution we know that $\lambda y.[t'/x]t = [t'/x](\lambda y.t)$. Note, that the next case subsumes this one, but we leave this case for illustrative purposes.

$$\text{Case.}\quad \frac{\begin{array}{l}D; \Sigma; - \vdash p : B_1 \rhd \Delta; \theta \\ D; \Sigma; \Gamma, x : [A]_r, \Gamma', \Delta' \vdash t : \theta B_2 \\ \mathsf{irrefutable}\ p\end{array}}{D; \Sigma; \Gamma, x : [A]_r, \Gamma' \vdash \lambda p.t : B_1 \to B_2}\ \textsc{abs}$$

By the induction hypothesis we know the following:

$$D; \Sigma; \Gamma + r \cdot \Delta + (\Gamma', \Delta') \vdash [t'/x]t : \theta B_2$$

In this case, $\Delta'$ consists of bound variables in $p$, and thus, must be disjoint from $\Gamma$, $\Delta$, and $\Gamma'$. Hence, $\Gamma + r \cdot \Delta + (\Gamma', \Delta') = (\Gamma + r \cdot \Delta + \Gamma'), \Delta'$, and thus, we know the following is equivalent to the previous judgment:

$$D; \Sigma; (\Gamma + r \cdot \Delta + \Gamma'), \Delta' \vdash [t'/x]t : \theta B_2$$

Thus, we obtain our result by reapplying the rule:

$$D; \Sigma; (\Gamma + r \cdot \Delta + \Gamma') \vdash \lambda p.[t'/x]t : B_1 \to B_2$$

By the definition of substitution we know that $\lambda p.[t'/x]t = [t'/x](\lambda p.t)$.

$$\text{Case.}\quad \frac{\begin{array}{l}D; \Sigma; \Gamma_1 \vdash t_1 : B_1 \to B_2 \\ D; \Sigma; \Gamma_2 \vdash t_2 : B_1\end{array}}{D; \Sigma; \Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B_2}\ \textsc{app}$$

In this case we know the following:
- $t = t_1\ t_2$,
- $B = B_2$, and
- $(\Gamma, x : [A]_r, \Gamma') = (\Gamma_1 + \Gamma_2)$.

Now it could be the case that $x$ is in $|\Gamma_1|$, or $|\Gamma_2|$, or both. Thus, we have the following cases to consider:

– Suppose $(x : [A]_r) \in \Gamma_1$, but $x$ is not in $|\Gamma_2|$. Then we are in the following situation:

$$\frac{D; \Sigma; \Gamma_1', x : [A]_r, \Gamma_1'' \vdash t_1 : B_1 \to B_2 \qquad D; \Sigma; \Gamma_2 \vdash t_2 : B_1}{D; \Sigma; (\Gamma_1', x : [A]_r, \Gamma_1'') + \Gamma_2 \vdash t_1\ t_2 : B_2}\ \textsc{app}$$

where $\Gamma_1 = (\Gamma_1', x : [A]_r, \Gamma_1'')$. First, in order to be sure we arrive at the required result we must know what $\Gamma$ and $\Gamma'$ are, but we know the following:

$$\begin{array}{lll} (\Gamma_1 + \Gamma_2) & = & (\Gamma_1', x : [A]_r, \Gamma_1'') + \Gamma_2 \\ & = & ((\Gamma_1' + \Gamma_{2|\Gamma_1}), x : [A]_r, (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1}})) \end{array}$$

Hence, given that we know $(\Gamma, x : [A]_r, \Gamma') = \Gamma_1 + \Gamma_2$, then the above implies that $\Gamma = (\Gamma_1' + \Gamma_{2|\Gamma_1})$ and $\Gamma' = (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1}})$.

By the induction hypothesis we know the following:

$$D; \Sigma; \Gamma_1' + r \cdot \Delta + \Gamma_1'' \vdash [t'/x]t_1 : B_1 \to B_2$$

Then we can reapply the rule to obtain:

$$D; \Sigma; (\Gamma_1' + r \cdot \Delta + \Gamma_1'') + \Gamma_2 \vdash ([t'/x]t_1)\ t_2 : B_2$$

By the definition of substitution we know that $([t'/x]t_1)\ t_2 = [t'/x](t_1\ t_2)$. Now all we need to show is that $((\Gamma_1' + r \cdot \Delta + \Gamma_1'') + \Gamma_2) = (\Gamma + r \cdot \Delta + \Gamma')$, but this follows from the following reasoning:

$$\begin{array}{lll} (\Gamma + r \cdot \Delta + \Gamma') & = & ((\Gamma_1' + \Gamma_{2|\Gamma_1}) + r \cdot \Delta + (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1}})) \\ & = & (\Gamma_1' + r \cdot \Delta + \Gamma_1'' + (\Gamma_{2|\Gamma_1} + \Gamma_{2|\overline{\Gamma_1}})) \\ & = & (\Gamma_1' + r \cdot \Delta + \Gamma_1'' + \Gamma_2) \\ & = & ((\Gamma_1' + r \cdot \Delta + \Gamma_1'') + \Gamma_2) \end{array}$$

The above reasoning holds by associativity, commutativity, and Lemma B.1.

– Suppose $(x : [A]_r) \in \Gamma_2$, but $x$ is not in $|\Gamma_1|$. Then we are in the following situation:

$$\frac{D; \Sigma; \Gamma_1 \vdash t_1 : B_1 \to B_2 \qquad D; \Sigma; \Gamma_2', x : [A]_r, \Gamma_2'' \vdash t_2 : B_1}{D; \Sigma; \Gamma_1 + (\Gamma_2', x : [A]_r, \Gamma_2'') \vdash t_1\ t_2 : B_2}\ \textsc{app}$$

where $\Gamma_2 = (\Gamma_2', x : [A]_r, \Gamma_2'')$. This case is similar to the previous case.

– Suppose $(x : [A]_{r_1}) \in \Gamma_1$ and $(x : [A]_{r_2}) \in \Gamma_2$ where $r = r_1 + r_2$. Then we are in the following situation:

$$\frac{D; \Sigma; \Gamma_1', x : [A]_{r_1}, \Gamma_1'' \vdash t_1 : B_1 \to B_2 \qquad D; \Sigma; \Gamma_2', x : [A]_{r_2}, \Gamma_2'' \vdash t_2 : B_1}{D; \Sigma; (\Gamma_1', x : [A]_{r_1}, \Gamma_1'') + (\Gamma_2', x : [A]_{r_2}, \Gamma_2'') \vdash t_1\ t_2 : B_2}\ \textsc{app}$$

where $\Gamma_1 = (\Gamma_1', x : [A]_{r_1}, \Gamma_1'')$ and $\Gamma_2 = (\Gamma_2', x : [A]_{r_2}, \Gamma_2'')$. Just as we did above we first need to know what $\Gamma$ and $\Gamma'$ are. We have the following reasoning:

$$\begin{array}{l} (\Gamma_1', x : [A]_{r_1}, \Gamma_1'') + (\Gamma_2', x : [A]_{r_2}, \Gamma_2'') \\ \quad = (\Gamma_1', \Gamma_1'', x : [A]_{r_1}) + (\Gamma_2', \Gamma_2'', x : [A]_{r_2}) \\ \quad = (((\Gamma_1', \Gamma_1'') + (\Gamma_2', \Gamma_2'')), x : [A]_{(r_1+r_2)}) \\ \quad = (((\Gamma_1' + \Gamma'_{2|\Gamma_1'} + \Gamma''_{2|\Gamma_1'}), (\Gamma_1'' + \Gamma'_{2|\overline{\Gamma_1'}} + \Gamma''_{2|\overline{\Gamma_1'}})), x : [A]_{(r_1+r_2)}) \\ \quad = ((\Gamma_1' + \Gamma'_{2|\Gamma_1'} + \Gamma''_{2|\Gamma_1'}), x : [A]_{(r_1+r_2)}, (\Gamma_1'' + \Gamma'_{2|\overline{\Gamma_1'}} + \Gamma''_{2|\overline{\Gamma_1'}})) \end{array}$$

All of the reasoning in the above holds by associativity and commutativity of context addition, the definition of context addition, and Lemma B.4. This implies that it must be the case that up to commutativity $\Gamma = (\Gamma'_1 + \Gamma'_{2\,|\Gamma'_1} + \Gamma''_{2\,|\Gamma'_1})$ and $\Gamma' = (\Gamma''_1 + \Gamma'_{2\,|\overline{\Gamma'_1}} + \Gamma''_{2\,|\overline{\Gamma'_1}})$. By the induction hypothesis we know the following:

$$\begin{array}{l} D;\Sigma;\Gamma'_1 + r_1\cdot\Delta + \Gamma''_1 \vdash [t'/x]t_1 : B_1 \to B_2 \\ D;\Sigma;\Gamma'_2 + r_2\cdot\Delta + \Gamma''_2 \vdash [t'/x]t_2 : B_1 \end{array}$$

Now we reapply the rule to obtain:

$$D;\Sigma;(\Gamma'_1 + r_1\cdot\Delta + \Gamma''_1) + (\Gamma'_2 + r_2\cdot\Delta + \Gamma''_2) \vdash ([t'/x]t_1)\,([t'/x]t_2) : B_2$$

By the definition of substitution we know that $([t'/x]t_1)\,([t'/x]t_2) = [t'/x](t_1\ t_2)$. All we have left to do is show that $(\Gamma'_1 + r_1\cdot\Delta + \Gamma''_1) + (\Gamma'_2 + r_2\cdot\Delta + \Gamma''_2) = \Gamma + r\cdot\Delta + \Gamma'$, but this follows from the following reasoning:

$$\begin{array}{l} \Gamma + r\cdot\Delta + \Gamma' \\ \quad = (\Gamma'_1 + \Gamma'_{2\,|\Gamma'_1} + \Gamma''_{2\,|\Gamma'_1}) + (r_1 + r_2)\cdot\Delta + (\Gamma''_1 + \Gamma'_{2\,|\overline{\Gamma'_1}} + \Gamma''_{2\,|\overline{\Gamma'_1}}) \\ \quad = ((\Gamma'_1 + (r_1 + r_2)\cdot\Delta + \Gamma''_1) + \Gamma'_{2\,|\Gamma'_1} + \Gamma'_{2\,|\overline{\Gamma'_1}} + \Gamma''_{2\,|\Gamma'_1} + \Gamma''_{2\,|\overline{\Gamma'_1}}) \\ \quad = (\Gamma'_1 + (r_1 + r_2)\cdot\Delta + \Gamma''_1) + (\Gamma'_2 + \Gamma''_2) \\ \quad = (\Gamma'_1 + r_1\cdot\Delta + r_2\cdot\Delta + \Gamma''_1) + (\Gamma'_2 + \Gamma''_2) \\ \quad = ((\Gamma'_1 + r_1\cdot\Delta + \Gamma''_1) + (\Gamma'_2 + r_2\cdot\Delta + \Gamma''_2)) \end{array}$$

The previous reasoning holds by commutativity and associativity of context addition, Lemma B.5, and Lemma B.1.

$$\text{Case.}\quad \frac{\begin{array}{l} D;\Sigma;\Gamma'', y : B' \vdash t : B \\ \Sigma \vdash R : \mathsf{Coeffect} \end{array}}{D;\Sigma;\Gamma'', y : [B']_{1:R} \vdash t : B}\ \textsc{der}$$

In this case $(\Gamma'', y : [B']_{1:R}) = (\Gamma, x : [A]_r, \Gamma')$, and thus, we have the following cases to consider:

– Suppose $\Gamma'' = \Gamma$, $y : [B']_{1:R} = x : [A]_r$, and $\Gamma' = \emptyset$. Then we are in the following situation:

$$\frac{\begin{array}{l} D;\Sigma;\Gamma, x : A \vdash t : B \\ \Sigma \vdash R : \mathsf{Coeffect} \end{array}}{D;\Sigma;\Gamma, x : [A]_{1:R} \vdash t : B}\ \textsc{der}$$

It suffices to show that $D;\Sigma;\Gamma + 1\cdot\Delta \vdash [t'/x]t : B$, but this is equivalent to $D;\Sigma;\Gamma + \Delta \vdash [t'/x]t : B$. This easily follows by applying Lemma 7.1 to the premise $D;\Sigma;\Gamma, x : A \vdash t : B$.

– Suppose $y : [B']_{1:R} \neq x : [A]_r$, $\Gamma'' = (\Gamma_1, x : [A]_r, \Gamma'_1)$. Then we are in the following situation:

$$\frac{\begin{array}{l} D;\Sigma;\Gamma_1, x : [A]_r, \Gamma'_1, y : B' \vdash t : B \\ \Sigma \vdash R : \mathsf{Coeffect} \end{array}}{D;\Sigma;\Gamma_1, x : [A]_r, \Gamma'_1, y : [B']_{1:R} \vdash t : B}\ \textsc{der}$$

This case follows similarly to the case for abstractions: apply the induction hypothesis, reorganise the context using the fact that $y$ is linear, and then reapply the rule.

$$\text{Case.}\quad \frac{\begin{array}{l} \Sigma \vdash r' : \uparrow R \qquad \Sigma \vdash R : \mathsf{Coeff} \\ D;\Sigma;[\Gamma_1] \vdash t : B \end{array}}{D;\Sigma;r'\cdot\Gamma_1 \vdash [t] : \Box_{r'} B}\ \textsc{pr}$$

In this case $(r'\cdot\Gamma_1) = (\Gamma, x : [A]_r, \Gamma')$. Thus, $\Gamma_1 = \Gamma'_1, x : [A]_{r''}, \Gamma''_1$ and $r = r'\cdot r''$ and $\Gamma = r'\cdot\Gamma'_1$ and $\Gamma' = r'\cdot\Gamma''_1$. We are in the following situation:

$$\frac{\Sigma \vdash r' : \uparrow R \qquad \Sigma \vdash R : \mathsf{Coeff} \qquad D; \Sigma; [\Gamma'_1, x : [A]_{r''}, \Gamma''_1] \vdash t : B}{D; \Sigma; r' \cdot (\Gamma'_1, x : [A]_{r''}, \Gamma''_1) \vdash [t] : \Box_{r'} B}\ \textsc{pr}$$

The inductive hypothesis on $t$ gives us that:

$$D; \Sigma; \Gamma'_1 + r'' \cdot \Delta + \Gamma''_1 \vdash [t'/x]t : B$$

Thus by the definition of substitution (that $[t'/x][t] = [[t'/x]t]$), then we can apply promotion again (since the context is still all graded) yielding:

$$\frac{\Sigma \vdash r' : \uparrow R \qquad \Sigma \vdash R : \mathsf{Coeff} \qquad D; \Sigma; \Gamma'_1 + r'' \cdot \Delta + \Gamma''_1 \vdash [t'/x]t : B}{D; \Sigma; r' \cdot \Gamma'_1 + r' \cdot (r'' \cdot \Delta) + r' \cdot \Gamma''_1 \vdash [t'/x][t] : \Box_{r'} B}\ \textsc{pr}$$

Satisfying the lemma statement by associativity of multiplication (such that $r' \cdot (r'' \cdot \Delta) = (r' \cdot r'') \cdot \Delta$, and distributivity of multiplication over addition.

Case. $$\frac{D; \Sigma; \Gamma'' \vdash t : B \qquad \Sigma \vdash R : \mathsf{Coeffect}}{D; \Sigma; \Gamma'' + [\Delta']_{0:R} \vdash t : B}\ \textsc{weak}$$

There are two cases to consider here:

– If $(x : [A]_r) \in [\Delta']_{0:R}$, then $r = 0$ and $x$ is unused by $t$ and thus $[t'/x]t = t$. Thus $\Delta' = \Delta_1, x : [A]_r, \Delta_2$ then can re-apply weakening to the premise here, with the weakening context $\Delta_1 + \Delta + \Delta_2$
– If $x : [A]_r \notin [\Delta']_{0:R}$ then we have the situation:

$$\frac{D; \Sigma; \Gamma_1, x : [A]_r, \Gamma_2 \vdash t : B \qquad \Sigma \vdash R : \mathsf{Coeffect}}{D; \Sigma; (\Gamma_1, x : [A]_r, \Gamma_2) + [\Delta']_{0:R} \vdash t : B}\ \textsc{weak}$$

where $\Gamma'' = (\Gamma_1, x : [A]_r, \Gamma_2)$. Then by the induction hypothesis we know the following:

$$D; \Sigma; \Gamma_1 + r \cdot \Delta + \Gamma_2 \vdash [t'/x]t : B$$

Then by reapplying the rule we obtain:

$$D; \Sigma; (\Gamma_1 + r \cdot \Delta + \Gamma_2) + [\Delta']_{0:R} \vdash [t'/x]t : B$$

which is our required result up to associativity of the context.

Case. $$\frac{D; \Sigma; \Gamma_1 \vdash t_1 : \Diamond_{\varepsilon_1} B' \qquad D; \Sigma; - \vdash p : B' \rhd \Delta'; \theta \qquad D; \Sigma; \Gamma_2, \Delta' \vdash t_2 : \Diamond_{\varepsilon_2} \theta B \qquad \text{irrefutable } p}{D; \Sigma; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ \langle p \rangle \leftarrow t_1\ \mathbf{in}\ t_2 : \Diamond_{(\varepsilon_1 \star \varepsilon_2)} B}\ \textsc{let}\Diamond$$

This case follows similar to the APP case, but when reorganising the context in the application to the induction hypothesis to the third premise above we use the fact that $\Delta'$ is disjoint from all the other contexts just as we did the case of ABS.

Case. $$\frac{D; \Sigma; \Gamma, x : [A]_r, \Gamma' \vdash t : B}{D; \Sigma; \Gamma, x : [A]_r, \Gamma' \vdash t : \Diamond_1 B}\ \textsc{pure}$$

This case follows directly by applying the induction hypothesis, and then reapplying the rule.

Case. $$\dfrac{\begin{array}{l}D;\Sigma;\Gamma_1, y : [B']_{r_1}, \Gamma_2 \vdash t : B\\ r_1 \sqsubseteq r_2\end{array}}{D;\Sigma;\Gamma_1, y : [B']_{r_2}, \Gamma_2 \vdash t : B}\ \sqsubseteq$$

In this case $(\Gamma, x : [A]_r, \Gamma') = (\Gamma_1, y : [B']_{r_2}, \Gamma_2)$. Then we must consider the following cases:

– Suppose $(x : [A]_r) \in \Gamma_1$. In this case we are in the following situation:
$$\dfrac{\begin{array}{l}D;\Sigma;\Gamma_1', x : [A]_r, \Gamma_1'', y : [B']_{r_1}, \Gamma_2 \vdash t : B\\ r_1 \sqsubseteq r_2\end{array}}{D;\Sigma;\Gamma_1', x : [A]_r, \Gamma_1'', y : [B']_{r_2}, \Gamma_2 \vdash t : B}\ \sqsubseteq$$
By the induction hypothesis we know the following:
$$D;\Sigma;\Gamma_1' + r \cdot \Delta + (\Gamma_1'', y : [B']_{r_1}, \Gamma_2) \vdash [t'/x]t : B$$
Using Lemma B.8 we can reorganise the typing context as follows:
$$\begin{array}{l}(\Gamma_1' + r \cdot \Delta + (\Gamma_1'', y : [B']_{r_1}, \Gamma_2))\\ \quad = ((\Gamma_1' + (r \cdot \Delta)_{|\Gamma_1'}), (r \cdot \Delta)_{\overline{|(\Gamma_1', \Gamma_1'', y:[B']_r, \Gamma_2)}}, (\Gamma_1'' + (r \cdot \Delta)_{|\Gamma_1''}), ((y : [B']_{r_1}) + (r \cdot \Delta)_{|(y:[B']_{r_1})}), (\Gamma_2 + (r \cdot \Delta)_{|\Gamma_2}))\\ \quad = ((\Gamma_1' + (r \cdot \Delta)_{|\Gamma_1'}), (r \cdot \Delta)_{\overline{|(\Gamma_1', \Gamma_1'', y:[B']_r, \Gamma_2)}}, (\Gamma_1'' + (r \cdot \Delta)_{|\Gamma_1''}), y : [B']_{(r_1+r_3)}, (\Gamma_2 + (r \cdot \Delta)_{|\Gamma_2}))\\ \quad = (\Gamma_3, y : [B']_{(r_1+r_3)}, \Gamma_3')\end{array}$$
where $(r \cdot \Delta)_{|(y:[B']_{r_1})} = y : [B']_{r_3}$, or $r_3 = 0$ when $(r \cdot \Delta)_{|(y:[B']_{r_1})} = \emptyset$, $\Gamma_3 = (\Gamma_1' + (r \cdot \Delta)_{|\Gamma_1'}), (r \cdot \Delta)_{\overline{|(\Gamma_1', \Gamma_1'', y:[B']_r, \Gamma_2)}}, (\Gamma_1'' + (r \cdot \Delta)_{|\Gamma_1''})$ and $\Gamma_3' = (\Gamma_2 + (r \cdot \Delta)_{|\Gamma_2})$.
Hence, we know the judgment above is equivalent to the following:
$$D;\Sigma;\Gamma_3, y : [B']_{(r_1+r_3)}, \Gamma_3' \vdash [t'/x]t : B$$
Now after reapplying the rule we know:
$$D;\Sigma;\Gamma_3, y : [B']_{(r_2+r_3)}, \Gamma_3' \vdash [t'/x]t : B$$
because we $r_1 \sqsubseteq r_2$ implies that $(r_1 + r_3) \sqsubseteq (r_2 + r_3)$. At this point we just need to show that $(\Gamma_3, y : [B']_{(r_2+r_3)}, \Gamma_3') = (\Gamma_1' + r \cdot \Delta + (\Gamma_1'', y : [B']_{r_2}, \Gamma_2))$, but this follows by first unfolding $\Gamma_3$ and $\Gamma_3'$, and then collapsing $\Delta$ similar to above.

– Suppose $(x : [A]_r) \in \Gamma_2$. This case is similar to the previous case.

– Suppose $x : [A]_r = y : [B']_{r_2}$. We are in the following situation:
$$\dfrac{\begin{array}{l}D;\Sigma;\Gamma, x : [A]_{r'}, \Gamma' \vdash t : B\\ r' \sqsubseteq r\end{array}}{D;\Sigma;\Gamma, x : [A]_r, \Gamma' \vdash t : B}\ \sqsubseteq$$
This case straightforwardly follows by applying the induction hypothesis, and reapplying the rule for each element of $r' \cdot \Delta$, because we know that $r \cdot \Delta$ is bigger.

□

**Lemma B.9.** [Typed value lemma] Given a term $t$ which is a value, i.e., $t = v$ and $D;\Sigma;\Gamma \vdash v : A$ then we can conclude the following depending on $A$:

- $(A = \mathsf{Int}) \implies v = n$ for some integer constant $n$;
- $(A = KA_0 \ldots A_n) \implies v = C\ v_0 \,..\, v_m$ for some values $v_0 \ldots v_m$;
- $(A = \Diamond_\varepsilon B) \implies v = \langle t' \rangle$ for some term $t'$;
- $(A = \Box_c B) \implies v = [v']$ for some value $v'$;
- $(A = B \multimap B') \implies v = \lambda p.t$ for some patterns $p$ and term $t$.
- *otherwise* $v = x$ for some variable in $\Gamma$.

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

Proof. This is a proof by induction on the form of $D;\Sigma;- \vdash p : A \rhd \Gamma;\theta$.

$$\text{Case.}\quad \frac{0 \sqsubseteq r \quad \Sigma \vdash A : \mathsf{Type}}{D;\Sigma;r:R \vdash \_ : A \rhd \emptyset;\emptyset}\;[\textsc{p\_}]$$

This case holds trivially, because $r : R$ is not $-$.

$$\text{Case.}\quad \frac{\Sigma \vdash A : \mathsf{Type}}{D;\Sigma;- \vdash x : A \rhd x : A;\emptyset}\;\textsc{pVar}$$

In this case $p = x$, $\Gamma = x : A$ and $\theta = \emptyset$. At this point we know the following:

$$(v \rhd x)t = [v/x]t$$

Thus, choose $t' = [v/x]t$. Then, since $\Gamma = x : A$, we know the following:

$$D;\Sigma;\Gamma_1, x : A \vdash t : \theta B$$

Thus, by Lemma 7.1 we know that $D;\Sigma;\Gamma_1 + \Gamma_2 \vdash [v/x]t : \theta B$.

$$\text{Case.}\quad \frac{\Sigma \vdash A' : \mathsf{Type}}{D;\Sigma;r:R \vdash x : A' \rhd x : [A']_{r:R};\emptyset}\;[\textsc{pVar}]$$

This case holds trivially, because $r : R$ is not $-$.

$$\text{Case.}\quad \frac{}{D;\Sigma;- \vdash n : \mathsf{Int} \rhd \emptyset;\emptyset}\;\textsc{pInt}$$

This case holds trivially, because irrefutable $n$ is false.

$$\text{Case.}\quad \frac{1 \sqsubseteq r}{D;\Sigma;r:R \vdash n : \mathsf{Int} \rhd \emptyset;\emptyset}\;[\textsc{pInt}]$$

This case holds trivially, because $r : R$ is not $-$.

$$\text{Case.}\quad \frac{D;\Sigma;r:R \vdash p : A' \rhd \Gamma;\theta \quad \Sigma \vdash r : R}{D;\Sigma;- \vdash [p] : \Box_r A' \rhd \Gamma;\theta}\;\textsc{p}\Box$$

We know by assumption that irrefutable $[p]$, but this implies that irrefutable $p$. In addition, we know that $A = \Box_r A'$, and hence, by Lemma B.9 we know that $v = [v']$ and $\Gamma_2 = r \cdot \Gamma_2'$, this then implies that we know $D;\Sigma;r \cdot \Gamma_2' \vdash [v'] : \Box_r A'$. So it must be the case that $D;\Sigma;[\Gamma_2'] \vdash v' : A'$. Furthermore, by assumption we know that $D;\Sigma;- \vdash p : \Box_r A' \rhd \Gamma;\theta$ holds, and hence, it must be the case that $p = [p']$, and by inversion we know that $D;\Sigma;r:R \vdash p' : A' \rhd \Gamma;\theta$.

So far from the above reasoning and the set of assumptions we know the following:

– irrefutable $p$,
– $D;\Sigma;r : R \vdash p' : A' \rhd \Gamma;\theta$,
– $D;\Sigma;[\Gamma_2'] \vdash v' : A'$,
– $D;\Sigma;\Gamma_1,\Gamma \vdash t : \theta B$

Then by Lemma 7.4 we know there exists a term $t''$ such that $(v' \rhd p')t = t''$ and $D;\Sigma;\Gamma_1 + r \cdot \Gamma_2' \vdash t'' : \theta B$. Thus, choose $t' = t''$. Then we know that $([v'] \rhd [p'])t = t''$, and we already know that $D;\Sigma;\Gamma_1 + \Gamma_2 \vdash t'' : \theta B$ because $\Gamma_2 = r \cdot \Gamma_2'$.

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

This case holds trivially, because $r : R$ is not $-$.

Case. $$\frac{\begin{array}{l}(C : (\forall \overrightarrow{\alpha : \kappa} \,.\, B_1 \to \ldots \to B_n \to A', \theta_\kappa)) \in D \\ \theta,\Sigma',\theta_\kappa' = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}, \theta_\kappa) \\ \Sigma,\Sigma' \vdash \theta A' \sim A \rhd \theta' \\ D;\Sigma,\Sigma';- \vdash p_i : (\theta_\kappa' \uplus \theta' \uplus \theta)B_i \rhd \Gamma_i;\theta_i\end{array}}{D;\Sigma,\Sigma';- \vdash C\,p_1 \mathinner{..} p_n : A \rhd \overrightarrow{\Gamma_i};\theta_\kappa' \uplus \theta' \uplus \theta_1 \uplus \ldots \uplus \theta_n}\ \textsc{p}\mathrm{C}$$

From irrefutable $(C_1\,p_0 \mathinner{..} p_n)$ in the premise we conclude that irrefutable $p_i$ for all subpatterns by the definition of irrefutability, and that $C \in K$ and that cardinality $K \equiv 1$ and we get $\Gamma_1 = \overrightarrow{\Gamma_i}$.
By the vaue lemma we then have that $v = C\,v_0 \mathinner{..} v_n$, which must have been typed by (C) with repeated (APP) such that: $D;\Sigma;\Gamma_i' \vdash v_i : B_i$.
Combined with irrefutable $p_i$ we then get the inductive hypotheses for all $i$ where $\Gamma_{2i}$ gives the typing context for each

$$(v_i \rhd p_i)t_i = t_{i+1} \ \wedge\ D;\Sigma;\Gamma_1 + r \cdot \Gamma_i' \vdash t_{i+1} : A$$

Thus by applying $(\rhd_C)$ to each inductive progress result we get progress overall with $(v_0 \rhd C\,p_0 \mathinner{..} p_n)t = t_{i+1}$.
Furthemore, by induction of $i$ we get the final typing that $D;\Sigma;G1 + r * G'0 + \ldots r * G'n| - tn + 1 : A$ since each substitution substituions a $vi$ in for each. $\Gamma_i$ in the context $\Gamma_1$. Thus by distributivity of $\cdot$ over $+$ gives $D;\Sigma;\Gamma_1 + r \cdot \Gamma_2 \vdash t_{n+1} : A$.

Case. $$\frac{\begin{array}{l}(C : (\forall \overrightarrow{\alpha : \kappa} \,.\, B_1 \to \ldots \to B_n \to A', \theta_\kappa)) \in D \\ 1 \sqsubseteq r \\ \theta,\Sigma',\theta_\kappa' = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}, \theta_\kappa) \\ \Sigma,\Sigma' \vdash \theta A' \sim A \rhd \theta' \\ D;\Sigma,\Sigma';r : R \vdash p_i : (\theta_\kappa' \uplus \theta' \uplus \theta)B_i \rhd \Gamma_i;\theta_i\end{array}}{D;\Sigma,\Sigma';r : R \vdash C\,p_1 \mathinner{..} p_n : A \rhd \overrightarrow{\Gamma_i};\theta_\kappa' \uplus \theta' \uplus \theta_1 \uplus \ldots \uplus \theta_n}\ [\textsc{p}\mathrm{C}]$$

This case holds trivially, because $r : R$ is not $-$.

□

**Lemma B.10.** [Monoidal action of flattening with respect to scalar multiplication] For coeffect types $R, R', S$ and grades $r : R$, $r' : R'$ and contexts $\Gamma$, then if $\mathsf{flatten}(r, R, r', R') = (s, S)$, we have:

$$(s : S) \cdot \Gamma \sqsubseteq (r : R) \cdot ((r' : R') \cdot \Gamma)$$

Proof. Without loss of generality we can focus on the $\Gamma = y : [A]_{s':S'}$ such that the proof reduces to proving the following (expanding the definition of scalar multiplication, Def 4.8), assuming that $R \sqcup S' \rhd S''; (\iota_1, \iota_2)$ and $R' \sqcup S'' \rhd S'''; (\iota'_1, \iota'_2)$ and $S \sqcup S' \rhd S'''; (\iota''_1, \iota''_2)$

$$\begin{aligned}
&(r : R) \cdot ((r' : R') \cdot y : [A]_{s':S'}) \\
=\ &(r : R) \cdot (y : [A]_{\iota_1 r' \cdot \iota_2 s'}) \\
=\ &y : [A]_{\iota'_1 r \cdot \iota'_2(\iota_1 r' \cdot \iota_2 s')} \\
\sqsubseteq\ &y : [A]_{\iota''_1 s \cdot \iota''_2 s'} \\
=\ &(s : S) \cdot y : [A]_{s':S'}
\end{aligned}$$

The proof then proceeds by case analysis, case analysis on flatten, expanding the definition of scalar multiplication and $\sqcup$. In most cases, the approximation $\sqsubseteq$ is really just equality.

For example, for $R = R' = \mathsf{Nat}$ then $\mathsf{flatten}(r, \mathsf{Nat}, r', \mathsf{Nat}) = (r \cdot r', \mathsf{Nat})$ Therefore $s = r \cdot r'$ and by associativity of $\cdot$ we get: $(r \cdot r') \cdot \iota''_2 s' = r \cdot (r' \cdot \iota'_2 s')$ and that there is a unique injection from $S' \longrightarrow \mathsf{Nat}$ therefore $\iota''_2 = \iota'_2$.

In cases involving levels, the approximation comes into play. For example, in the case of $R = R' = \mathsf{Level}$. then $\mathsf{flatten}(r, \mathsf{Level}, r', \mathsf{Level}) = (r \sqcap r', \mathsf{Level})$. Since $\cdot$ for Level is meet $\sqcup$ and since $(r \sqcap s) \sqsubseteq (r \sqcup s)$ we get the approximation that $(r \sqcup r') \cdot \Gamma \sqsubseteq (r \cdot r') \cdot \Gamma_2$ since $\cdot = \sqcup$ for levels (again also applying associativity of $\cdot$).

The rest of this proof has been mechanised in Agda (see supplement). This proof utilises the fact that Gr coeffects products are commutative and associative, and it utilises the syntactic restriction which essentially stratifies a coeffect types so that: level 0 comprises Nat and Level; level 1 comprises Ext $R$ for all level-0 coeffect types $R$; level 2 comprises Int $R$ for all level-1 and level-0 coeffect types $R$; level 3 comprises $R \times S$ for all coeffect types $R$ and $S$ and $R \neq S$. Thus products occur only at the top-level. □

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

Proof. This is a proof by induction on the form of $D; \Sigma; r : R \vdash p : A \rhd \Gamma; \theta$.

$$\text{Case.}\quad \frac{0 \sqsubseteq r \qquad \Sigma \vdash A : \mathsf{Type}}{D; \Sigma; r : R \vdash \_ : A \rhd \emptyset; \emptyset}\ [\text{P\_}]$$

In this case, $t' = t$, because $(v \rhd \_)t = t$.
Now we also know by assumption that $D; \Sigma; \Gamma_1 \vdash t : \theta B$, thus we can apply weakening to get $D; \Sigma; \Gamma_1 + [\Gamma_2]_{0:R} \vdash t : \theta B$, and finally, we can apply the approximation rule to get $D; \Sigma; \Gamma_1 + r \cdot \Gamma_2 \vdash t : \theta B$, because we know that $0 \sqsubseteq r$ from the premise of [P_].

$$\text{Case.}\quad \frac{\Sigma \vdash A : \mathsf{Type}}{D; \Sigma; - \vdash x : A \rhd x : A; \emptyset}\ \text{pVar}$$

This case holds trivially, because $-$ is not $r : R$.

$$\text{Case.}\quad \frac{\Sigma \vdash A' : \mathsf{Type}}{D; \Sigma; r : R \vdash\ x : A\ \rhd\ x : [A]_{r:R}; \emptyset}\ [\textsc{pVar}]$$

In this case we know that $p = x$, $\Gamma = x : [A]_{r:R}$, and $\theta = \emptyset$. We know that $(v \rhd p)t = (v \rhd x)t = [v/x]t$. Thus, choose $t' = [v/x]t$. Now we know that $D; \Sigma; \Gamma_1, x : [A]_{r:R} \vdash t : B$ and $D; \Sigma; [\Gamma_2] \vdash v : A$ imply by Lemma 7.2 that $D; \Sigma; \Gamma_1 + r \cdot \Gamma_2 \vdash [v/x]t : B$.

$$\text{Case.}\quad \frac{}{D; \Sigma; - \vdash\ n : \mathsf{Int}\ \rhd\ \emptyset; \emptyset}\ \textsc{pInt}$$

This case holds trivially, because $-$ is not $r : R$.

$$\text{Case.}\quad \frac{1 \sqsubseteq r}{D; \Sigma; r : R \vdash\ n : \mathsf{Int}\ \rhd\ \emptyset; \emptyset}\ [\textsc{pInt}]$$

This case holds trivially, because irrefutable $n$ is false.

$$\text{Case.}\quad \frac{\begin{array}{l} D; \Sigma; r : R \vdash\ p : A'\ \rhd\ \Gamma; \theta \\ \Sigma \vdash r : R \end{array}}{D; \Sigma; - \vdash\ [p] : \Box_r A'\ \rhd\ \Gamma; \theta}\ \textsc{p}\Box$$

This case holds trivially, because $-$ is not $r : R$.

$$\text{Case.}\quad \frac{\begin{array}{l} D; \Sigma; s : S \vdash\ p : A'\ \rhd\ \Delta; \theta \\ \Sigma \vdash r' : \uparrow R' \\ \mathsf{flatten}(r, R, r', R') = (s, S) \end{array}}{D; \Sigma; r : R \vdash\ [p] : \Box_{r'} A'\ \rhd\ \Delta; \theta}\ [\textsc{p}\Box]$$

We know from the assumptions that irrefutable $[p]$, which implies that irrefutable $p$. In addition, we know that $A = \Box_{r'} A'$, and hence, by Lemma B.9 we know that $v = [v']$. There are three possibilities for the typing of $[v']$:

(1) (promotion)

$$\text{Case.}\quad \frac{\begin{array}{c} \Sigma \vdash r' : \uparrow R' \qquad \Sigma \vdash R' : \mathsf{Coeff} \\ D; \Sigma; [\Gamma_2'] \vdash v' : A' \end{array}}{D; \Sigma; r' \cdot \Gamma_2' \vdash [v'] : \Box_{r'} A'}\ \textsc{pr}$$

where $\Gamma_2 = r' \cdot \Gamma_2'$
By induction on $D; \Sigma; s : S \vdash\ p : A' \rhd \Delta; \theta$ with $D; \Sigma; \Gamma_2' \vdash v' : A'$, we then get the following:

$$\begin{array}{l} (v' \rhd p)t = t' \\ \wedge\ D; \Sigma; \Gamma_1 + s \cdot \Gamma_2' \vdash t' : \theta B \end{array}$$

From this we can conclude that we have progress:

$$\frac{(v' \rhd p)t = t'}{([v'] \rhd [p])t = t'}\ \rhd_{\Box}$$

For preservation, our goal is to get $D;\Sigma;\Gamma_1 + r \cdot \Gamma_2 \vdash t' : \theta B$ which is equal to $D;\Sigma;\Gamma_1 + r \cdot (r' \cdot \Gamma_2') \vdash t' : \theta B$.
Given that $\mathsf{flatten}(r, R, r', R') = (s, S)$, we apply Lemma B.10 with the induction hypothesis to get:

$$(s : S) \cdot \Gamma \sqsubseteq (r : R) \cdot ((r' : R') \cdot \Gamma)$$

(2) (weakening) If we conclude with weakening, then there must have still been an application applied previously, i.e.:

$$\dfrac{\Sigma \vdash R : \mathsf{Coeff} \qquad \dfrac{\Sigma \vdash r' : \uparrow R' \qquad \Sigma \vdash R' : \mathsf{Coeff} \qquad D;\Sigma;[\Gamma_2'] \vdash v' : A'}{D;\Sigma;r' \cdot \Gamma_2' \vdash [v'] : \Box_{r'} A'}\ \textsc{pr}}{D;\Sigma;r' \cdot \Gamma_2' + [\Delta]_{0:R} \vdash [v] : \Box_{r'} A}\ \textsc{weak}$$

i.e. $\Gamma_2 = r' \cdot \Gamma_2' + [\Delta]_{0:R}$. We can thus apply our lemma to the promotion rule (see case (1)) above yielding, $([v'] \rhd [p])t = t'$ and then reapply weakening:

$$\dfrac{\Sigma \vdash R : \mathsf{Coeff} \qquad D;\Sigma;\Gamma_1 + r' \cdot \Gamma_2' \vdash t' : \theta B}{D;\Sigma;\Gamma_1 + r' \cdot \Gamma_2' + [\Delta]_{0:R} \vdash t' : \theta B}\ \textsc{weak}$$

which by absorption has the context equal to $\Gamma_1 + r' \cdot (\Gamma_2' + [\Delta]_{0:R})$ reaching our goal.

(3) (approximation) If we conclude with approximation, then the approximation must still be applied to a promotion since syntactically we have $[v']$, thus there must be a derivation such that:

$$\dfrac{\dfrac{\Sigma \vdash r' : \uparrow R' \qquad \Sigma \vdash R' : \mathsf{Coeff} \qquad D;\Sigma;[\Gamma_2'] \vdash v' : A'}{D;\Sigma;r' \cdot \Gamma_2' \vdash [v'] : \Box_{r'} A'}\ \textsc{pr} \qquad r' \sqsubseteq s}{D;\Sigma;s \cdot \Gamma_2' \vdash [v] : \Box_{r'} A'}\ \sqsubseteq$$

The reasoning is then similar to above, applying the lemma to the promotion rule of $[v']$ and then reapply approximation.

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

This case holds trivially, because $-$ is not $r : R$.

Case. $$\dfrac{\begin{array}{l}(C : (\forall \overrightarrow{\alpha : \kappa} \,.\, B_0 \to \ldots \to B_n \to K A_0 \ldots A_m, \theta_\kappa)) \in D \\ 1 \sqsubseteq r \\ \theta, \Sigma', \theta_\kappa' = \mathsf{instantiate}(\overrightarrow{\alpha : \kappa}, \theta_\kappa) \\ \Sigma, \Sigma' \vdash \theta(K A_0 \ldots A_m) \sim A \rhd \theta' \\ D;\Sigma,\Sigma';r : R \vdash\ p_i : (\theta_\kappa' \uplus \theta' \uplus \theta) B_i \ \rhd\ \Gamma_i;\theta_i\end{array}}{D;\Sigma,\Sigma';r : R \vdash\ C\,p_0 \mathinner{..} p_n : A \ \rhd\ \overrightarrow{\Gamma_i};\theta_\kappa' \uplus \theta' \uplus \theta_0 \uplus \ldots \uplus \theta_n}\ [\textsc{pC}]$$

From irrefutable $(C_1\, p_0 \mathinner{..} p_n)$ in the premise we conclude that irrefutable $p_i$ for all subpatterns by the definition of irrefutability, and that $C \in K$ and that cardinality $K \equiv 1$ and we get $\Gamma_1 = \overrightarrow{\Gamma_i}$.
By the value lemma we then have that $v = C\, v_0 \mathinner{..} v_n$, which must have been typed by (C) with repeated (APP) such that: $D; \Sigma; \Gamma'_i \vdash v_i : B_i$.
Combined with irrefutable $p_i$ we then get inductive hypotheses for all $i$ where $\Gamma_{2\,i}$ gives the typing context for each

$$(v_i \rhd p_i)t_i = t_{i+1} \;\wedge\; D; \Sigma; \Gamma_1 + r \cdot \Gamma'_i \vdash t_{i+1} : A$$

Thus by applying $(\rhd_C)$ to each inductive progress result we get progress overall with $(v_0 \rhd C\, p_0 \mathinner{..} p_n)t = t_{i+1}$.
Furthermore, by induction of $i$ we get the final typing that $D; \Sigma; G1 + r * G'0 + \ldots r * G'n| - tn + 1 : A$ since each substitution substitutions a $vi$ in for each. $\Gamma_i$ in the context $\Gamma_1$. Thus by distributivity of $\cdot$ over $+$ gives $D; \Sigma; \Gamma_1 + r \cdot \Gamma_2 \vdash t_{n+1} : A$.

□

Lemma B.1 (Approximation Preservation). *If* $\Sigma \vdash r : R$ *and* $r \sqsubseteq r'$, *then* $\Sigma \vdash r' : R$.

Proof. Suppose $\Sigma \vdash r : R$ and $r \sqsubseteq r'$. Then we know it must be the case that $r' \in R$ or else $r$ cannot be related to $r'$, and thus, $\Sigma \vdash r' : R$. □

**Lemma B.11.** [Context grading is monotonic] For all contexts $\Gamma_1, \Gamma'_1, \Gamma_2$, if $\Sigma \vdash [\Gamma_1]_R \rhd \Gamma'_1; \theta$ and $\Gamma_1 \sqsubseteq \Gamma_2$ then there exists a context $\Gamma'_2$ such that $\Sigma \vdash [\Gamma_2]_R \rhd \Gamma'_2; \theta'$. and $\Gamma'_1 \sqsubseteq \Gamma'_2$.

Proof. By induction on $\Sigma \vdash [\Gamma_1]_R \rhd \Gamma'_1; \theta$ and the definition of context grading.
Case (discAlg_em)

$$\frac{}{\Sigma \vdash [\emptyset]_R \rhd \emptyset;\, \emptyset}\;\textsc{discAlg\_em}$$

In this case $\Gamma_1 = \Gamma'_1 = \emptyset$, and this plus the assumption that $\Gamma_1 \sqsubseteq \Gamma_2$ implies that $\Gamma_2 = \emptyset$. Thus, choose $\Gamma'_2 = \emptyset$ and we obtain our result.
Case (discAlg_lin)

$$\frac{\Sigma \vdash [\Gamma]_R \rhd \Gamma';\, \theta}{\Sigma \vdash [\Gamma, x : A]_R \rhd \Gamma', x : [A]_{1:R};\, \theta}\;\textsc{discAlg\_lin}$$

In this case $\Gamma_1 = \Gamma, x : A$ and $\Gamma'_1 = \Gamma', x : [A]_{1:R}$. These imply that $\Gamma_2 = \Gamma'', x : A$ where $\Gamma \sqsubseteq \Gamma''$. By the induction hypothesis there exists a context $\Gamma'''$ such that $\Sigma \vdash [\Gamma'']_R \rhd \Gamma'''; \theta$ and $\Gamma' \sqsubseteq \Gamma'''$. Thus, by reapplying the above rule we know that $\Sigma \vdash [\Gamma'', x : A]_R \rhd \Gamma''', x : A; \theta$ and by definition of approximation $\Gamma', x : A \sqsubseteq \Gamma''', x : A$. Thus, choose $\Gamma'_2 = \Gamma''', x : A$ and we obtain our result.
Case (discAlg_grad)

$$\frac{\begin{array}{l}\Sigma \vdash [\Gamma]_R \rhd \Gamma';\, \theta' \\ \Sigma \vdash r : R' \\ \Sigma \vdash R \sqcup R' \rhd S; \iota; \theta''\end{array}}{\Sigma \vdash [\Gamma, x : [A]_r]_R \rhd \Gamma', x : [A]_{r:S};\, \theta' \uplus \theta''}\;\textsc{discAlg\_grad}$$

In this case $\Gamma_1 = (\Gamma, x : [A]_r)$, $\Gamma'_1 = (\Gamma', x : [A]_{r:S})$, and $\theta = (\theta' \uplus \theta'')$. We also know that $\Gamma_1 \sqsubseteq \Gamma_2$, thus $\Gamma_2 = (\Gamma'', x : [A]_{r'})$ where $\Gamma \sqsubseteq \Gamma''$ and $r \sqsubseteq r'$. By the induction hypothesis there exists a context $\Gamma'''$ such that $\Sigma \vdash [\Gamma'']_R \rhd \Gamma'''; \theta'$ and $\Gamma' \sqsubseteq \Gamma'''$. In addition, we know by assumption that $\Sigma \vdash r : R'$ and $r \sqsubseteq r'$, and thus, by monotonicity for grade typing (Lemma ??) we know that $\Sigma \vdash r' : R'$. Thus, by reapplying the above rule we know that $\Sigma \vdash [\Gamma'', x : [A]_{r'}]_R \rhd \Gamma''', x : [A]_{r' : S}; \theta' \uplus \theta''$ and by definition of approximation $\Gamma', x : [A]_{r:S} \sqsubseteq \Gamma''', x : [A]_{r':S}$. Thus, choose $\Gamma'_2 = \Gamma''', x : [A]_{r':S}$ and we obtain our result.

□

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

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

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

Proof. By induction on the typing derivation of $t$:

- Case (int). value $n$
- Case (var) (def) (def) . value $x$
- Case (abs) . value $(\lambda p.t)$
- Case (app)

$$\frac{\begin{array}{c} D;\Sigma;\Gamma_1 \vdash t_1 : A \to B \\ D;\Sigma;\Gamma_2 \vdash t_2 : A \end{array}}{D;\Sigma;\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \textsc{app}$$

  – Case value $t_1$ and therefore $t_1 = \lambda p.t'_1$, and therefore we must have a derivation ended in the (abs) rule for $t_1$, i.e.:

$$\frac{\begin{array}{c} D;\Sigma;- \vdash p : A \rhd \Delta;\theta \\ D;\Sigma;\Gamma_1,\Delta \vdash t'_1 : \theta B \\ \text{irrefutable } p \end{array}}{D;\Sigma;\Gamma_1 \vdash \lambda p.t'_1 : A \to B}\ \textsc{abs}$$

  ∗ Case value $t_2$ therefore let $t_2 = v$.
  By Lemma 7.3 and the derivation for $p$ and $t_1$ we can then reduce with (p$\beta$) $(\lambda p.t'_1)\ v \leadsto (v \rhd p)t'_1$ which is well-typed by Lemma 7.3 as:

$$D;\Sigma;\Gamma_1 + \Gamma_2 \vdash (v \rhd p)t'_1 : B$$

  ∗ Case ¬value $t_2$. The inductive hypothesis on $t_2$ then provides:

$$\exists t'_2, \Gamma'_2, A'.t_2 \leadsto t'_2 \ \wedge\ D;\Sigma;\Gamma'_2 \vdash t'_2 : A' \ \wedge\ \Gamma'_2 \sqsubseteq \Gamma_2 \ \wedge\ A' \leq A \qquad \text{(ih)}$$

  We can then take a step with (app$_R$):

$$\frac{t_2 \leadsto t'_2}{v\ t_2 \leadsto v\ t'_2}\ \textsc{app}_R$$

  and we can give the following typing (applying the congruence of $\leq$ with respect to types):

$$\frac{D;\Sigma;\Gamma_1 \vdash v : A \to B \qquad \dfrac{D;\Sigma;\Gamma'_2 \vdash t'_2 : A' \qquad A' \leq A}{D;\Sigma;\Gamma'_2 \vdash t'_2 : A}\ \leq}{D;\Sigma;\Gamma_1 + \Gamma'_2 \vdash v\ t'_2 : B}\ \textsc{app}$$

  and by monotonicity, from $\Gamma'_2 \sqsubseteq \Gamma_2$ we get $(\Gamma_1 + \Gamma'_2) \sqsubseteq (\Gamma_1 + \Gamma_2)$.

– Case $\neg\text{value}\ t_1$.
The inductive hypothesis on $t_1$ then provides:

$$\exists t_1', \Gamma_1', A', B'.\ t_1 \rightsquigarrow t_1' \ \wedge\ D;\Sigma;\Gamma_1' \vdash t_1' : A' \to B' \ \wedge\ \Gamma_1' \sqsubseteq \Gamma_1 \ \wedge\ (A' \to B') \leq (A \to B) \qquad \text{(ih)}$$

Deconstructing the congruence of $\leq$ on types, which is contravariant on function arguments, gives us that $A \leq A'$ and $B' \leq B$.
We can then take a step with (APP$_L$):

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{APP}_L$$

and we can give the typing by:

$$\frac{D;\Sigma;\Gamma_1' \vdash t_1' : A' \to B' \qquad \dfrac{D;\Sigma;\Gamma_2 \vdash t_2 : A \qquad A \leq A'}{D;\Sigma;\Gamma_2 \vdash t_2 : A'}\ \leq}{D;\Sigma;\Gamma_1' + \Gamma_2 \vdash t_1'\ t_2 : B'}\ \text{APP}$$

and by monotonicity, from $\Gamma_1' \sqsubseteq \Gamma_1$ we have that: $\Gamma_1' + \Gamma_2 \sqsubseteq \Gamma_1 + \Gamma_2$.

– Case (DER).

$$\frac{\begin{array}{l} D;\Sigma;\Gamma, x : A \vdash t : B \\ \Sigma \vdash R : \text{Coeffect} \end{array}}{D;\Sigma;\Gamma, x : [A]_{1:R} \vdash t : B}\ \text{DER}$$

The induction hypothesis implies:

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

If value $t$, then we are done, and so suppose the latter. Since $x$ is linear, then $\Gamma' = (\Gamma'', x : A)$, and we know $D;\Sigma;\Gamma'', x : A \vdash t' : B'$. Finally, by reapplying the dereliction typing rule we get our goal:

$$D;\Sigma;\Gamma'', x : [A]_{1:R} \vdash t' : B'$$

– Case (PR)

$$\frac{\begin{array}{l} \Sigma \vdash R : \text{Coeffect} \\ \Sigma \vdash r : \uparrow R \\ D;\Sigma;[\Gamma] \vdash t : A \end{array}}{D;\Sigma; r \cdot \Gamma \vdash [t] : \Box_r A}\ \text{PR}$$

The induction hypothesis implies:

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

If value $t$, then we are done, so suppose the latter. Then we know that $t \rightsquigarrow t'$ implies that $[t] \rightsquigarrow [t']$, and $\Gamma'$ is discharged, because of the grading operation on $[\Gamma]$. We can therefore apply the promotion rule to get:

$$\frac{\begin{array}{c} \Sigma \vdash r : \uparrow R \qquad \Sigma \vdash R : \text{Coeff} \\ D;\Sigma;[\Gamma''] \vdash t' : A' \end{array}}{D;\Sigma; r \cdot \Gamma'' \vdash [t'] : \Box_r A'}\ \text{PR}$$

By monotonicity of $\cdot$, from $\Gamma'' \sqsubseteq \Gamma$ we have $r \cdot \Gamma'' \sqsubseteq r \cdot \Gamma$ and by congruence of $\leq$ from $A' \leq A$ we have that $\Box_r A' \leq \Box_r A$.

– Case (LET$\Diamond$)

$$\frac{\begin{array}{l} D;\Sigma;\Gamma_1 \vdash t_1 : \Diamond_{\varepsilon_1} A \\ D;\Sigma;- \vdash p : A \rhd \Delta;\theta \\ D;\Sigma;\Gamma_2,\Delta \vdash t_2 : \Diamond_{\varepsilon_2}\theta B \\ \text{irrefutable } p \end{array}}{D;\Sigma;\Gamma_1 + \Gamma_2 \vdash \textbf{let}\,\langle p\rangle \leftarrow t_1\,\textbf{in}\,t_2 : \Diamond_{(\varepsilon_1 \star \varepsilon_2)} B}\ \textsc{let}\Diamond$$

Applying the induction hypothesis to the first premise implies:

$$(\text{value } t_1) \vee \exists t_1', \Gamma_1', \Diamond_{\varepsilon_1'} A'.\ t_1 \rightsquigarrow t_1' \ \wedge\ D;\Sigma;\Gamma_1' \vdash t_1' : \Diamond_{\varepsilon_1'} A' \ \wedge\ \Gamma_1' \sqsubseteq \Gamma_1 \ \wedge\ \Diamond_{\varepsilon_1'} A' \leq \Diamond_{\varepsilon_1} A \qquad \text{(ih)}$$

We have two cases to consider:

* Suppose value $t_1$. Thus by the value lemma $\exists t_1'$ such that $t_1 = \langle t_1'\rangle$. In this case, (since $t_1$) is well typed, we must have a typing derivation for $t_1'$, why by inversion gives: $D;\Sigma;\Gamma_1' \vdash t_1' : A$.
  By induction, we then know that:

  $$(\text{value } t_1') \vee \exists t_1'', \Gamma_1'' A''.\ t_1' \rightsquigarrow t_1'' \ \wedge\ D;\Sigma;\Gamma_1'' \vdash t_1'' : A'' \ \wedge\ \Gamma_1'' \sqsubseteq \Gamma_1 \ \wedge\ A'' \leq A \qquad \text{(ih)}$$

  · Case value $t_1'$ then $t_1' = v_1'$ and $t_1 = \langle v_1'\rangle$ then we can reduce by LET$_\beta$:

  $$\frac{}{\textbf{let}\,\langle p\rangle \leftarrow \langle v_1'\rangle\,\textbf{in}\,t_2 \rightsquigarrow (v_1' \rhd p)t_2}\ \textsc{let}\beta$$

  Since a typing derivation for $\langle v_1'\rangle$ must either conclude with PURE or can have PURE followed by approximation or weakening, we know that we have a judgment $D;\Sigma;\Gamma_1 \vdash t_1 : \Diamond_1 A$, thus $\varepsilon_1 = 1$ and then we by Lemma 7.3 we then have that

  $$D;\Sigma;\Gamma_1' + \Gamma_2' \vdash (v_1' \rhd p)t_2 : \Diamond_{\varepsilon_2}(\theta B)$$

  with $1 \star \varepsilon_2 = \varepsilon_2$ by unitality.

  · Otherwise, $t_1' \rightsquigarrow t_1''$ thus, we can apply (LET$_2$) to reduce inside the modality:

  $$\frac{t_1' \rightsquigarrow t_1''}{\textbf{let}\,\langle p\rangle \leftarrow \langle t_1'\rangle\,\textbf{in}\,t_2 \rightsquigarrow \textbf{let}\,\langle p\rangle \leftarrow \langle t_1''\rangle\,\textbf{in}\,t_2}\ \textsc{let}_2$$

  Then by the inductive hypothesis we have that giving:

  $$D;\Sigma;\Gamma_1'' \vdash t_1'' : A''$$

  We can then give the following typing derivation:

  $$\frac{\begin{array}{c} \dfrac{\dfrac{D;\Sigma;\Gamma_1' \vdash t_1'' : A''}{D;\Sigma;\Gamma_1' \vdash \langle t_1''\rangle : \Diamond_1 A''}\ \textsc{pure}}{D;\Sigma;\Gamma_1' \vdash \langle t_1''\rangle : \Diamond_{\varepsilon_1} A''}\ \leq \\ D;\Sigma;- \vdash p : A \rhd \Delta;\theta \\ D;\Sigma;\Gamma_2,\Delta \vdash t_2 : \Diamond_{\varepsilon_2}\theta B \\ \text{irrefutable } p \end{array}}{D;\Sigma;\Gamma_1'' + \Gamma_2 \vdash \textbf{let}\,\langle p\rangle \leftarrow \langle t_1''\rangle\,\textbf{in}\,t_2 : \Diamond_{(\varepsilon_1 \star \varepsilon_2)} B}\ \textsc{let}\Diamond$$

  where by monotonicity $\Gamma_1'' \sqsubseteq \Gamma_1$ implies $\Gamma_1'' + \Gamma_2 \sqsubseteq \Gamma_1 + \Gamma_2$.

* Suppose $\neg(\text{value } t_1)$. Recalling from above that $t_1$ has the following smaller typing derivation:

  $$D;\Sigma;\Gamma_1 \vdash t_1 : \Diamond_{\varepsilon_1} A$$

Thus, by applying the induction hypothesis we know the following:

$$(\text{value } t_1) \vee (\exists t', \Gamma'.\ \ t_1 \rightsquigarrow t_1' \ \ \wedge\ \ D;\Sigma;\Gamma_1' \vdash t_1' : \Diamond_{\varepsilon_1'} A' \ \ \wedge\ \ \Gamma_1' \sqsubseteq \Gamma_1 \ \ \wedge\ \ \Diamond_{\varepsilon_1'} A' \leq \Diamond_{\varepsilon_1} A)$$

But, we know $\neg(\text{value } t_1)$ by assumption, thus, we know the following must be the case:

$$\exists t', \Gamma'.\ \ t_1 \rightsquigarrow t_1' \ \ \wedge\ \ D;\Sigma;\Gamma_1' \vdash t_1' : \Diamond_{\varepsilon_1'} A' \ \ \wedge\ \ \Gamma_1' \sqsubseteq \Gamma_1 \ \ \wedge\ \ \Diamond_{\varepsilon_1'} A' \leq \Diamond_{\varepsilon_1} A$$

Thus, by LET$_1$ we have the following:

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let}\, \langle p \rangle \leftarrow t_1 \textbf{ in } t_2 \rightsquigarrow \textbf{let}\, \langle p \rangle \leftarrow t_1' \textbf{ in } t_2}\ \textsc{let}_1$$

It suffices to show that:

$$D;\Sigma;\Gamma_1'' + \Gamma_2'' \vdash \textbf{let}\, \langle p \rangle \leftarrow t_1' \textbf{ in } t_2 : \Diamond_{(\varepsilon_1'' \star \varepsilon_2'')} B$$

for some $\Gamma_1''$, $\Gamma_2''$, $\varepsilon_1''$, and $\varepsilon_2''$ such that $(\Gamma_1'' + \Gamma_2'') \Leftarrow (\Gamma_1 + \Gamma_2)$ and $(\varepsilon_1'' \star \varepsilon_2'') \Leftarrow (\varepsilon_1 \star \varepsilon_2)$. Recall that the application to the induction hypothesis above yielded:

$$\exists t', \Gamma'.\ \ t_1 \rightsquigarrow t_1' \ \ \wedge\ \ D;\Sigma;\Gamma_1' \vdash t_1' : \Diamond_{\varepsilon_1'} A' \ \ \wedge\ \ \Gamma_1' \sqsubseteq \Gamma_1 \ \ \wedge\ \ \Diamond_{\varepsilon_1'} A' \leq \Diamond_{\varepsilon_1} A$$

Using this we know the following:
- $D;\Sigma;\Gamma_1' \vdash t_1' : \Diamond_{\varepsilon_1'} A'$
- $A' \sqsubseteq A$
- $\Diamond_{\varepsilon_1'} A' \leq \Diamond_{\varepsilon_1} A$

The final two imply that $(\varepsilon_1' \star \varepsilon_2) \Leftarrow (\varepsilon_1 \star \varepsilon_2)$ and $D;\Sigma;\Gamma_1' \vdash t_1' : \Diamond_{\varepsilon_1'} A$. Using the latter, we can now apply the LET$\Diamond$ typing rule to obtain:

$$D;\Sigma;\Gamma_1' + \Gamma_2 \vdash \textbf{let}\, \langle p \rangle \leftarrow t_1' \textbf{ in } t_2 : \Diamond_{(\varepsilon_1' \star \varepsilon_2)} B$$

Thus, we obtain our result.

– Case (PURE)

$$\frac{D;\Sigma;\Gamma \vdash t : A}{D;\Sigma;\Gamma \vdash t : \Diamond_1 A}\ \textsc{pure}$$

This case follows from the induction hypothesis, and reapplying the rule.

– Case ($\sqsubseteq$)

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

Applying the induction hypothesis implies:

$$\begin{aligned}(\text{value } t) \vee \exists t', \Gamma_1', \Gamma_2', s', B'.\ t \rightsquigarrow t' \wedge {} & D;\Sigma;\Gamma_1', x : [A]_{r'}, \Gamma_2' \vdash t' : B' \\ & \wedge (\Gamma_1', x : [A]_{r'}, \Gamma_2') \sqsubseteq (\Gamma, x : [A]_r, \Gamma') \\ & \wedge B' \leq B\end{aligned} \qquad \text{(ih)}$$

If value $t$, then we are done, so suppose the latter. Then from the above we have that $r' \sqsubseteq r$ which by transitivity with approximation in the premise implies $r' \sqsubseteq s$ thus the inductive hypothesis $D;\Sigma;\Gamma_1', x : [A]_{r'}, \Gamma_2' \vdash t' : B'$ satisfies the form of the lemma.

$\square$