Graded Call-By-Push-Value

ANONYMOUS AUTHOR(S)

Call-by-push-value (CBPV) is a simply typed lambda calculus that polarizes types into value and computation types and can thus express both call-by-name and call-by-value evaluation in the presence of effects. Semantically, effects are modeled by a monad, and computation types as algebras over this monad.

Effect type systems usually express more information than the presence of an effect; often effects are categorized by preordered monoid where the monoid operation represents accumulation of effects and the order expresses effect subsumption, in analogy to subtyping. In this work, a *graded* version of CBPV is presented where the typing of computations likens effect typing. Semantically, computation types are then represented as graded monad algebras.

Observing that the value types of CBPV can be interpreted as comonad coalgebras, we further present a version of CBPV that has coeffects graded by a preordered semiring. Value types and contexts are interpreted as graded comonad coalgebras, allowing resource-aware interpretations of CBPV.

Finally, we combine the two systems into a fully graded version of CBPV where both effects and coeffects are graded. This turns out to be possible without specifying any interaction between effects and coeffects.

Additional Key Words and Phrases: effects, coeffects, call-by-name, call-by-value, linear types

1 INTRODUCTION

Levy's call-by-push-value calculus (CBPV) [2006] is a simply typed lambda calculus with disjoint sums and eager and lazy products that allows the modelling of "lazy" effects. For instance, the CBPV-program

$$\mathsf{print}\,\text{"}\texttt{function}\text{"}.\,\lambda x.\,\mathsf{print}\,\text{"}\texttt{argument}\text{"}.\,x$$

of type $P \Rightarrow \diamond P$, will, unlike the corresponding ML-program, not print the word "function" and return a function; it will simply wait for an input value of type P. Only after an argument has been supplied, the program will print the words "function" and "argument" and then return the argument. The reason is that effects in CBPV can only be observed at *value types* like P, and function types are not value types but computation types. Effectful actions (like printing) at computation types N are "pushed down" the type structure until they reach a position at a monadic type $\diamond P$ where they can be executed. Semantically, this is facilitated by interpreting computation types as monad algebras with an action $\mathsf{run} : \mathsf{T}\,N \to N$ that allows to formally run effects transported by the monad T at type N, even though N is not directly a monadic type $\diamond P$.

CBPV polarizes types into value types P and computation types N where the former embed into the latter as $\diamond P$ by the formal "monad" $\diamond$ and the latter embed into the former as $\Box N$ by "thunking" $\Box$. In Levy's *behavioral semantics*, functor $\diamond$ is a left adjoint to functor $\Box$ and need not be a monad. Quite the opposite: when modelling store, $\diamond P = P \times S$ is a comonad and $\Box N = S \to N$ a monad. Contrastingly, in the above explained *algebra semantics*, $\diamond$ is indeed modeled by a monad T, yet $\Box$ just by the identity. It is however possible to model $\Box$ by a monoidal comonad D, and positive types by comonad coalgebras with an action $\mathsf{expose} : P \to \mathsf{D}\,P$ that makes the services of the comonad available at all value types, not just at $\Box N$.

Building on these observations, we investigate in this article how CBPV can keep track of not just the presence of an effect or coeffect but also the *kind* of effect and coeffect. Semantically, precise information about effects can be obtained by using a family of monads T_e graded over effect classifiers e drawn from a preordered monoid [Katsumata 2014]. For CBPV, we generalize the concept of monad algebras N to *graded* such ones, N_e. Syntactically, computation typing

2021. 2475-1421/2021/1-ART1 $15.00
https://doi.org/

$\Gamma \vdash t : N \mid e$ is extended by an effect classifier e (as usual in type and effect systems [Nielson and Nielson 1999]) such that a computation can be interpreted by a morphism $\Gamma \to N_e$.

Coeffect type systems [Brunel et al. 2014; Ghica and Smith 2014; Petricek et al. 2014] have been developed to track resource consumption by attaching usage information to each variable in the typing context. Coeffect typing generalizes linear typing [Girard 1987] to quantitative typing [Atkey 2018; McBride 2016; Sergey et al. 2014] and subsumes sensitivity analysis [Reed and Pierce 2010] and static information control flow [Volpano et al. 1996] aka security typing [Abadi et al. 1999]. Coeffects can be modeled by a comonad D_r graded over resource qualifiers r drawn from a preordered semiring. For the sake of coeffect-graded CBPV, we generalize graded comonads to graded comonad coalgebras. Value types P and contexts Γ are then interpreted as such coalgebras. Value typing $\gamma\Gamma \vdash v : P$ and computation typing $\gamma\Gamma \vdash t : N$ are based on linear typing and come equipped with a resource context γ. Semantically, values are interpreted as morphisms $\Gamma_{r\gamma} \to P_r$ allowing the "multiplication" or *scaling* of a value by r and computations as morphisms $\Gamma_\gamma \to N$.

Finally, effect and coeffect graded CBPV can be combined into a fully graded CBPV calculus, surprisingly without sorting out any interaction between effects and coeffects, such as the distributive laws of Gaboardi et al. [2016]. We credit the smoothness of the integration to the careful placement of monad and comonad in CBPV's type system, so that scaling is restricted to values and does not arise for computations.

Contributions.

(1) We introduce graded monad algebras and an effect-graded version of CBPV in Section 2, after recapitulating monads and their algebras and graded monad.
(2) We further introduce graded comonad coalgebras and a coeffect-graded version of CBPV in Section 3. We give its denotational and operational semantics and adapt Atkey and Wood's substitution theorem [2019; 2020] to this version of CBPV.
(3) We present an effect- and coeffect-graded version of CBPV in Section 4.

Preliminaries. The reader should bring some elementary knowledge of category theory, such as the interpretation of simply-typed lambda calculus in cartesian-closed categories. However, we try to be gentle with categorical concepts such as monads and recapitulate their definition where needed. In many cases, it is sufficient to think in terms of the category SET where objects are sets and morphisms functions between sets, or in the functor category $[C \to \mathsf{SET}]$ where objects are monotone families of sets $(A_i)_{i:C}$ indexed by objects i of C such that $C_i \to C_j$ in SET for $i \to j$ in C and morphisms are natural transformations $(f_i : A_i \to B_i)_{i:C}$.

2 AN EFFECT-GRADED VERSION OF CBPV

2.1 Recapitulation: modelling effects via monads

In this section, we recapitulate monads and some essential vocabulary of category theory. There are no surprises, thus, the experienced reader is invited to skip this section.

Consider a categorical model C of the simply-typed lambda calculus (STLC), i.e., where types τ and contexts Γ are interpreted as objects $[\![\tau]\!]$ and $[\![\Gamma]\!]$ of C and terms $\Gamma \vdash t : \tau$ as morphisms $(\!|t|\!) \in C([\![\Gamma]\!], [\![\tau]\!])$. Such a category could be SET, interpreting types as sets and terms as functions, mapping the valuation of their free variables to their value, or CPO, interpreting types as complete partial orders and terms as monotone functions, or a presheaf model of the STLC etc. Typically, C is a cartesian-closed category, i.e., has products $A_1 \times A_2$ of objects to model product types and contexts, and exponentials $A \Rightarrow B$ to model function types. Further C maybe be distributive, *i.e.*, have coproducts $A_1 + A_2$ that distribute over products, to model variants aka disjoint sum types.

Recall that effects are modeled by a suitable monad $\mathsf{T} : C \to C$ in C. We shall refer to elements of a monadic type $\mathsf{T}\,A$ as *computations* when $C = \mathsf{SET}$. For a general C, computations shall be the morphisms $C\,(A,\ \mathsf{T}\,B)$ called *Kleisli arrows*. An example monad would be the *writer monad* $\mathsf{T}\,A = \mathsf{String} \times A$ that models the effect *output*. Another example would be the *exception monad* $\mathsf{Exc} + \_$ where the result of a computation is either an exception $e : \mathsf{Exc}$ or a regular result. The monad operations, together with their implementation for *writer* in SET, are the following:

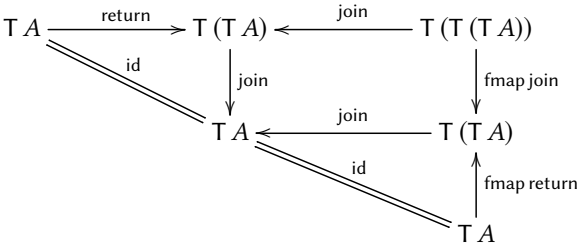
$$\begin{array}{lcl} \mathsf{fmap}_\mathsf{T} & : & (A \to B) \to \mathsf{T}\,A \to \mathsf{T}\,B \\ \mathsf{fmap}_\mathsf{T}\ f\ (s,\ a) & = & (s,\ f\ b) \\ \mathsf{return}_\mathsf{T} & : & A \to \mathsf{T}\,A \\ \mathsf{return}_\mathsf{T}\ a & = & (\texttt{""},\ a) \\ \mathsf{join}_\mathsf{T} & : & \mathsf{T}\,(\mathsf{T}\,A) \to \mathsf{T}\,A \\ \mathsf{join}_\mathsf{T}\ (s_1, (s_2, a)) & = & (s_1 \mathbin{+\!\!+} s_2,\ a) \end{array}$$

Herein, "" shall denote the empty string and ++ string concatenation.

Monad unit return (written η in category speak) turns a value into a pure computation and monad multiplication join (written μ in category speak) combines two effects and thus allows sequencing of computations via *Kleisli composition* $(g : C\,(B,\ \mathsf{T}\,C)) \circ_\mathsf{T} (f : C\,(A,\ \mathsf{T}\,B)) = (\mathsf{join} \circ \mathsf{fmap}\ g \circ f : C\,(A,\ \mathsf{T}\,C))$.

The presence of $\mathsf{fmap} : C\,(A,\ B) \to C\,(\mathsf{T}\,A,\ \mathsf{T}\,B)$ satisfying the functor laws $\mathsf{fmap}\ \mathsf{id} = \mathsf{id}$ and $\mathsf{fmap}\,(f \circ g) = \mathsf{fmap}\ f \circ \mathsf{fmap}\ g$ makes T an *(endo)functor*, written $\mathsf{T} : [C \to C]$. The endofunctors $F : [C \to C]$ form a category, the *functor category*, with identity $\mathsf{Id}\,A = A$ and composition $(F \circ G)\,A = F\,(G\,A)$. We shall write $f : F \dot{\to} G$ for morphisms in the functor category, called *natural transformations*. These are families $f_A : C\,(F\,A,\ G\,A)$ of morphisms that commute with the functor action fmap, *i.e.*, $\mathsf{fmap}_G\ h \circ f_A = f_B \circ \mathsf{fmap}_F\ h$ for any $h : C\,(A,\ B)$.

Unit $\mathsf{return} : \mathsf{Id} \dot{\to} \mathsf{T}$ and multiplication $\mathsf{join} : \mathsf{T} \circ \mathsf{T} \dot{\to} \mathsf{T}$ are *natural transformations*—which breaks down to $\mathsf{fmap}_\mathsf{T}\ h \circ \mathsf{return} = \mathsf{return} \circ h$ and $\mathsf{fmap}_\mathsf{T}\ h \circ \mathsf{join} = \mathsf{join} \circ \mathsf{fmap}\ (\mathsf{fmap}\ h)$. The three equational laws of join can be visualized compactly in the following commutative diagram.



2.2 Effect algebras and graded monads

Following Katsumata [2014] we can obtain more information about effects using a monad T_e *graded* over elements e of a suitable effect algebra Eff. An effect algebra be a preordered monoid $(\mathsf{Eff}, \bullet, \varepsilon, \leq)$ such that $\_\bullet\_$ is monotone wrt. the preorder $\leq$ in both arguments. The unit ε shall mean *no effect* and the operation $\_\bullet\_$ serves to accumulate effects, possibly in a sequential order—unless the monoid is commutative. The preorder represents effect subsumption, i.e., loss in precision of the effect analysis. Note that the unit ε is not necessarily the least element wrt. $\leq$.

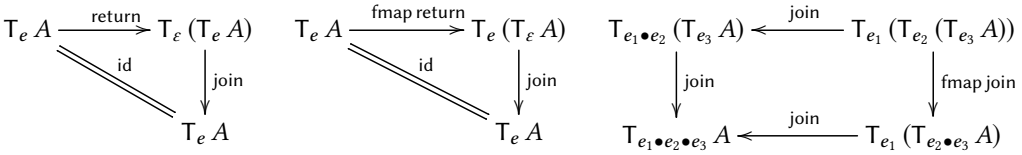
Say we want to track an upper bound on the length of the output produced by a program. To this end, we can use the preordered monoid $\mathsf{Eff} = \mathbb{N} \cup \{\infty\}$ under addition $\_\bullet\_ = \_+\_$ with unit $\varepsilon = 0$ and the natural order $\leq$. The effect ∞ then denotes unbounded output, or output whose length we cannot track in the type system (*e.g.*, when it depends on some variable). Increasing the

upper bound along $\leq$ means loss of precision of our analysis, with ∞ the least precise information, meaning no upper bound. The corresponding graded writer monad is $\mathsf{T}_e\, A = (\mathsf{String}_{\leq}\, e) \times A$ where the output is an element of $\mathsf{String}_{\leq}\, e$, a string of length at most $e \in \mathbb{N} \cup \{\infty\}$. The operations of a graded monad are, again given with their SET-implementation for the graded writer:

$$\begin{array}{lcl} \mathsf{fmap} & : & (A \to B) \to \mathsf{T}_e\, A \to \mathsf{T}_e\, B \\ \mathsf{fmap}\, f\, (s,\ a) & = & (s,\ f\, b) \\ \\ \mathsf{return} & : & A \to \mathsf{T}_\varepsilon\, A \\ \mathsf{return}\, a & = & (\text{""}, a) \\ \\ \mathsf{join} & : & \mathsf{T}_{e_1}\, (\mathsf{T}_{e_2}\, A) \to \mathsf{T}_{e_1 \bullet e_2}\, A \\ \mathsf{join}\, (s_1, (s_2, a)) & = & (s_1 +\!\!+ s_2,\ a) \\ \\ \mathsf{cast} & : & \mathsf{T}_e\, A \to \mathsf{T}_{e'}\, A \text{ for } e \leq e' \\ \mathsf{cast} & = & \mathsf{id} \end{array}$$

A graded version of the exception monad would use effect algebra $\mathsf{Eff} = \mathcal{P}\,\mathsf{Exc}$ under union and subset; an effect e is a set of possible exceptions thrown by a computation.[1]

The interesting laws for graded monads are given by the following commutative diagrams.



Further, cast commutes with fmap, namely $\mathsf{fmap}\, f \circ \mathsf{cast} = \mathsf{cast} \circ \mathsf{fmap}\, f$, and in two ways with join, namely $\mathsf{join} \circ \mathsf{cast} = \mathsf{cast} \circ \mathsf{join}$ and $\mathsf{join} \circ \mathsf{fmap}\, \mathsf{cast} = \mathsf{cast} \circ \mathsf{join}$. We may write $\mathsf{T}_{e_1 \leq e_2}$ for the natural transformation $\mathsf{cast} : \mathsf{T}_{e_1} \dot{\to} \mathsf{T}_{e_2}$.

Remark 1 (Eff-graded monad are lax monoidal functors $\mathsf{T} : [\mathsf{Eff} \to [C \to C]]$). The concept of a graded monad can be more succinctly expressed by using more advanced language of category theory. (This reformulation is not essential for the remainder of the exposition and may be skipped on first reading.)

Recall that a *monoidal category* $\mathcal{E}$ has a designated object $\mathsf{I} : \mathcal{E}$ and an operation $\_\otimes\_ : [\mathcal{E} \to [\mathcal{E} \to \mathcal{E}]]$, the *tensor product* on objects of $\mathcal{E}$ that is functorial in both positions. Further, there are natural isomorphisms $\lambda : (\mathsf{I} \otimes A) \cong A$ and $\rho : (A \otimes \mathsf{I}) \cong A$ witnessing the unitality of I and $\alpha : (A \otimes (B \otimes C)) \cong ((A \otimes B) \otimes C)$ witnessing associativity of $\otimes$.

Any preordered monoid, such as $(\mathsf{Eff}, \_ \bullet \_, \varepsilon, \leq)$, makes a monoidal category $\mathcal{E} = \mathsf{Eff}$ with Homset $\mathcal{E}(e, e') = \{() \mid e \leq e'\}$, tensor $\otimes = \bullet$ and unit $\mathsf{I} = \varepsilon$. Further, this category is *thin*, *i.e.*, there is at most one morphism between any two objects e, e'.

A graded monad $\mathsf{T} : [\mathsf{Eff} \to [C \to C]]$ is then a morphism from monoidal category $(\mathsf{Eff}, \bullet, \varepsilon)$ to monoidal category $([C \to C], \circ, \mathsf{Id})$. The monoidal structure in the latter is just functor composition and identity functor. Operations cast and fmap witness the functoriality of T in its first and second argument. The operations return and join witness (in a directed way) the preservation of unit ε and multiplication $\bullet$ by T, making T a *lax monoidal functor*. (End of remark.)

2.3 CBPV, monad algebras, and their graded version

Call-by-push value [Levy 2006] is a refinement of Moggi's computational lambda-calculus [1991] that allows effects not only in monadic types, *i.e.*, in objects $\mathsf{T}\, A$, but more generally in computation

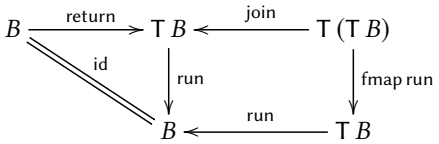
[1] An example of a exception-tracing type system is the Java language.

types. These correspond to monad algebras for T in C, aka T-algebras. Those algebras are objects B together with a morphism $\mathsf{run}_B : \mathsf{T}\,B \to B$ that allows to formally *run* the monad, "merging" its effects into B. The prime example of a monad algebra is simply a monadic type, because $\mathsf{run}_{\mathsf{T}\,A} : \mathsf{T}\,(\mathsf{T}\,A) \to \mathsf{T}\,A$ is just join. Levy [2006] shows that monad algebras are closed under products ($\times$) and exponentiation ($\Rightarrow$) with arbitrary objects. *E.g.*, in SET we can define, for *writer* algebras:

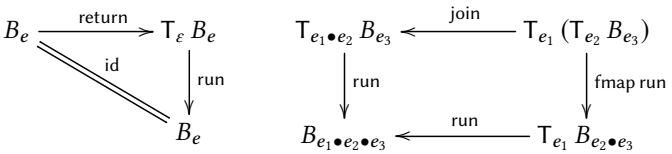
$$\begin{array}{lcl} \mathsf{run}_{B\times B'} & : & \mathsf{T}\,(B\times B') \to B\times B' \\ \mathsf{run}_{B\times B'}\,(s,(b,b')) & = & (\mathsf{run}_B(s,b),\ \mathsf{run}_{B'}(s,b')) \\ \mathsf{run}_{A\Rightarrow B} & : & \mathsf{T}\,(A\Rightarrow B) \to (A\Rightarrow B) \\ \mathsf{run}_{A\Rightarrow B}\,(s,f) & = & \lambda a.\ \mathsf{run}_B\,(s,\ f\ a) \end{array}$$

These definition implement *lazy* effects that cannot be observed at computation types such as $A \Rightarrow B$, but only at value types; the run of the monad algebra pushes the effects towards result types that are eventually types of observable objects (values).

With run being a generalization of join, the laws for run are in analogy of those for join (if B were $\mathsf{T}\,A$):



In graded CBPV, monad algebras get replaced by *graded monad algebras*. Given a graded monad T, a T-algebra is a family of objects $(B_e)_{e:\mathsf{Eff}}$ and morphisms $\mathsf{run}_B : \mathsf{T}_{e_1}\,B_{e_2} \to B_{e_1\bullet e_2}$ satisfying these laws:



Further the family B should be functorial in the sense that there is a family of coercion morphisms $B_{e_1\le e_2} : B_{e_1} \to B_{e_2}$ with $B_{e\le e} = \mathsf{id}$ and $B_{e_2\le e_3} \circ B_{e_1\le e_2} = B_{e_1\le e_3}$.

Graded monad algebras are, like non-graded ones, closed under pointwise products $(B\times B')_e = B_e \times B'_e$ and exponentiation with objects $(A\Rightarrow B)_e = A \to B_e$; here the graded writer example:

$$\begin{array}{lcl} \mathsf{run}_{B\times B'} & : & \mathsf{T}_{e_1}\,(B\times B')_{e_2} \to (B\times B')_{e_1\bullet e_2} \\ \mathsf{run}_{B\times B'}\,(s,(b,b')) & = & (\mathsf{run}_B(s,b),\ \mathsf{run}_{B'}(s,b')) \\ \mathsf{run}_{A\Rightarrow B} & : & \mathsf{T}_{e_1}\,(A\Rightarrow B)_{e_2} \to (A\Rightarrow B)_{e_1\bullet e_2} \\ \mathsf{run}_{A\Rightarrow B}\,(s,f) & = & \lambda a.\ \mathsf{run}_B\,(s,\ f\ a) \end{array}$$

Remark 2 (Naturality of run). The T-algebra $B : [\mathsf{Eff} \to C]$ comes with a *natural* morphism $\mathsf{run}_{e_1,e_2} : C\,(\mathsf{T}_{e_1}\,B_{e_2},\ B_{e_1\bullet e_2})$ in the sense that $B_{(e_1\le e'_1)\bullet(e_2\le e'_2)} \circ \mathsf{run}_{e_1,e_2} = \mathsf{run}_{e'_1,e'_2} \circ \mathsf{T}_{e_1\le e'_1}\,B_{e_2\le e'_2}$.

2.4 Effect-graded CBPV: syntax and typing

With the theory of graded monads in place, we design a graded version of CBPV. The syntax and typing rules for effect-graded CBPV are given in Fig. 1. The differences to pure CBPV are in gray boxes.

Types are classified into value types $P \in \mathsf{Ty}^+$ (written A in Levy [2006]) and computation types $N \in \mathsf{Ty}^-$ (written $\underline{B}$ in *loc. cit.*). These are positive and negative types in the terminology of focusing

Types.

$$\begin{array}{llllll} \mathsf{Ty}^+ & \ni & P & ::= & [e]N \mid o \mid \Sigma_{i:I} P_i \mid \otimes_{i:I} P_i & \text{Value types (positive types)} \\ \mathsf{Ty}^- & \ni & N & ::= & \diamond P \mid P \Rightarrow N \mid \Pi_{i:I} N_i & \text{Computation types (negative types)} \\ \mathsf{Cxt} & \ni & \Gamma & ::= & \emptyset \mid \Gamma.x{:}P & \text{Typing context} \end{array}$$

Terms.

$$\begin{array}{llllll} \mathsf{Tm}^+ & \ni & v, w & ::= & x \mid \mathsf{thunk}\, t \mid \mathsf{in}_i\, v \mid \mathsf{tup}\, \bar{v} & \text{Values (positive terms)} \\ \mathsf{Tm}^- & \ni & t, u & ::= & v \,\mathsf{be}\, x.\, t & \text{Computations (negative terms):} \\ & & & \mid & \mathsf{force}\, v \mid v \,\mathsf{cases}\, \{\overline{x.\, t}\} \mid v \,\mathsf{split}\, \bar{x}.\, t & \text{value eliminations} \\ & & & \mid & \mathsf{ret}\, v \mid u \,\mathsf{to}\, x.\, t & \text{monad operations} \\ & & & \mid & \lambda x.\, t \mid t\, v & \text{functions} \\ & & & \mid & \mathsf{record}\{\overline{i : t}\} \mid \mathsf{proj}_i\, t & \text{lazy tuples (records)} \end{array}$$

Value typing $\boxed{\Gamma \vdash v : P}$.

$$\textsc{var}\ \frac{x{:}P \in \Gamma}{\Gamma \vdash x : P} \qquad \square\textsc{-intro}\ \frac{\Gamma \vdash t : N \mid e}{\Gamma \vdash \mathsf{thunk}\, t : [e]N}$$

$$\Sigma\textsc{-intro}\ \frac{\Gamma \vdash v : P_i}{\Gamma \vdash \mathsf{in}_i\, v : \Sigma_I P} \qquad \otimes\textsc{-intro}\ \frac{\forall i{:}I,\ \Gamma \vdash v_i : P_i}{\Gamma \vdash \mathsf{tup}\, v : \otimes_I P}$$

Computation typing $\boxed{\Gamma \vdash t : N \mid e}$.

$$\textsc{let}\ \frac{\Gamma \vdash v : P \qquad \Gamma.x{:}P \vdash t : N \mid e}{\Gamma \vdash v \,\mathsf{be}\, x.\, t : N \mid e} \qquad \square\textsc{-elim}\ \frac{\Gamma \vdash v : [e]N}{\Gamma \vdash \mathsf{force}\, v : N \mid e}$$

$$\Sigma\textsc{-elim}\ \frac{\Gamma \vdash v : \Sigma_I P \qquad \forall i{:}I,\ \Gamma.x_i{:}P_i \vdash t_i : N \mid e}{\Gamma \vdash v \,\mathsf{cases}\, \{x_i.\, t_i\}_{i:I} : N \mid e} \qquad \otimes\textsc{-elim}\ \frac{\Gamma \vdash v : \otimes_I P \qquad \Gamma.\overline{x_i{:}P_i}^{\,i:I} \vdash t : N \mid e}{\Gamma \vdash v \,\mathsf{split}\, \bar{x}.\, t : N \mid e}$$

$$\diamond\textsc{-intro}\ \frac{\Gamma \vdash v : P}{\Gamma \vdash \mathsf{ret}\, v : \diamond P \mid \varepsilon} \qquad \diamond\textsc{-elim}\ \frac{\Gamma \vdash u : \diamond P \mid e_1 \qquad \Gamma.x{:}P \vdash t : N \mid e_2}{\Gamma \vdash u \,\mathsf{to}\, x.\, t : N \mid e_1 \bullet e_2}$$

$$\Rightarrow\textsc{-intro}\ \frac{\Gamma.x{:}P \vdash t : N \mid e}{\Gamma \vdash \lambda x.\, t : P \Rightarrow N \mid e} \qquad \Rightarrow\textsc{-elim}\ \frac{\Gamma \vdash t : P \Rightarrow N \mid e \qquad \Gamma \vdash v : P}{\Gamma \vdash t\, v : N \mid e}$$

$$\Pi\textsc{-intro}\ \frac{\forall i{:}I,\ \Gamma \vdash t_i : N_i \mid e}{\Gamma \vdash \mathsf{record}\{i : t_i\}_{i:I} : \Pi_I N \mid e} \qquad \Pi\textsc{-elim}\ \frac{\Gamma \vdash t : \Pi_I N \mid e}{\Gamma \vdash \mathsf{proj}_i\, t : N_i \mid e}$$

$$\textsc{sub}\ \frac{\Gamma \vdash t : N \mid e}{\Gamma \vdash t : N \mid e'}\ e \leq e'$$

Fig. 1. Effect-graded call-by-push-value.

[Zeilberger 2009]. Positive types are generated from base types o via disjoint sums $\Sigma_{i:I} P_i$ with tag set I, eager products $\otimes_{i:I} P_i$ (composed from 1 and $A_1 \times A_2$ in *loc. cit.*) of arity I and thunking $[e]N$ (written $U\, \underline{B}$ in *loc. cit.*). In contrast to pure CBPV, thunk types $[e]N$ are annotated with an effect e that can be triggered when the thunk is forced. Negative types are just as in pure CBPV: monadic

types $\diamond P$ (written $F\,A$ in *loc. cit.*), function types $P \Rightarrow N$ (written $A \to \underline{B}$) in *loc. cit.*) and record types $\Pi_{i:I} N_i$ with label set I. Records are *lazy* tuples whose components are only computed by demand. We abbreviate $\Sigma_{i:I} P_i$ by its "meta-level η-contraction" $\Sigma_I P$; notations $\otimes_I P$ and $\Pi_I N$ are understood analogously.

Terms are separated into values $v \in \mathsf{Tm}^+$ and computations $t \in \mathsf{Tm}^-$ and are identical to pure CBPV, modulo changes in the concrete syntax. Values introduce positive types, computations introduce negative types and eliminate both positive and negative types. We use bars to indicate sequences, e.g., $\bar{v}$ for a sequence of values, but drop the bar when the context of discourse makes clear that we are dealing with sequences rather than single objects. For instance, "v_i" (where $i : I$) in the premise of $\otimes$-intro indicates that v is a sequence of values with elements $(v_i)_{i:I}$. We may abbreviate record$\{i : t_i\}_{i:I}$ by record$_I\, t$ where I is the label set and t a mapping from labels $i : I$ to terms t_i.

The meaning of the term constructors is best understood via their typing. Typing contexts Γ are finite maps from variables x to value types P, with $\Gamma.x{:}P$ denoting the update of the finite map Γ at key x with value P.

Value typing $\Gamma \vdash v : P$ is just as in pure CBPV, however, computation typing $\Gamma \vdash t : N \mid e$ also records effects $e : \mathsf{Eff}$ potentially produced at runtime by computation t. Thunking a computation (rule $\Box$-intro) stores the inferred effect classifier e in the thunk type $[e]N$.

Effects are accumulated via the introduction and elimination rule for the graded monad. The unit ret v of the monad is effect-free ($\diamond$-intro); running this computation just produces the pure value v. Sequencing computations u and t via the *bind* construct "u to x. t" composes the effects e_1 of u with the effects e_2 of t in that order. The intuition is that first u is run, producing effects classified by e_1, and its result is bound to x to run t, producing effects classified by e_2. The sum of the effects is classified by $e_1 \bullet e_2$.

The other introduction and elimination rules are just as in pure CBPV, except that they propagate the effect classifier e from hypothesis to conclusion. Note that in case distinction (Σ-elim) and record construction (Π-intro) all subterms t_i are required to produce effects classified by the same e. However, in reality, different branches of *e.g.* a case distinction may produce very different effects. To end up with a unique classifier e, the branches may have to be typed using effect subsumption (sub). In fact, the uses of sub can be confined to the hypotheses of Σ-elim and Π-intro, except for a final invocation of sub at the very end of the typing derivation. Alternatively, we could have introduced effect algebras with suprema $\sup_{i:I} e_i$ instead of a preorder $e \le e'$. However, suprema might not always exist; by using subsumption sub, we delegate the problem of partiality to the construction of a typing derivation.

Remark 3. We recover pure CBPV from graded CBPV using the trivial effect algebra $\mathsf{Eff} = \{\varepsilon\}$.

2.5 Effect graded CBPV: denotational semantics

The denotational semantics of the novel parts of graded CBPV has been informally explained in sections 2.1 to 2.3 already; in the following, we spell out the details. We assume a distributive cartesian-closed category C with a strong graded monad $\mathsf{T} : [\mathsf{Eff} \to [C \to C]]$. Let us agree on some notation for the constructions on objects and morphisms:

- Product $\prod_{i:I} A_i$ with projections $\pi_i : C(\prod_I A, A_i)$ and tupling $\langle f_i \rangle_{i:I} : C(C, \prod_i A)$ for $f_i : C(C, A_i)$. Binary products $\prod_{\{1,2\}} A$ are written $A_1 \times A_2$, and the nullary product (terminal object) is written 1 with nullary tupling $\langle\rangle : C(C, 1)$.
- Coproduct $\coprod_{i:I} A_i$ with injections $\iota_i : C(A_i, \coprod_I A)$, cotupling $[f_i]_{i:I} : C(\coprod_I A, B)$ for $f_i : C(A_i, B)$, and distribution morphism dist : $C\,(C \times \coprod_I A,\ \coprod_{i:I} (C \times A_i))$.
- Exponential $A \Rightarrow B$ with $\Lambda : C(C \times A, B) \to C(C, A \Rightarrow B)$ and eval : $C((A \Rightarrow B) \times A, B)$.

- Graded monad T with functoriality $\mathsf{fmap}_{\mathsf{T}}\ f : C(\mathsf{T}_e A, \mathsf{T}_e B)$ for $f : C(A, B)$, unit $\mathsf{return} : C(A, \mathsf{T}_\varepsilon A)$, multiplication $\mathsf{join} : C(\mathsf{T}_{e_1}(\mathsf{T}_{e_2} A), \mathsf{T}_{e_1 \bullet e_2} A)$, strength $\mathsf{strength}^{\mathsf{l}} : C(A \times \mathsf{T}_e B, \mathsf{T}_e(A \times B))$ and coercion $\mathsf{cast} : C(\mathsf{T}_e A, \mathsf{T}_{e'} A)$ for $e \le e'$. Costrength $\mathsf{strength}^{\mathsf{r}} : C(\mathsf{T}_e A \times B, \mathsf{T}_e(A \times B))$ is derivable in the standard way.

2.5.1 Interpretation of types. Positive types P are interpreted as objects $[\![P]\!]^+$ of C, where a denotation $[\![o]\!]^+ : C$ of base types o is assumed. This interpretation lifts to contexts Γ via $[\![\Gamma]\!]^+ = \prod_{x:\mathrm{dom}(\Gamma)} [\![\Gamma(x)]\!]^+$.

Negative types N are interpreted as functors $[\![N]\!]^- : [\mathsf{Eff} \to C]$ mapping effect classifiers e to objects $[\![N]\!]^-_e$ and effect subsumption $e \le e'$ to morphisms $[\![N]\!]^-_{e \le e'} : [\![N]\!]^-_e \to [\![N]\!]^-_{e'}$.

$$
\begin{array}{lcl@{\qquad}lcl}
[\![\_]\!]^+ & : & \mathsf{Ty}^+ \to C & [\![\_]\!]^- & : & \mathsf{Ty}^- \to \mathsf{Eff} \to C \\
[\![[e]N]\!]^+ & = & [\![N]\!]^-_e & [\![\diamond P]\!]^-_e & = & \mathsf{T}_e [\![P]\!]^+ \\
[\![\Sigma_I P]\!]^+ & = & \coprod_{i:I} [\![P_i]\!]^+ & [\![P \Rightarrow N]\!]^-_e & = & [\![P]\!]^+ \Rightarrow [\![N]\!]^-_e \\
[\![\otimes_I P]\!]^+ & = & \prod_{i:I} [\![P_i]\!]^+ & [\![\Pi_I N]\!]^-_e & = & \prod_{i:I} [\![N_i]\!]^-_e
\end{array}
$$

The graded T-algebra structure $\mathsf{run}_{[\![N]\!]^-}$ is constructed by induction on N, as well as functoriality $[\![N]\!]^-_{e_1 \le e_2}$:

$$
\begin{array}{lcl@{\qquad}rcl}
\mathsf{run}_{[\![N]\!]^-} & : & C\,(\mathsf{T}_{e_1}([\![N]\!]^-_{e_2}),\ [\![N]\!]^-_{e_1 \bullet e_2}) & [\![N]\!]^-_{e_1 \le e_2} & : & C\,([\![N]\!]^-_{e_1},\ [\![N]\!]^-_{e_2}) \\
\mathsf{run}_{[\![\diamond P]\!]^-} & = & \mathsf{join} & [\![\diamond P]\!]^-_{e_1 \le e_2} & = & \mathsf{cast} \\
\mathsf{run}_{[\![P \Rightarrow N]\!]^-} & = & \Lambda(\mathsf{run}_{[\![N]\!]^-} \circ \mathsf{fmap}_{\mathsf{T}}\ \mathsf{eval} \circ \mathsf{strength}^{\mathsf{r}}) & [\![P \Rightarrow N]\!]^-_{e_1 \le e_2} & = & \Lambda([\![N]\!]^-_{e_1 \le e_2} \circ \mathsf{eval}) \\
\mathsf{run}_{[\![\Pi_I N]\!]^-} & = & \langle \mathsf{run}_{[\![N_i]\!]^-} \circ \mathsf{fmap}_{\mathsf{T}}\ \pi_i \rangle_{i:I} & [\![\Pi_I N]\!]^-_{e_1 \le e_2} & = & \langle [\![N]\!]^-_{e_1 \le e_2} \circ \pi_i \rangle_{i:I}
\end{array}
$$

This construction is the same as Levy [2006], modulo grading. The algebra laws ensue. That the coercions $[\![N]\!]^-_{e_1 \le e_2}$ satisfy identity and composition—the functor laws for $[\![N]\!]^-$—is easy to verify.

2.5.2 Interpretation of terms. Values $\Gamma \vdash v : P$ are interpreted as morphisms $(\!|v|\!) : C\,([\![\Gamma]\!]^+,\ [\![P]\!]^+)$ just as in pure CBPV. Computations $\Gamma \vdash t : N \mid e$ are interpreted as morphisms $(\!|t|\!) : C\,([\![\Gamma]\!]^+,\ [\![N]\!]^-_e)$. Most of the cases are straightforward and in analogy to CBPV, so let us focus on the modified rules where grading comes into play. A subtlety is that we interpret *typing derivations* rather than terms, because rule SUB is a silent construction on raw terms but becomes a coercion in the denotational semantics. The most natural way to make this precise is to use intrinsically well-typed syntax. Going from typing rules to such well-typed syntax is a routine transformation which we do not spell out here. The reader interested in well-typed syntax for CBPV is referred to Abel and Sattler [2019]. The other subtlety, that we ignore variable names in the interpretation, can also be made precise by well-typed syntax which uses de Bruijn indices.

As in Levy's algebra semantics [2006], creation and forcing of thunks is invisible in the model. The bind operation ($\diamond$-ELIM) utilizes the generalization of join to run. Subsumption is interpreted by functoriality in Eff.

$$
\begin{array}{lcl}
\Box\text{-INTRO} & (\!|\mathsf{thunk}\ t|\!) & = (\!|t|\!) \\
\Box\text{-ELIM} & (\!|\mathsf{force}\ v|\!) & = (\!|v|\!) \\
\diamond\text{-INTRO} & (\!|\mathsf{ret}\ v|\!) & = \mathsf{return} \circ (\!|v|\!) \\
\diamond\text{-ELIM} & (\!|u\ \mathsf{to}\ x.\ t|\!) & = \mathsf{run}_{[\![N]\!]^-} \circ \mathsf{fmap}_{\mathsf{T}}\ \mathsf{eval} \circ \mathsf{strength}^{\mathsf{l}} \circ \langle \Lambda(\!|t|\!),\ (\!|u|\!) \rangle \\
\text{SUB} & (\!|t|\!) & = [\![N]\!]^-_{e \le e'} \circ (\!|t|\!)
\end{array}
$$

These definitions are understood in the context of the given typing rules.

2.6 Example effects

We replay some of Levy's [2006] effect examples in graded CBPV.

Printing. Outputting a fixed string s before computing t is facilitated by "print s. t" which we type by the following rule:

$$\frac{\Gamma \vdash t : N \mid m}{\Gamma \vdash \mathsf{print}\, s.\, t : N \mid n + m}\; s : \mathsf{String}_{\leq} n$$

Herein, we use effect algebra $(\mathbb{N} \cup \{\infty\}, +, 0, \leq)$ and graded monad $\mathsf{T}_n A = \mathsf{String}_{\leq} n \times A$. This allows us to implement a family of morphisms $\mathsf{output} : \mathsf{String}_{\leq} n \to \mathsf{SET}(1, \mathsf{T}_n\, 1)$ to interpret the print statement as $(\!|\mathsf{print}\, s.\, t|\!) = \mathsf{run}_{[\![N]\!]} \circ \mathsf{fmap}_{\mathsf{T}}\, \pi_2 \circ \mathsf{strength}^{\mathsf{r}} \circ \langle \mathsf{output}\, s \circ \langle\rangle,\, (\!|t|\!)\rangle$:

$$[\![\Gamma]\!] \xrightarrow{\langle \mathsf{output}\, s \circ \langle\rangle,\, (\!|t|\!)\rangle} \mathsf{T}_n\, 1 \times [\![N]\!]^-_m \xrightarrow{\mathsf{strength}^{\mathsf{r}}} \mathsf{T}_n(1 \times [\![N]\!]^-_m) \xrightarrow{\mathsf{fmap}_{\mathsf{T}}\, \pi_2} \mathsf{T}_n\, [\![N]\!]^-_m \xrightarrow{\mathsf{run}} [\![N]\!]^-_{n+m}$$

Exceptions. Given a set Exc of exceptions whose elements we refer to by e and whose subsets by E, consider the effect algebra $(\mathcal{P}\,\mathsf{Exc}, \cup, \emptyset, \subseteq)$. Primitives for throwing and catching exceptions can be added to graded CBPV by the following rules:

$$\frac{}{\Gamma \vdash \mathsf{throw}\, e : N \mid \{e\}} \qquad \frac{\Gamma \vdash u : N \mid E_1 \qquad \Gamma \vdash t : N \mid E_2}{\Gamma \vdash u\, \mathsf{catch}\, e \mapsto t : N \mid E_1 \setminus \{e\} \cup E_2}$$

If u throws exception e, then "u catch $e \mapsto t$" computes t, else u. In SET we use the graded monad $\mathsf{T}_E\, A = E + A$ and a family of morphism $\mathsf{raise}\, e : \mathsf{SET}(1, \mathsf{T}_{\{e\}} A)$ to interpret $(\!|\mathsf{throw}\, e|\!) = \mathsf{raise}\, e \circ \langle\rangle$. To interpret catch, define a family $\mathsf{handle}_N\, e : \mathsf{SET}\,(N_{E_1} \times N_{E_2},\, N_{E_1 \setminus \{e\} \cup E_2})$ by induction on $N \in \mathsf{Ty}^-$:

$$\begin{array}{lll} \mathsf{handle}_{\diamond P}\quad e\,(g,h) & = & \begin{cases} h & \text{if } g = \iota_1 e \\ g & \text{otherwise} \end{cases} \\ \mathsf{handle}_{P \Rightarrow N}\; e\,(g,h)\,a & = & \mathsf{handle}_N\, e\,(g\,a, h\,a) \\ \mathsf{handle}_{\otimes_I N}\; e\,(g,h)\,i & = & \mathsf{handle}_{N_i}\, e\,(g\,i, h\,i) \end{array}$$

Catching is then $(\!|u\, \mathsf{catch}\, e \mapsto t|\!) = \mathsf{handle}_N\, e \circ \langle (\!|u|\!), (\!|t|\!)\rangle$. Note that handle cannot be defined in terms of run, but we have to break N down to monadic type $\diamond P$ to get access to the exception thrown by u.

2.7 Digression: grading via a partial monoid

Not all sequences of effects are always meaningful: For instance, reading from a file before it was opened is impossible and could be prevented statically by graded effect typing. This could be modelled by adding a maximal element $\top : \mathsf{Eff}$ (with $e \leq \top$ for all $e : \mathsf{Eff}$) that signifies an inconsistent state. This error element would also be dominant in sequences, *i.e.*, $\top \bullet e = e \bullet \top = \top$. A program $\Gamma \vdash t : N \mid e$ would only be accepted if $e \neq \top$.

Alternatively, we could work with a partial monoid, *i.e.*, a carrier Eff with a predicate $\varepsilon \leq \_$ and a ternary relation $\_ \bullet \_ \leq \_$ such that the following laws hold:

(1) Unit: $e \leq e' :\Longleftrightarrow \exists e_0.\, \varepsilon \leq e_0 \wedge e_0 \bullet e \leq e'$ iff $\exists e_0.\, \varepsilon \leq e_0 \wedge e \bullet e_0 \leq e'$.
(2) Associativity: $\exists e_{12}.\, e_1 \bullet e_2 \leq e_{12} \wedge e_{12} \bullet e_3 \leq e_{123}$ iff $\exists e_{23}.\, e_1 \bullet e_{23} \leq e_{123} \wedge e_2 \bullet e_3 \leq e_{23}$.
(3) Monotonicity of $\varepsilon \leq \_$: If $\varepsilon \leq e$ and $e \leq e'$ then $\varepsilon \leq e'$.
(4) Monotonicity of $\_ \bullet \_ \leq \_$: If $e_1' \leq e_1$ and $e_2' \leq e_2$ and $e \leq e'$ and $e_1 \bullet e_2 \leq e$ then $e_1' \bullet e_2' \leq e'$.
(5) Reflexivity and transitivity of $\leq$.

The typing rules for $\diamond$ would change accordingly.

$$\diamond\text{-INTRO}\;\frac{\Gamma \vdash v : P}{\Gamma \vdash \mathsf{ret}\, v : \diamond P \mid e}\;\boxed{\varepsilon \leq e} \qquad \diamond\text{-ELIM}\;\frac{\Gamma \vdash u : \diamond P \mid e_1 \qquad \Gamma.x{:}P \vdash t : N \mid e_2}{\Gamma \vdash u\, \mathsf{to}\, x.\, t : N \mid e}\;\boxed{e_1 \bullet e_2 \leq e}$$

Rule SUB would be admissible.

A monad $\mathsf{T} : \mathsf{Eff} \to [C \to C]$ graded by a partial monoid Eff has natural transformations $\mathsf{Id} \dot{\to} \mathsf{T}_e$ for $\varepsilon \leq e$ and $\mathsf{T}_{e_1} \circ \mathsf{T}_{e_2} \dot{\to} \mathsf{T}_e$ for $e_1 \bullet e_2 \leq e$. A T-algebra B has morphism $\mathsf{run}_B : \mathsf{T}_{e_1} B_{e_2} \to B_e$ for $e_1 \bullet e_2 \leq e$.

Remark 4. Another way of restricting effect composition is to use a 2-category Eff where the objects i, j, k are state types, the morphisms e effect classifiers and the 2-cells effect subsumption $e \leq e'$. The necessary theory has been worked out by Orchard et al. [2020]. An example would be a typed state monad $\mathsf{T}_{e:i\to j} A = S_i \to (A \times S_j)$ where the state type S_i is indexed, and effects $e : i \to j$ may only make valid modifications to the state. For instance, the index could be a set of pointers denoting the allocated heap cells and a read/write/deallocate effect would require the respective pointer to be a member of this set (and remove it in case of deallocate).

3 COEFFECT-GRADED CBPV

CBPV places the monad $\diamond P$ at the transition from positive types to negative types. Dually, the transition $\Box N$ from negative types to positive types is a vessel that can be filled with a comonad. Just like negative types N are monad algebras, positive types P can be *comonad coalgebras*.

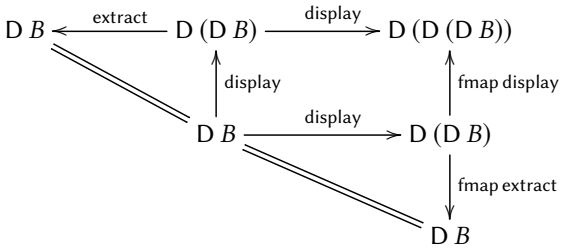
3.1 Comonadic CBPV and their comonad coalgebras

Let us consider the *stream comonad*[2] $\mathsf{D}\, B = \mathbb{N} \to B$, with $\mathsf{head}\, s = s\, 0$ and $\mathsf{tail}\, s = s \circ (\_+1)$. The stream comonad lets us work in a setting where values are not single data points, but streams of data. Besides functoriality, a comonad has the natural transformations extract and display (in category theory called ε and δ), dual to return and join. The implementation of the stream comonad in SET is as follows:

$$\begin{array}{lcl} \mathsf{extract} & : & \mathsf{D}\, B \to B \\ \mathsf{extract}\, s & = & \mathsf{head}\, s \\ \\ \mathsf{display} & : & \mathsf{D}\, B \to \mathsf{D}\, (\mathsf{D}\, B) \\ \mathsf{head}\, (\mathsf{display}\, s) & = & s \\ \mathsf{tail}\quad (\mathsf{display}\, s) & = & \mathsf{display}\, (\mathsf{tail}\, s) \end{array}$$

The generic operations of a comonad allow us to *extract* the value wrapped in a comonadic structure and to *display* another layer of that structure.

The comonad laws are a simple dualization of the monad laws:



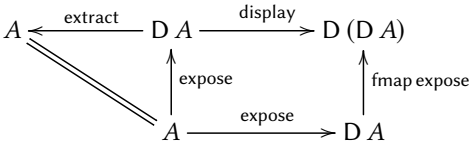
A *monoidal* comonad implements a morphism zip that combines a tuples of comonadic values into one comonadic tuple—here implemented for the stream comonad:

$$\begin{array}{lcl} \mathsf{zip} & : & \otimes_{i:I} (\mathsf{D}\, B_i) \to \mathsf{D}(\otimes_I B) \\ \mathsf{zip}\, (s_i)_{i:I}\, n & = & (s_i\, n)_{i:I} \end{array}$$

Many other comonads are monoidal, like the store comonad $(S \Rightarrow \_) \times S$, though not all, *e.g.*, the context comonad $S \times \_$. In this article, we consider only monoidal comonads.

[2] https://bartoszmilewski.com/2017/01/02/comonads/

If type constructor $\Box$ is interpreted as a monoidal comonad D, then all positive types P can be interpreted as D-coalgebras $A = [\![P]\!]$, *i.e.*, implement a morphism $\mathsf{expose}_A : A \to \mathsf{D}\,A$ satisfying the laws of a comonad coalgebra:



The implementation of $\mathsf{expose}_{[\![P]\!]}$ proceeds by induction on P.

$$\begin{array}{lcl}
\mathsf{expose}_{[\![\Box N]\!]} & : & \mathsf{D}\,[\![N]\!] \to \mathsf{D}\,(\mathsf{D}\,[\![N]\!]) \\
\mathsf{expose}_{[\![\Box N]\!]} & = & \mathsf{display} \\
\mathsf{expose}_{[\![\otimes_I P]\!]} & : & \otimes_{i:I}[\![P_i]\!] \to \mathsf{D}\,(\otimes_{i:I}[\![P_i]\!]) \\
\mathsf{expose}_{[\![\otimes_I P]\!]} & = & \mathsf{zip} \circ \otimes_{i:I} \mathsf{expose}_{[\![P_i]\!]} \\
\mathsf{expose}_{[\![\Sigma_I P]\!]} & : & \Sigma_{i:I}[\![P_i]\!] \to \mathsf{D}\,(\Sigma_{i:I}[\![P_i]\!]) \\
\mathsf{expose}_{[\![\Sigma_I P]\!]} & = & \left[\mathsf{fmap}\,\iota_i \circ \mathsf{expose}_{[\![P_i]\!]}\right]_{i:I}
\end{array}$$

Positive base types $o \in \mathsf{Ty}^+$ are required to be D-coalgebras. For instance, elements of a base type like $\mathsf{Float} \in \mathsf{Ty}^+$ could represent streams of floating point numbers, *e.g.*, continuous measurements from a sensor.

Because comonad coalgebras are closed under products for monoidal coalgebras, contexts Γ are interpreted as comonad coalgebras $[\![\Gamma]\!]$. In the presence of a comonad interpretation of $\Box$, the operations thunk and force are no longer the identity, but "generalized cobind" and extract:

$$\Box\text{-INTRO}\quad \frac{\Gamma \vdash t : N}{\Gamma \vdash \mathsf{thunk}\,t : \Box N} \qquad (\!|\mathsf{thunk}\,t|\!) \;=\; \mathsf{fmap}_{\mathsf{D}}\,(\!|t|\!) \circ \mathsf{expose}_{[\![\Gamma]\!]}$$

$$\Box\text{-ELIM}\quad \frac{\Gamma \vdash v : \Box N}{\Gamma \vdash \mathsf{force}\,v : N} \qquad (\!|\mathsf{force}\,v|\!) \;=\; \mathsf{extract} \circ (\!|v|\!)$$

The map $(\!|\mathsf{thunk}|\!) : \mathcal{C}\,(A,\,B) \to \mathcal{C}\,(A,\,\mathsf{D}\,B)$ generalizes $\mathsf{cobind} : \mathcal{C}\,(\mathsf{D}\,C,\,B) \to \mathcal{C}\,(\mathsf{D}\,C,\,\mathsf{D}\,B)$ to D-coalgebras A in the same way that the monadic $\mathsf{bind} : \mathcal{C}\,(A,\,\mathsf{T}\,D) \to \mathcal{C}\,(\mathsf{T}\,A,\,\mathsf{T}\,D)$ is generalized to $\mathcal{C}\,(A,\,B) \to \mathcal{C}\,(\mathsf{T}\,A,\,B)$ for T-algebras B in CBPV.

3.2 Graded comonads and their coalgebras

Graded comonads have been utilized to give semantics to context-dependent computation [Petricek et al. 2014]. For grading, *loc. cit.* uses a *resource algebra* in form of a preordered semiring $(\mathsf{R}, +, 0, \cdot, 1, \leq)$. The semantics is *resource aware* and thus not constructed in a cartesian-closed category but in a symmetric monoidal closed category $(\mathcal{C}, \otimes, \mathsf{I}, \multimap)$. There, introduction Λ and elimination eval of exponentials are $\Lambda : \mathcal{C}\,(C \otimes A,\,B) \to \mathcal{C}\,(C,\,A \multimap B)$ and $\mathsf{eval} : \mathcal{C}\,((A \multimap B) \otimes A,\,B)$. The tensor product $\otimes$ is a bifunctor and it is customary to overload $\otimes$ for the functorial action $f_1 \otimes f_2 : \mathcal{C}\,(A_1 \otimes A_2,\,B_1 \otimes B_2)$ where $f_i : \mathcal{C}\,(A_i,\,B_i)$. We shall also use I-ary products $\otimes_{i:I} A_i$ and their functorial action $\otimes_{i:I} f_i$. Further, the symmetric monoidal structure is usually witnessed by natural isomorphisms for left $\lambda_A : \mathcal{C}\,(\mathsf{I} \otimes A,\,A)$ and right unit $\rho_A : \mathcal{C}\,(A \otimes I,\,A)$, associativity $\alpha_{A,B,C} : \mathcal{C}\,(A \otimes (B \otimes C),\,(A \otimes B) \otimes C)$ and swap $\sigma_{A,B} : \mathcal{C}\,(A \otimes B,\,B \otimes A)$. We shall summarize combinations of these isomorphisms under the name $\mathsf{iso}^{\otimes} : \mathcal{C}\,(\otimes_I A,\,\otimes_J B)$ when the multisets $\{A_i\}_{i:I}$ and $\{B_j\}_{j:J}$ coincide modulo addition and deletion of units (I).

As a running example for resource accounting, we use the semiring $\mathsf{R} = \mathcal{P}\,\mathbb{N} \setminus \emptyset$ of *multiplicities* ordered by $\supseteq$ with pointwise sum and product, *e.g.*, $r + s = \{n + m \mid n \in r, m \in s\}$. Subsemirings of

R have been used for quantitative typing [Atkey 2018; McBride 2016; Sergey et al. 2014]. The order expresses precision of the quantities, *e.g.*, $\{1\} \geq \{0,1\} \geq \mathbb{N}$ states that linear use of a variable is more informative than affine use than unrestricted use.

A matching symmetric monoidal closed category C can be obtained as follows: Assume a preordered commutative monoid $(\mathsf{W}, \oplus, 0_\mathsf{W}, \sqsubseteq, \sqcup)$ that is also a $\sqcup$-semilattice with distribution law $\sqcup_{i:I}(w_i \oplus w'_i) \sqsubseteq \sqcup_I w \oplus \sqcup_I w'$. The elements ("worlds") can be seen as collection of available resources under choice. The order $w \sqsubseteq w'$ again expresses precision of information, *i.e.*, if we can construct something from resources w we can also construct it from w' where we have additional choices. The supremum $\sqcup_{i:I} w_i$ expresses we can build the "thing" from any of the w_i. Multiplication $n \cdot w$ is understood as $w \oplus \cdots \oplus w$ with n summands. Given a commutative monoid $(\mathsf{M}, \uplus, \emptyset)$ whose elements can be thought of as bags of atomic resources, an instance for W would be $\mathsf{W} = \mathcal{P}\,\mathsf{M}$ under $\sqsubseteq = \subseteq$ and pointwise union $w \oplus w' = \{m \uplus m' \mid m \in w, m' \in w'\}$.

Resource qualifiers $r \in \mathsf{R}$ operate on worlds via $r \cdot w = \sqcup_{n \in r}(n \cdot w)$. For instance the *affine* qualifier $\{0,1\}w = 0_\mathsf{W} \oplus w$ gives us the choice of using resources w or not (0_W). It is routine to verify that W is almost a left semimodule to R, *i.e.*, the following laws hold:

$$\begin{array}{rclcll}
1 \cdot w & = & \sqcup_{n \in \{1\}} nw & = & 1w & = & w \\
(rs) \cdot w & = & \sqcup_{n \in r, m \in s}(nm)w & = & \sqcup_{n \in r} n(\sqcup_{m \in s} mw) & = & r \cdot (s \cdot w) \\
0 \cdot w & = & \sqcup_{n \in \{0\}} nw & = & 0w & = & 0_\mathsf{W} \\
(r+s) \cdot w & = & \sqcup_{n \in r, m \in s}(n+m)w & = & \sqcup_{n \in r} \sqcup_{m \in s} (nw \oplus mw) & = & (r \cdot w) \oplus (s \cdot w) \\
r \cdot 0_\mathsf{W} & = & \sqcup_{n \in r} n 0_\mathsf{W} & = & \sqcup_{n \in r} 0_\mathsf{W} & = & 0_\mathsf{W} \\
r \cdot (w \oplus w') & = & \sqcup_{n \in r} n(w \oplus w') & \sqsubseteq & \sqcup_{n \in r} nw \oplus \sqcup_{n \in r} nw' & = & (r \cdot w) \oplus (r \cdot w')
\end{array}$$

Because the last law is not an equality but just the inequality $r(w \oplus w') \sqsubseteq rw \oplus rw'$, we have "almost" a semimodule.

Objects of C are functors $A : [(\mathsf{W}, \sqsubseteq) \to \mathsf{SET}]$ and morphisms $f : C\,(A,\, B)$ are natural transformations $(f_w : A_w \to B_w)_{w:W}$, *i.e.*, $B_{w \sqsubseteq w'} \circ f_w = f_{w'} \circ A_{w \sqsubseteq w'}$. The tensor $A \otimes B$ is Day's convolution $(A \otimes B)_w = \bigcup_{w_1 \oplus w_2 \sqsubseteq w}(A_{w_1} \times B_{w_2})$ with unit $\mathsf{I}_w = \bigcup_{0_\mathsf{W} \sqsubseteq w} 1$. The unit I is constructible at world w from nothing (the unit set 1) if $0_\mathsf{W} \sqsubseteq w$, *i.e.*, if the world w includes the choice of using no resources. A tensor $A \otimes B$ is constructible at world w if w includes the choice to split the resources into w_1 and w_2 to construct A and B, resp. The exponential $A \multimap B$ is determined by currying $C \otimes A \multimap B \cong C \multimap (A \multimap B)$ and given by $(A \multimap B)_w = \bigcap_{w_1 \oplus w \sqsubseteq w_2}(A_{w_1} \to B_{w_2})$.

The thus constructed symmetric monoidal closed category C has a R-graded comonad

$$\begin{array}{lcl}
\mathsf{D} & : & [(\mathsf{R}, \leq) \to [C \to C]] \\
(\mathsf{D}_r A)_w & = & \bigcup_{rw' \sqsubseteq w} A_{w'}
\end{array}$$

implementing trivially the following natural transformations. Herein, we use as second monoidal structure on the functor category $[C \to C]$, the pointwise tensor product of functors $(\dot\otimes_{i:I} F_i)\,A = \otimes_{i:I}(F_i\,A)$ in $[C \to C]$, in particular $\dot{\mathsf{I}}\,A = \mathsf{I}$.

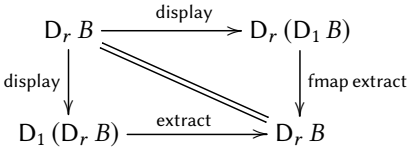
$$\begin{array}{lcl}
\mathsf{extract} & : & \mathsf{D}_1 \dot\to \mathsf{Id} \\
\mathsf{display} & : & \mathsf{D}_{rs} \dot\to \mathsf{D}_r \circ \mathsf{D}_s \\
\mathsf{drop} & : & \mathsf{D}_0 \dot\to \dot{\mathsf{I}} \\
\mathsf{duplicate} & : & \mathsf{D}_{r+s} \dot\to \mathsf{D}_r \mathbin{\dot\otimes} \mathsf{D}_s
\end{array}$$

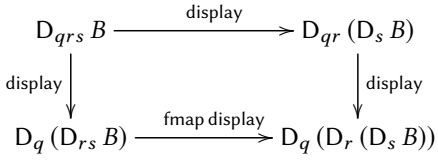
For example, duplicate takes $a \in \bigcup_{rw' \oplus sw' \sqsubseteq w} A_{w'}$ to $(a,a) \in \bigcup_{w_1 \oplus w_2 \sqsubseteq w} \bigcup_{rw'_1 \leq w_1} \bigcup_{sw'_2 \leq w_2} A_{w'_1} \times A_{w'_2}$. The intermediate worlds are $w'_1 = w'_2 = w'$ and $w_1 = rw'$ and $w_2 = sw'$.

The monoidal character of D is witnessed by the trivial morphism zip:

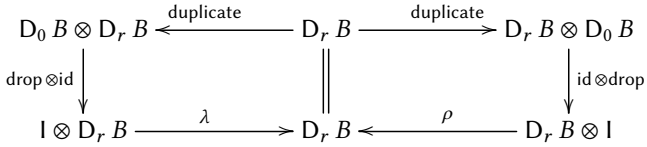
$$\begin{array}{lcl} \mathsf{zip} & : & C\,(\otimes_{i:I}(\mathsf{D}_r\,A_i),\ \mathsf{D}_r\,(\otimes_I A)) \\ \mathsf{zip}_w & : & \bigcup_{\oplus_{i:I} w_i \sqsubseteq w} \Pi_{i:I}(\mathsf{D}_r\,A_i)_{w_i} \to \bigcup_{r w' \sqsubseteq w} (\otimes_I A)_{w'} \\ \mathsf{zip}_w & : & \bigcup_{\oplus_{i:I} w_i \sqsubseteq w} \Pi_{i:I}(\bigcup_{r w'_i \sqsubseteq w_i} A_{i w'_i}) \to \bigcup_{r w' \sqsubseteq w} \bigcup_{\oplus_{i:I} w'_i \sqsubseteq w'} \Pi_{i:I} A_{i w'_i} \\ \mathsf{zip}_w\,(a_i)_{i:I} & = & (a_i)_{i:I} \end{array}$$

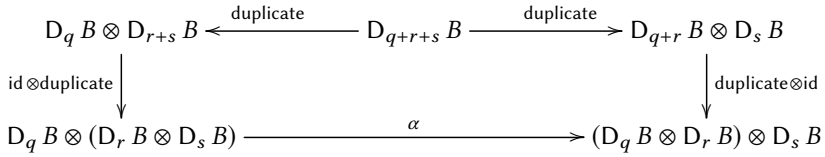
The comonad laws generalize to graded comonads as follows:

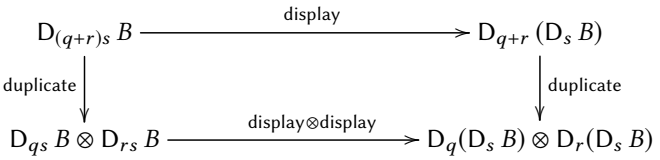


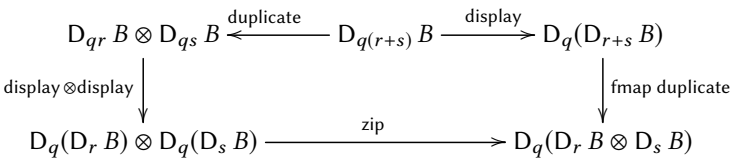


These laws reflect that D maps the multiplicative monoid $(\mathsf{R}, \cdot, 1)$ to the monoid structure $([C \to C], \circ, \mathsf{Id})$ of composition in the functor category $[C \to C]$. Similar laws need to hold for the additive monoid and distributivity.

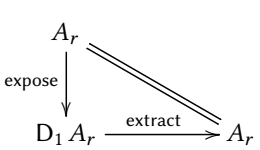


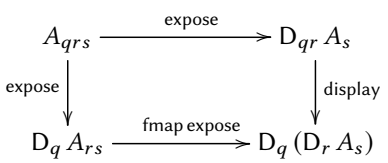






A D-coalgebra is a functor $A : [(\mathsf{R}, \leq) \to C]$ with family of morphisms $\mathsf{expose}_{r,s} : A_{rs} \to \mathsf{D}_r\,A_s$ natural in r and s. For the D-coalgebra $\mathsf{D}\_\,B$ the family expose is just display, and the suitably generalizable laws for display are required to hold for expose:

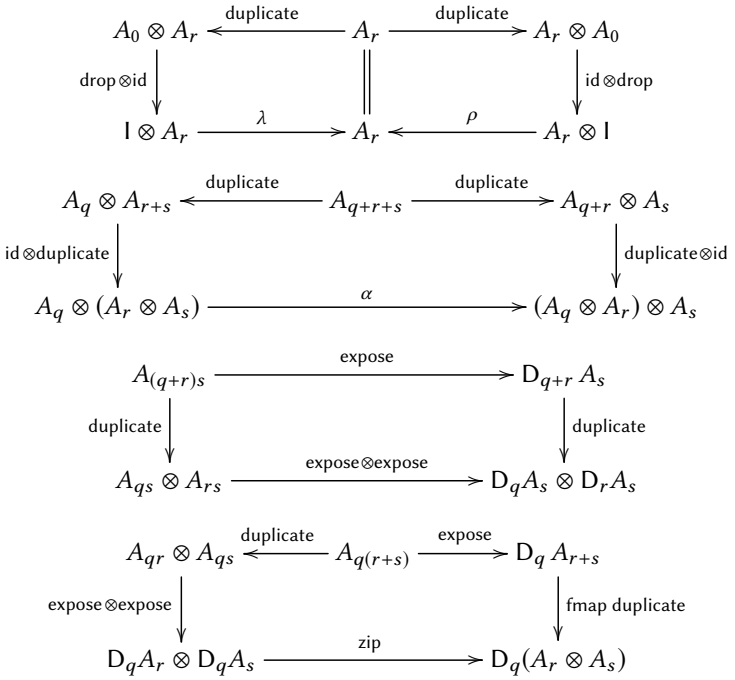




Further, A needs to map the additive monoidal structure $(\mathsf{R}, +, 0)$ to the monoidal structure $(C, \otimes, \mathsf{I})$ associated with the tensor product in C. We choose to overload the names duplicate and drop here to accommodate for the generalization from R-comonads to R-comonad algebras:

$$\begin{array}{lcl} \mathsf{duplicate} & : & C\,(A_{r+s},\, A_r \otimes A_s) \\ \mathsf{drop} & : & C\,(A_0,\, \mathsf{I}) \end{array}$$

The laws of duplicate and drop for comonadic objects $\mathsf{D}_r\, B$ immediately generalize to comonad algebras A_r:



Graded comonad algebras are closed under pointwise sums $(\Sigma_I A)_r = \Sigma_{i:I} A_{i,r}$ and products $(\otimes_I A)_r = \otimes_{i:I} A_{i,r}$.

$$\begin{array}{lcl} \mathsf{expose}_{\Sigma_I A} & : & C\,((\Sigma_I A)_{rs},\, \mathsf{D}_r(\Sigma_I A)_s) \\ \mathsf{expose}_{\Sigma_I A} & = & \left[\mathsf{fmap}\ \iota_i \circ \mathsf{expose}_{A_i}\right]_{i:I} \\ \mathsf{drop}_{\Sigma_I A} & : & C\,((\Sigma_I A)_0,\, \mathsf{I}) \\ \mathsf{drop}_{\Sigma_I A} & = & \left[\mathsf{drop}_{A_i}\right]_{i:I} \\ \mathsf{duplicate}_{\Sigma_I A} & : & C\,((\Sigma_I A)_{r+s},\, (\Sigma_I A)_r \otimes (\Sigma_I A)_s) \\ \mathsf{duplicate}_{\Sigma_I A} & = & \left[(\iota_i \otimes \iota_i) \circ \mathsf{duplicate}_{A_i}\right]_{i:I} \\ \\ \mathsf{expose}_{\otimes_I A} & : & C\,((\otimes_I A)_{rs},\, \mathsf{D}_r(\otimes_I A)_s) \\ \mathsf{expose}_{\otimes_I A} & = & \mathsf{zip} \circ \otimes_{i:I} \mathsf{expose}_{A_i} \\ \mathsf{drop}_{\otimes_I A} & : & C\,((\otimes_I A)_0,\, \mathsf{I}) \\ \mathsf{drop}_{\otimes_I A} & = & \mathsf{iso}^{\otimes} \circ \otimes_{i:I} \mathsf{drop}_{A_i} \\ \mathsf{duplicate}_{\otimes_I A} & : & C\,((\otimes_I A)_{r+s},\, (\otimes_I A)_r \otimes (\otimes_I A)_s) \\ \mathsf{duplicate}_{\otimes_I A} & = & \mathsf{iso}^{\otimes} \circ \otimes_{i:I} \mathsf{duplicate}_{A_i} \end{array}$$

This enables us to interpret all positive types of CBPV as graded comonad coalgebras.

3.3 Structured graded comonads and their algebras

Coeffect, quantitative, and many modal type systems maintain a typing context where each variable x is annotated by a resource qualifier r in addition to its type P. Such a typing context $(r_1 x_1{:}P_1, \ldots, r_n x_n{:}P_n) =: \gamma\Gamma$ can be split into a pure typing context $\Gamma = x_1{:}P_1, \ldots, x_n{:}P_n$ and a resource context $\gamma = x_1{:}r_1, \ldots, x_n{:}r_n$ such that $\operatorname{dom}\gamma = \operatorname{dom}\Gamma$. We shall freely mix the two notations as it suits our purpose.

In coeffect-graded CBPV, we wish to interpret each type in the context as a D-coalgebra for a fixed graded comonad D. The interpretation $[\![\Gamma]\!]$ of the context Γ should then be a comonad coalgebra over the grading γ such that judgements $\gamma\Gamma \vdash t : N$ can be interpreted as morphisms $(\!|t|\!) : C\,([\![\Gamma]\!]_\gamma,\ [\![N]\!]^-)$. Since each variable comes with its own resource qualifier, we cannot simply model the context as a tensor product $\otimes_{x:\operatorname{dom}\Gamma}[\![\Gamma(x)]\!]$ of D-coalgebras since this product would be indexed by a single resource qualifier $r : \mathsf{R}$ rather than a resource context $\gamma : \operatorname{dom}\Gamma \to \mathsf{R}$. The solution offered by Petricek et al. [2014] are *structured indexed comonads*. We shall generalize this to comonad coalgebras to the extend needed for interpreting contexts.

McBride [2016] observed that resource contexts $\gamma : \mathsf{R}^I$ form a left R-semimodule under pointwise addition $(\gamma + \delta)(i) = \gamma(i) + \delta(i)$ and scaling $(r \cdot \gamma)(i) = r \cdot \gamma(i)$. Contexts Γ can thus be interpreted as (*structured* or) $\mathsf{R}^{\operatorname{dom}\Gamma}$*-graded* D-coalgebras C with the following operations:

$$\begin{array}{lcl} \mathsf{drop}_C & : & C\,(C_0,\ \mathsf{I}) \\ \mathsf{duplicate}_C & : & C\,(C_{\gamma+\delta},\ C_\gamma \otimes C_\delta) \\ \mathsf{duplicate}_C & : & C\,(C_{\Sigma_I \gamma},\ \otimes_{i:I} C_{\gamma_i}) \\ \mathsf{expose}_C & : & C\,(C_{r\gamma},\ \mathsf{D}_r\, C_\gamma) \end{array}$$

The second, generalized form of duplicate will be used to split resources accumulated from I parties. Note that $A_r := C_{r\gamma}$ flattens a structured D-coalgebra C into an ordinary one, A.

We will also need to interpret context extension. To this end, we shall employ a structured product $(C \boxtimes D)_{\gamma.\delta} = C_\gamma \otimes D_\delta$ where $\gamma : \mathsf{R}^I$ and $\delta : \mathsf{R}^J$ and thus $C \boxtimes D$ is a R^{I+J}-graded D-coalgebra. Further, we implicitly use the isomorphism between 1-structured D-coalgebras $C : \mathsf{R}^1 \to C$ and R-graded D-coalgebras $A : \mathsf{R} \to C$ and define

$$[\![\Gamma.x{:}P]\!] = [\![\Gamma]\!] \boxtimes [\![P]\!]^+,$$

sweeping name issues under the carpet (to be handled by de Bruijn indices).

3.4 Coeffect-graded CBPV: syntax and typing

We now have the mathematical structures in place to define a coeffect-graded variant of CBPV (see Fig. 2). The difference to pure CBPV is laid out in *gray boxes*.

The fundamental novelty is that the typing judgements $\gamma\Gamma \vdash v : P$ and $\gamma\Gamma \vdash t : N$ are equipped with resource contexts γ matching the typing contexts Γ. A common pattern in the rules is that resource requirements of the subterms are *added* when *both* subterms are or may be evaluated at runtime (rules $\otimes$-INTRO, LET, $\Sigma/\otimes/\diamond/\multimap$-ELIM). Note that in Σ-ELIM, the branches t_i of the case statement (Σ-ELIM) share a resource context γ since only one of the branches is executed at runtime. Via rule WEAK, the different resource requirements of the branches can be subsumed under their maximum wrt. $\leq_{\mathsf{R}}$. Similarly, the components t_i of a record (Π-INTRO) share a resource context since projection only retrieves one of the components (Π-ELIM). This is the opposite of eager tuples ($\otimes$-INTRO) where the elimination makes all components available at the same time ($\otimes$-ELIM), thus, all components have to be evaluated at runtime.

Function types $rP \multimap N$ are now graded by a resource qualifier r that specifies how the function argument is to be used in the function body. (This type is often written $!_r P \multimap N$.) In quantitative typing with $\mathsf{R} = \mathcal{P}\,\mathbb{N}$, qualifier r gives the possible usage quantities of the function argument, e.g.

Types.

$$\begin{array}{llllll}
\mathsf{Ty}^{+} & \ni & P & ::= & \Box N \mid o \mid \Sigma_{i:I} P_i \mid \otimes_{i:I} P_i & \\
\mathsf{Ty}^{-} & \ni & N & ::= & \langle r \rangle P \mid rP \multimap N \mid \Pi_{i:I} N_i & \\
\mathsf{Cxt} & \ni & \Gamma & ::= & \emptyset \mid \Gamma.x{:}P & \\
\mathsf{R} & \ni & r, s & ::= & 0 \mid 1 \mid r + s \mid rs & \text{Resource qualifiers} \\
\mathsf{RCxt} & \ni & \gamma, \delta & ::= & \emptyset \mid \gamma.x{:}r & \text{Resource context}
\end{array}$$

Terms.

$$\begin{array}{llllll}
\mathsf{Tm}^{+} & \ni & v, w & ::= & x \mid \mathsf{thunk}\, t \mid \mathsf{in}_i\, v \mid \mathsf{tup}\, \bar{v} & \\
\mathsf{Tm}^{-} & \ni & t, u & ::= & v \,\mathsf{be}\, r\, x.\, t & \\
 & & & \mid & \mathsf{force}\, v \mid v \,\mathsf{cases}\, \{ r\, \overline{x.\, t} \} \mid v \,\mathsf{split}\, r\, \bar{x}.\, t & \\
 & & & \mid & \mathsf{ret}\, v \mid u \,\mathsf{to}\, x.\, t & \\
 & & & \mid & \lambda x.\, t \mid t\, v & \\
 & & & \mid & \mathsf{record}\{\overline{i : t}\} \mid \mathsf{proj}_i\, t &
\end{array}$$

Value typing $\boxed{\gamma\, \Gamma \vdash v : P}$.

$$\textsc{var}\ \frac{}{0\, \Gamma.\, 1\, x{:}P \vdash x : P} \qquad \Box\textsc{-intro}\ \frac{\gamma\Gamma \vdash t : N}{\gamma\Gamma \vdash \mathsf{thunk}\, t : \Box N}$$

$$\Sigma\textsc{-intro}\ \frac{\gamma\Gamma \vdash v : P_i}{\gamma\Gamma \vdash \mathsf{in}_i\, v : \Sigma_I P} \qquad \otimes\textsc{-intro}\ \frac{\forall i{:}I,\ \gamma_i\, \Gamma \vdash v_i : P_i}{(\Sigma_{i:I} \gamma_i)\, \Gamma \vdash \mathsf{tup}\, v : \otimes_I P}$$

Computation typing $\boxed{\gamma\, \Gamma \vdash t : N}$.

$$\textsc{let}\ \frac{\delta\, \Gamma \vdash v : P \qquad \gamma\, \Gamma.\, r\, x{:}P \vdash t : N}{(\gamma + r\delta)\, \Gamma \vdash v \,\mathsf{be}\, r\, x.\, t : N} \qquad \Box\textsc{-elim}\ \frac{\gamma\Gamma \vdash v : \Box N}{\gamma\Gamma \vdash \mathsf{force}\, v : N}$$

$$\Sigma\textsc{-elim}\ \frac{\delta\, \Gamma \vdash v : \Sigma_I P \quad \forall i{:}I,\ \gamma\, \Gamma.\, r\, x_i{:}P_i \vdash t_i : N}{(\gamma + r\delta)\, \Gamma \vdash v \,\mathsf{cases}\, \{ r\, x_i.\, t_i \}_{i:I} : N} \qquad \otimes\textsc{-elim}\ \frac{\delta\, \Gamma \vdash v : \otimes_I P \quad \gamma\, \Gamma.\, \overline{r\, x_i{:}P_i}^{\,i:I} \vdash t : N}{(\gamma + r\delta)\, \Gamma \vdash v \,\mathsf{split}\, r\, \bar{x}.\, t : N}$$

$$\Diamond\textsc{-intro}\ \frac{\gamma\Gamma \vdash v : P}{r\gamma\Gamma \vdash \mathsf{ret}\, v : \langle r \rangle P} \qquad \Diamond\textsc{-elim}\ \frac{\delta\, \Gamma \vdash u : \langle r \rangle P \qquad \gamma\, \Gamma.\, r\, x{:}P \vdash t : N}{(\gamma + \delta)\, \Gamma \vdash u \,\mathsf{to}\, x.\, t : N}$$

$$\multimap\textsc{-intro}\ \frac{\gamma\, \Gamma.\, r\, x{:}P \vdash t : N}{\gamma\, \Gamma \vdash \lambda x.\, t : rP \multimap N} \qquad \multimap\textsc{-elim}\ \frac{\gamma\, \Gamma \vdash t : rP \multimap N \qquad \delta\, \Gamma \vdash v : P}{(\gamma + r\delta)\, \Gamma \vdash t\, v : N}$$

$$\Pi\textsc{-intro}\ \frac{\forall i{:}I,\ \gamma\Gamma \vdash t_i : N_i}{\gamma\Gamma \vdash \mathsf{record}\{i : t_i\}_{i:I} : \Pi_I N} \qquad \Pi\textsc{-elim}\ \frac{\gamma\Gamma \vdash t : \Pi_I N}{\gamma\Gamma \vdash \mathsf{proj}_i\, t : N_i} \qquad \textsc{weak}\ \frac{\gamma'\, \Gamma \vdash t : N}{\gamma\, \Gamma \vdash t : N}\ \gamma \leq \gamma'$$

Fig. 2. Coeffect-graded call-by-push-value.

{1} for exactly one use (linear), {0, 1} for at most one use (affine), {0} for no use (constant), and $\mathbb{N}$ for arbitrary use (unrestricted). A resource qualifier could also be a security level (public or private), or a sensitivity level (a non-negative real) [Reed and Pierce 2010]. If a lambda abstraction $\lambda x.\, t$ is typed with $rP \multimap N$, qualifier r is attached to variable x in the resource context (rule $\multimap$-INTRO). If a function $t : rP \multimap N$ is applied (rule $\multimap$-ELIM), we need an r-qualified argument $v : rP$. We could have given a qualified value typing judgement $\gamma\Gamma \vdash v : rP$ in the style of McBride [2016] that—in the quantitative interpretation—provides r copies of v to be consumed by the function. Such a judgement could come with a scaling rule

$$\frac{\gamma\Gamma \vdash v : sP}{r\gamma\Gamma \vdash v : (rs)P}$$

that allows to scale the production of v by r if the resources are scaled accordingly (from γ to $r\gamma$). However, this would have prevented the typing of force v (rule $\Box$-ELIM), bcause there is no place for the scaling factor in computation typing. Semantically, the v in force v lives in $\mathsf{D}_{\_}[\![N]\!]^-$, and we can only extract the computation in $[\![N]\!]^-$ if we can instantiate the coeffect qualifier _ to 1. Such is not possible if scaling already happened (and needs to be respected). Instead, we bake scaling into the transition from values to computations. Thus, $\multimap$-ELIM receives an argument $\delta\Gamma \vdash v : P$, and to satisfy the demands of the function $\gamma\Gamma \vdash t : rP \multimap N$, the resources δ for the argument are scaled by r, summing the resource requirements for the application to $\gamma + r\delta$. Analogous scaling of the eliminatee is baked into the other value eliminators (Σ-ELIM, $\otimes$-ELIM), into LET, and into $\diamond$-INTRO.

The monadic type $\langle r\rangle P$ records the multiplicity r of the value of type P resulting from the computation. This construction is dual to the comonadic type $[e]N$ from effect-graded CBPV (Section 2.4). Since negative types are not D-coalgebras and do not support scaling, the scaling in $\diamond$-INTRO is the last opportunity for scaling before entering the monad. Typing of bind (u to $x.\, t$) attaches the resource qualifier r stored in $\langle r\rangle P$ to the variable x ($\diamond$-ELIM).

3.5 Coeffect-graded CBPV: denotational semantics

Positive types P are interpreted as D-coalgebras $[\![P]\!]^+ : [(\mathsf{R}, \leq) \to C]$ and negative types N as objects $[\![N]\!]^- : C$. The interpretation of contexts is $[\![\Gamma]\!] = \boxtimes_{x:\mathrm{dom}\,\Gamma} [\![\Gamma(x)]\!]^+$.

$$\begin{array}{lcl} [\![\_]\!]^+ & : & \mathsf{Ty}^+ \to [(\mathsf{R}, \leq) \to C] \\ [\![\Box N]\!]^+_r & = & \mathsf{D}_r\, [\![N]\!]^- \\ [\![\Sigma_I P]\!]^+_r & = & \coprod_{i:I} [\![P_i]\!]^+_r \\ [\![\otimes_I P]\!]^+_r & = & \otimes_{i:I} [\![P_i]\!]^+_r \end{array} \qquad \begin{array}{lcl} [\![\_]\!]^-\_ & : & \mathsf{Ty}^- \to C \\ [\![\langle r\rangle P]\!]^- & = & \mathsf{T}[\![P]\!]^+_r \\ [\![rP \multimap N]\!]^- & = & [\![P]\!]^+_r \multimap [\![N]\!]^- \\ [\![\Pi_I N]\!]^- & = & \prod_{i:I} [\![N_i]\!]^- \end{array}$$

The symmetric monoidal category C needs to be equipped with cartesian products $\Pi_I B$ as well as with distributive coproducts $\coprod_I A$ with distribution morphism $\mathrm{dist} : C\,(C \otimes \coprod_I A,\ \coprod_{i:I} (C \otimes A_i))$.

Computations $\gamma\Gamma \vdash t : N$ are interpreted as morphisms $(\!|t|\!) : C\,([\![\Gamma]\!]_\gamma,\ [\![N]\!]^-)$ and values $\gamma\Gamma \vdash v : P$ as families $(\!|v|\!)_r : C\,([\![\Gamma]\!]_{r\gamma},\ [\![P]\!]^+_r)$ natural in r. Naturality here means that $[\![P]\!]^+_{r\leq s} \circ (\!|v|\!)_r = (\!|v|\!)_s \circ \left([\![\Gamma]\!]_{(\_\cdot\gamma)}\right)_{r\leq s}$. Again, because of WEAK, we interpret typing derivations rather than terms. The interpretation is now rather straightforward, but we spell it out for reference.

$$\begin{array}{lcl} \textsc{var} & (\!|0\Gamma.1x{:}P \vdash x : P|\!)_r & = & \rho \circ (\mathrm{drop}_{[\![\Gamma]\!]} \otimes \mathrm{id}_{[\![P]\!]^+_r}) \\ \Box\textsc{-intro} & (\!|\gamma\Gamma \vdash \mathrm{thunk}\, t : \Box N|\!)_r & = & \mathrm{fmap}_{\mathsf{D}_r} (\!|t|\!) \circ \mathrm{expose}_{[\![\Gamma]\!]} \\ \Sigma\textsc{-intro} & (\!|\gamma\Gamma \vdash \mathrm{in}_i\, v : \Sigma_I P|\!)_r & = & \iota_i \circ (\!|v|\!)_r \\ \otimes\textsc{-intro} & (\!|(\Sigma_I \gamma)\Gamma \vdash \mathrm{tup}^v : \otimes_I P|\!)_r & = & \otimes_{i:I} (\!|v_i|\!)_r \circ \mathrm{duplicate} \end{array}$$

LET	$(\!(\gamma + r\delta)\Gamma \vdash v \,\mathsf{be}\, r\,x.\,t : N)\!)$	$=$	$(\!(t)\!) \circ (\mathrm{id}_{[\![\Gamma]\!]_\gamma} \otimes (\!(v)\!)_r) \circ \mathrm{duplicate}$
□-ELIM	$(\!(\gamma\Gamma \vdash \mathsf{force}\, v : N)\!)$	$=$	$\mathrm{extract}_{\mathsf{D}} \circ (\!(v)\!)_1$
Σ-ELIM	$(\!(\gamma + r\delta)\Gamma \vdash v \,\mathsf{cases}\, \{rx_i.\,t_i\}_{i:I} : N)\!)$	$=$	$[(\!(t_i)\!)]_{i:I} \circ \mathrm{dist} \circ (\mathrm{id}_{[\![\Gamma]\!]_\gamma} \otimes (\!(v)\!)_r) \circ \mathrm{duplicate}$
⊗-ELIM	$(\!(\gamma + r\delta)\Gamma \vdash v \,\mathsf{split}\, r\,\bar{x}.\,t : N)\!)$	$=$	$(\!(t)\!) \circ \mathrm{iso}^{\otimes} \circ (\mathrm{id}_{[\![\Gamma]\!]_\gamma} \otimes (\!(v)\!)_r) \circ \mathrm{duplicate}$
◇-INTRO	$(\!(r\gamma)\Gamma \vdash \mathsf{ret}\, v : \langle r\rangle P)\!)$	$=$	$\mathrm{return} \circ (\!(v)\!)_r$
◇-ELIM	$(\!(\gamma + r\delta)\Gamma \vdash u \,\mathsf{to}\, r\,x.\,t : N)\!)$	$=$	$\mathrm{run}_{[\![N]\!]} \circ \mathrm{fmap}_{\mathsf{T}}(\!(t)\!) \circ \mathrm{strength}^{\mathsf{l}}$ $\circ\, (\mathrm{id}_{[\![\Gamma]\!]_\gamma} \otimes (\!(u)\!)) \circ \mathrm{duplicate}$
⊸-INTRO	$(\!(\gamma\Gamma \vdash \lambda x.\,t : rP \multimap N)\!)$	$=$	$\Lambda(\!(t)\!)$
⊸-ELIM	$(\!(\gamma + r\delta)\Gamma \vdash t\, v : N)\!)$	$=$	$\mathrm{eval} \circ ((\!(t)\!) \otimes (\!(v)\!)_r) \circ \mathrm{duplicate}$
Π-INTRO	$(\!(\gamma\Gamma \vdash \mathsf{record}_I\, t : \Pi_I N)\!)$	$=$	$\langle (\!(t_i)\!) \rangle_{i:I}$
Π-ELIM	$(\!(\gamma\Gamma \vdash \mathsf{proj}_i\, t : N_i)\!)$	$=$	$\pi_i \circ (\!(t)\!)$
WEAK	$(\!(\gamma\Gamma \vdash t : N)\!)$	$=$	$(\!(t)\!) \circ [\![\Gamma]\!]_{\gamma \le \gamma'}$

It is remarkable that coeffect-graded CBPV works *without any distributive law* [Gaboardi et al. 2016] for the monad T and the graded comonad D. This was a crucial design criterion, leading to resource qualifiers in the monadic type $\langle r\rangle P$. A distributive law would be required if we allowed scaling of computations, not just of values.

3.6 Coeffect-graded CBPV: equational theory and operational semantics

The equational theory and operational semantics of coeffect-graded CBPV is identical to the one of CBPV [Levy 2006, Fig. 11]. In this section, we make just a few remarks on an alternative presentation of the permutation laws (called "sequencing laws" in *loc. cit.*).

The equational theory of pure CBPV comprises β and η laws, and *sequencing laws* that syntactically express that negative types are monad algebras [Levy 2006, Fig. 11]. The sequencing laws contain a generalization of the associativity law for monads and allow to permute a bind under a λ or record construction. In the presence of η for functions and records, they are inter-derivable with the following permutation laws:

$$\begin{array}{lll} \pi\text{-}\diamond & (t_1 \,\mathsf{to}\, x_1.\,t_2) \,\mathsf{to}\, x_2.\,t_3 & = \quad t_1 \,\mathsf{to}\, x_1.\,(t_2 \,\mathsf{to}\, x_2.\,t_3) \\ \pi\text{-}\multimap & (u \,\mathsf{to}\, x.\,t)\, v & = \quad u \,\mathsf{to}\, x.\,t\, v \\ \pi\text{-}\Pi & \mathsf{proj}_i\,(u \,\mathsf{to}\, x.\,t) & = \quad u \,\mathsf{to}\, x.\,\mathsf{proj}_i\, t \end{array}$$

Note that permutations for let-bindings like $(v \,\mathsf{be}\, r\,x.\,t)\,w = v \,\mathsf{be}\, r\,x.\,t\,w$ are instances of the β-law $v \,\mathsf{be}\, r\,x.\,t = t[v/rx]$. Herein, $\boxed{t[v/rx]}$ shall the denote the (capture-avoiding) replacement $t[v/x]$ of an r-annotated positive variable x by value v in term t. Permutations for value-eliminations like $(v \,\mathsf{split}\, r\,\bar{x}.\,t)\,w = v \,\mathsf{split}\, r\,\bar{x}.\,t\,w$ are derivable (using the β-laws) from the η-laws, like the η-⊗-law $t'[v/rz] = v \,\mathsf{split}\, r\,\bar{x}.\,t'[\mathsf{tup}\,\bar{x}/rz]$ $(\mathsf{let}\, t' = (z \,\mathsf{split}\, r\,\bar{x}.\,t)\,w)$.

3.7 Coeffect-graded CBPV: substitution and metatheory

Substitution for coeffect-graded type system has been worked out in detail by Atkey and Wood [2019]; Wood and Atkey [2020]. It straightforwardly extends to coeffect-graded CBPV. A substitution σ is a finite map from variable to terms, mapping positive variables x to values v. Substitution typing $\boxed{\Psi\Gamma \vdash \sigma : \Delta}$ is equipped with a matrix $\Psi : \mathsf{R}^{\mathrm{dom}\,\Gamma \times \mathrm{dom}\,\Delta}$ recording the usage vector $\Psi(z) : \mathsf{R}^{\mathrm{dom}\,\Gamma}$ for each variable $z \in \mathrm{dom}\,\Delta = \mathrm{dom}\,\sigma$ in the domain of the substitution.

$$\frac{}{0\Gamma \vdash \emptyset : \emptyset} \qquad \frac{\Psi\Gamma \vdash \sigma : \Delta \qquad \gamma\Gamma \vdash v : P}{(\Psi, \gamma)\Gamma \vdash (\sigma, v/x) : (\Delta.x{:}P)}$$

Capture-avoiding parallel substitution into values $\boxed{v\sigma}$ and computations $\boxed{t\sigma}$ is defined as usual by recursion on the term.

The matrix $\Psi : \mathsf{R}^{\mathrm{dom}\,\Gamma \times \mathrm{dom}\,\Delta}$ acts as a linear map $(\delta : \mathsf{R}^{\mathrm{dom}\,\Delta}) \mapsto (\delta\Psi : \mathsf{R}^{\mathrm{dom}\,\Gamma})$ and allows us to state the substitution theorem.

Theorem 3.1 (Substitution preserves typing). *Let* $\Psi\Gamma \vdash \sigma : \Delta$.

(1) *If* $\delta\Delta \vdash v : P$ *then* $(\delta\Psi)\Gamma \vdash v\sigma : P$.
(2) *If* $\delta\Delta \vdash t : N$ *then* $(\delta\Psi)\Gamma \vdash t\sigma : N$.

Proof. Straightforward adaptation of Atkey and Wood [2019]; Wood and Atkey [2020]. □

Corollary 3.2 (Single substitution preserves typing).

(1) *If* $\delta\Gamma \vdash v : P$ *and* $\gamma\Gamma.rx{:}P \vdash w : P'$ *then* $(\gamma + r\delta)\Gamma \vdash w[v/rx] : P'$.
(2) *If* $\delta\Gamma \vdash v : P$ *and* $\gamma\Gamma.rx{:}P \vdash t : N$ *then* $(\gamma + r\delta)\Gamma \vdash t[v/rx] : N$.

Corollary 3.3 (Subject reduction). *β-reduction of coeffect-graded CBPV preserves types.*

In the pure (effect-free) version, every computation at value type returns a value.

Conjecture 3.4 (Canonicity of pure coeffect-graded CBPV). *If* $\vdash t : \diamond P$ *then* $t =_\beta \mathrm{ret}\, v$ *for some* $\vdash v : P$.

This normalization result can be proved with standard techniques such as reducibility candidates or normalization by evaluation (cf. Abel and Sattler [2019]).

The denotation of a substitution $\Psi\Gamma \vdash \sigma : \Delta$ is a natural transformation $(\!|\sigma|\!)_\delta : C\,(\llbracket\Gamma\rrbracket_{\delta\Psi}, \llbracket\Delta\rrbracket_\delta)$ such that $\llbracket\Delta\rrbracket_{\delta \le \delta'} \circ (\!|\sigma|\!)_\delta = (\!|\sigma|\!)_{\delta'} \circ \llbracket\Gamma\rrbracket_{\delta\Psi \le \delta'\Psi}$. It is defined by recursion as follows:

$$\frac{}{0\Gamma \vdash \emptyset : \emptyset} \qquad \begin{aligned}(\!|\emptyset|\!)_\delta \quad &: C\,(\llbracket\Gamma\rrbracket_0, \mathsf{I}) \\ &= \mathsf{drop}_{\llbracket\Gamma\rrbracket}\end{aligned}$$

$$\frac{\Psi\Gamma \vdash \sigma : \Delta \qquad \gamma\Gamma \vdash v : P}{(\Psi,\gamma)\Gamma \vdash (\sigma, v/x) : (\Delta.x{:}P)} \qquad \begin{aligned}(\!|\sigma, v/x|\!)_{(\delta,r)} \quad &: C\,(\llbracket\Gamma\rrbracket_{(\delta,r)(\Psi,\gamma)}, \llbracket\Delta\rrbracket_\delta \otimes \llbracket P\rrbracket^+_r) \\ &= ((\!|\sigma|\!)_\delta \otimes (\!|v|\!)_r) \circ \mathsf{duplicate}_{\llbracket\Gamma\rrbracket}\end{aligned}$$

Observe that $(\delta, r)(\Psi, \gamma) = \delta\Psi + r\gamma$ and, consequently, $\mathsf{duplicate}_{\llbracket\Gamma\rrbracket} : C\,(\llbracket\Gamma\rrbracket_{(\delta,r)(\Psi,\gamma)}, \llbracket\Gamma\rrbracket_{\delta\Psi} \otimes \llbracket\Gamma\rrbracket_{r\gamma})$.

Theorem 3.5 (Soundness of substitution). *Let* $\Psi\Gamma \vdash \sigma : \Delta$.

(1) *If* $\delta\Delta \vdash v : P$ *then* $(\!|v\sigma|\!)_r = (\!|v|\!)_r \circ (\!|\sigma|\!)_{r\delta} : C\,(\llbracket\Gamma\rrbracket_{r\delta\Psi}, \llbracket P\rrbracket^+_r)$.
(2) *If* $\delta\Delta \vdash t : N$ *then* $(\!|t\sigma|\!) = (\!|t|\!) \circ (\!|\sigma|\!)_\delta : C\,(\llbracket\Gamma\rrbracket_{\delta\Psi}, \llbracket N\rrbracket^-)$.

Proof. By induction on the typing derivation of the term v and t, resp. □

Corollary 3.6 (Soundness of single substitution).

(1) *If* $\delta\Gamma \vdash v : P$ *and* $\gamma\Gamma.rx{:}P \vdash w : P'$ *then*

$$(\!|w[v/rx]|\!)_s = (\!|w|\!)_s \circ (\mathsf{id}_{\llbracket\Gamma\rrbracket_{s\gamma}} \otimes (\!|v|\!)_{sr}) \circ \mathsf{duplicate}_{\llbracket\Gamma\rrbracket}.$$

(2) *If* $\delta\Gamma \vdash v : P$ *and* $\gamma\Gamma.rx{:}P \vdash t : N$ *then*

$$(\!|t[v/rx]|\!) = (\!|t|\!) \circ (\mathsf{id}_{\llbracket\Gamma\rrbracket_\gamma} \otimes (\!|v|\!)_r) \circ \mathsf{duplicate}_{\llbracket\Gamma\rrbracket}.$$

Proof. After comprehending that the semantics of identity substitutions is just id, we observe that the semantics of single substitutions is $(\!|v/rx|\!) = (\mathsf{id} \otimes (\!|v|\!)_r) \circ \mathsf{duplicate}$. □

Types.

$$\begin{array}{rcrcl} \mathsf{Ty}^+ & \ni & P & ::= & [\ e\ ]N \mid o \mid \Sigma_{i:I} P_i \mid \otimes_{i:I} P_i \\ \mathsf{Ty}^- & \ni & N & ::= & \langle r \rangle P \mid rP \multimap N \mid \Pi_{i:I} N_i \end{array}$$

Value typing $\boxed{\gamma\Gamma \vdash v : P}$.

$$\textsc{var}\ \frac{}{0\Gamma.1x{:}P \vdash x : P} \qquad \Box\textsc{-intro}\ \frac{\gamma\Gamma \vdash t : N \mid e}{\gamma\Gamma \vdash \mathsf{thunk}\, t : [\ e\ ]N}$$

$$\Sigma\textsc{-intro}\ \frac{\gamma\Gamma \vdash v : P_i}{\gamma\Gamma \vdash \mathsf{in}_i\, v : \Sigma_I P} \qquad \otimes\textsc{-intro}\ \frac{\forall i{:}I,\ \gamma_i\Gamma \vdash v_i : P_i}{(\Sigma_{i:I}\gamma_i)\Gamma \vdash \mathsf{tup}\, v : \otimes_I P}$$

Computation typing $\boxed{\gamma\Gamma \vdash t : N \mid e}$.

$$\textsc{let}\ \frac{\delta\Gamma \vdash v : P \qquad \gamma\Gamma.rx{:}P \vdash t : N \mid e}{(\gamma + r\delta)\Gamma \vdash v\, \mathsf{be}\, r\, x.\, t : N \mid e} \qquad \Box\textsc{-elim}\ \frac{\gamma\Gamma \vdash v : [\ e\ ]N}{\gamma\Gamma \vdash \mathsf{force}\, v : N \mid e}$$

$$\Sigma\textsc{-elim}\ \frac{\delta\Gamma \vdash v : \Sigma_I P \quad \forall i{:}I,\ \gamma\Gamma.rx_i{:}P_i \vdash t_i : N \mid e}{(\gamma + r\delta)\Gamma \vdash v\, \mathsf{cases}\, \{rx_i.\, t_i\}_{i:I} : N \mid e} \qquad \otimes\textsc{-elim}\ \frac{\delta\Gamma \vdash v : \otimes_I P \quad \gamma\Gamma.\overline{rx_i{:}P_i}^{\,i:I} \vdash t : N \mid e}{(\gamma + r\delta)\Gamma \vdash v\, \mathsf{split}\, r\, \bar{x}.\, t : N \mid e}$$

$$\Diamond\textsc{-intro}\ \frac{\gamma\Gamma \vdash v : P}{r\gamma\Gamma \vdash \mathsf{ret}\, v : \langle r \rangle P \mid \varepsilon} \qquad \Diamond\textsc{-elim}\ \frac{\delta\Gamma \vdash u : \langle r \rangle P \mid e_1 \qquad \gamma\Gamma.rx{:}P \vdash t : N \mid e_2}{(\gamma + \delta)\Gamma \vdash u\, \mathsf{to}\, x.\, t : N \mid e_1 \bullet e_2}$$

$$\multimap\textsc{-intro}\ \frac{\gamma\Gamma.rx{:}P \vdash t : N \mid e}{\gamma\Gamma \vdash \lambda x.\, t : rP \multimap N \mid e} \qquad \multimap\textsc{-elim}\ \frac{\gamma\Gamma \vdash t : rP \multimap N \mid e \qquad \delta\Gamma \vdash v : P}{(\gamma + r\delta)\Gamma \vdash t\, v : N \mid e}$$

$$\Pi\textsc{-intro}\ \frac{\forall i{:}I,\ \gamma\Gamma \vdash t_i : N_i \mid e}{\gamma\Gamma \vdash \mathsf{record}\{i : t_i\}_{i:I} : \Pi_I N \mid e} \qquad \Pi\textsc{-elim}\ \frac{\gamma\Gamma \vdash t : \Pi_I N \mid e}{\gamma\Gamma \vdash \mathsf{proj}_i\, t : N_i \mid e}$$

$$\textsc{sub}\ \frac{\gamma'\Gamma \vdash t : N \mid e}{\gamma\Gamma \vdash t : N \mid e'}\ \gamma \le \gamma',\ e \le e'$$

Fig. 3. Fully graded call-by-push-value.

4 FULLY GRADED CBPV

In this section we finally present a *fully graded* version of CBPV where both effects and coeffects are graded. Fig. 3 presents its types and typing rules where the differences to the coeffect-graded version are highlighted.

We reintroduce effect grading into the computation typing $\gamma\Gamma \vdash t : N \mid e$. As in the effect-graded version, effects are accumulated in a single place, the monadic bind construct ($\Diamond$-elim).

The crucial insight concerning the absence of distributive laws is the laziness of effects in CBPV. They are only observed at positive types [Levy 2006], thus, can be "bottled up" in $\mathsf{thunk}\, t : [e]N$ and reactivated via forcing in $\mathsf{force}\, v : N \mid e$.

4.1 Graded CBPV: semantics

As in effect-graded CBPV, negative types N are interpreted as graded monad algebras $[\![N]\!]^- : [(\mathsf{Eff}, \le) \to C]$. Dually, as in coeffect-graded CBPV, positive types are interpreted as graded comonad

coalgebras $[\![P]\!]^+ : [(\mathsf{R}, \leq) \to C]$. Values $\gamma\Gamma \vdash v : P$ are interpreted as natural transformations $(\!|v|\!) : [\![\Gamma]\!]_{(\_\cdot\gamma)} \dot{\to} [\![P]\!]^+$ and computations $\gamma\Gamma \vdash t : N \mid e$ as morphisms $(\!|t|\!) : C\,([\![\Gamma]\!]_\gamma, [\![N]\!]^-_e)$. Except for some change in typing, the denotation of terms is unchanged from coeffect-graded CBPV. We recapitulate the most interesting cases here with the updated typing:

$$(\!|\mathsf{thunk}\ t|\!)_r \qquad [\![\Gamma]\!]_{r\gamma} \xrightarrow{\text{expose}} \mathsf{D}_r\,[\![\Gamma]\!]_\gamma \xrightarrow{\mathsf{fmap}_{\mathsf{D}_r}(\!|t|\!)} \mathsf{D}_r\,[\![N]\!]^-_e$$

$$(\!|\mathsf{force}\ v|\!) \qquad [\![\Gamma]\!]_\gamma \xrightarrow{(\!|v|\!)_1} \mathsf{D}_1\,[\![N]\!]^-_e \xrightarrow{\text{extract}} [\![N]\!]^-_e$$

$$(\!|\mathsf{ret}\ v|\!) \qquad [\![\Gamma]\!]_{r\gamma} \xrightarrow{(\!|v|\!)_r} [\![P]\!]^+_r \xrightarrow{\text{return}} \mathsf{T}_\varepsilon\,[\![P]\!]^+_r$$

$$(\!|u\ \mathsf{to}\ x.\ t|\!) \qquad [\![\Gamma]\!]_{\gamma+\delta} \xrightarrow{\text{duplicate}} [\![\Gamma]\!]_\gamma \otimes [\![\Gamma]\!]_\delta \xrightarrow{\mathsf{id}\otimes(\!|u|\!)} [\![\Gamma]\!]_\gamma \otimes \mathsf{T}_{e_1}[\![P]\!]^+_r \downarrow \mathsf{strength}^{\mathsf{l}}$$
$$[\![N]\!]^-_{e_1 \bullet e_2} \xleftarrow{\mathsf{run}_{[\![N]\!]^-}} \mathsf{T}_{e_1}\,[\![N]\!]^-_{e_2} \xleftarrow{\mathsf{fmap}_{\mathsf{T}_{e_1}}(\!|t|\!)} \mathsf{T}_{e_1}([\![\Gamma]\!]_\gamma \otimes [\![P]\!]^+_r)$$

Subsumption (SUB) is interpreted by pre- and post-composition with the functorial actions of $[\![\Gamma]\!]$ and $[\![N]\!]^-$:

$$[\![\Gamma]\!]_\gamma \xrightarrow{[\![\Gamma]\!]_{\gamma\leq\gamma'}} [\![\Gamma]\!]_{\gamma'} \xrightarrow{(\!|t|\!)} [\![N]\!]^-_e \xrightarrow{[\![N]\!]^-_{e\leq e'}} [\![N]\!]^-_{e'}$$

The operational semantics of graded CBPV is identical to the one of CBPV, and the substitution typing of coeffect-graded CBPV (Section 3.7) can be mechanically transferred to fully graded CBPV.

4.2 Discussion

Graded CBPV provides syntax to work with graded monads T_e and comonads D_r. *A priori*, one could have expected that these constructs are directly reflected in the syntax of types, as $\langle e\rangle P$ and $[r]N$. Surprisingly, the adaptation of graded effect and coeffect typing to CBPV places the qualifiers in the opposite way, as $\langle r\rangle P$ and $[e]N$. Given our semantics of types as (co)monad (co)algebras, this apparent oddity has a natural explanation: the qualifiers do not instantiate the grade in the monad or comonad, but in the respective (co)algebra. The alternative concrete syntax $\diamond P_r$ and $\Box N_e$ would maybe transport the intended semantics, $\mathsf{T}_-[\![P]\!]^+_r$ and $\mathsf{D}_-[\![N]\!]^-_e$, more directly.

How does Graded CBPV work as a programming language? A program is usually a set of definitions and then an entrypoint in form of an expression or simply the name of the main procedure. In Graded CBPV, the definitions are given as a sequence of let-bindings whose final body serves as the entrypoint. The entrypoint should be an expression of type $\langle 1\rangle P$, producing an observable value and thus, the effects leading up to this result. The expressions bound by the lets will often be functions (or records), but wrapped in thunks $[e]N$ to satisfy the formal requirement to be of positive type. This thunk is where the effects are declared that a function produces. Thus, in the big picture, the effects of a computation *are* stated in the types, not just in the typing judgement $\gamma\Gamma \vdash t : N \mid e$, even though the effect annotation is in a *a priori* unexpected location.

5 RELATED WORK

Our style of effect tracking and its interpretation via graded monads is based on Wadler and Thiemann [2003] and Katsumata [2014].

5.1 Effect tracking in CBPV

McDermott and Mycroft [2019] observe that CBPV can represent call-by-name and call-by-value evaluation strategies but not call-by-need. As a remedy, they extend CBPV to ECBPV by variables $\underline{x} : \diamond P$ that hold monadic values, plus a corresponding let-binding construct. They further present an effect-graded version of ECBPV, drawing effect qualifiers from a preordered monoid. In contrast to our version, effect annotations f are stored at monadic types $\langle f \rangle P$. Instead of our judgement $\Gamma \vdash t : N \mid e$ (that accumulates effect traces e) and a semantic interpretation of computation types N as graded monad algebras, they define a syntactic operation $\langle f \rangle N$ that pushes down effect qualifier f to the monadic types $\langle f' \rangle P$ in N where the effects are combined to $\langle f \bullet f' \rangle P$. This way of handling effect grading seems equivalent to our approach, albeit it is syntactic rather than semantic.

In previous work, McDermott and Mycroft [2018] use coeffect typing to track variable usage in call-by-need simply-typed lambda calculus with conditionals. Compared to the usual resource semiring R tailored to call-by-name usage analysis, their coeffect algebra is more expressive: it can produce traces of variable usage, including usage of bound variables. They show how graded effect tracking can be simulated by the coeffect typing, *e.g.*, for the effect of non-determinism. Naturally, they focus on operational semantics.

To accomodate several sorts of effects in one language (CBPV hosts only a single effect type), Kammar and Plotkin [2012] present MAIL, the Multi-Adjunctive Intermediate Language, which is a version of CBPV *parameterized* by effect qualifiers. In contrast to our effect-graded version of CBPV, effect qualifiers are a parameter to the type system as a whole, basically a *mode* attached to the typing judgement. Via effect subsumption, one can switch to a "wider" mode. However, as effects qualifiers are drawn from a preordered set rather than a monoid, more detailed static effect traces cannot be captured. In their own words:

> Instead of one kind of computation, for each effectset $\varepsilon \in E$, we have ε-computations $\mathrm{Comp}(\varepsilon)$ that can cause effects in ε. We view MAIL as multiple copies of CBPV, one for each ε, sharing the same values. One can translate between these different CBPVs by means of coercion [...]

5.2 Coeffect typing

Our style of coeffect typing is heavily influenced by McBride [2016] and Atkey [2018] who keep the quantification of variable *usage* in usage context γ separate from the variable *declaration* in an ordinary typing context Γ. This division of labor has removed two major obstacle in the integration of linear and dependent types: the customary sorting of linear and unrestricted variables into separate typing contexts, and the dominance of context concatenation operations in the typing of the multiplicative connectives of linear logic, *e.g.*:

$$\frac{\Delta_1 \vdash v_1 : A_1 \qquad \Delta_2 \vdash v_2 : A_2}{\Delta_1.\Delta_2 \vdash (v_1, v_2) : A_1 \otimes A_2}$$

In contrast to our style, the coeffect type systems of Reed and Pierce [2010], Petricek et al. [2014], Brunel et al. [2014], Ghica and Smith [2014], and Orchard et al. [2019] keep context concatenation. This does no harm in simply-typed settings, but seems to duplicate a mechanism already present in form of the resource qualifier 0.

Our treatment of substitution in coeffect-graded CBPV is based on Wood and Atkey [2020], who formalize (in Agda) substitution for intuitionistic linear logic with a subexponential representing a graded comonad. They observe that instead of a preordered semiring, a left skew semiring can be used as resource algebra. In a skew semiring, the laws involving multiplication only hold as inequalities, not equalities.

5.3 Interaction of effects and coeffects

Gaboardi et al. [2016] investigate systematically the possible interactions between semiring-graded coeffects and monoid-graded effects, in form of distributive laws between the graded comonad and monad. We do not rely on such distributive laws in Graded CBPV. The strict separation of value and computation types and the placement of the monad and comonad at the transition points makes interaction optional. How to integrate distributive laws into Graded CBPV is left for future research.

6 CONCLUSIONS

In this article, we have demonstrated that CBPV can accommodate graded effects and coeffects in a smooth way, keeping the term grammar virtually unchanged and only adding effect and coeffect annotations at the transition points between value and computation types. The surprising simplicity of our solution speaks for the design quality of Levy's CBPV calculus [2006]. For instance, the analogy to intuitionistic linear logic—that lies at the heart of coeffect typing—is already implicitly present in the separation of value ($\otimes$) and computation type products (Π).

For the semantics of Graded CBPV, we have evolved the concepts of monad algebra and comonad coalgebra to their graded versions.

In future work, we would like to investigate whether the distributive laws of Gaboardi et al. [2016] carry over to Graded CBPV, *e.g.*, whether and how a graded comonad can be distributed over a graded monad algebra or, in the symmetric case, a graded monad distributes over a graded comonad coalgebra.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proc. of the 26th ACM Symp. on Principles of Programming Languages, POPL'99*, Andrew W. Appel and Alex Aiken (Eds.). ACM Press, 147–160. https://doi.org/10.1145/292540.292555

Andreas Abel and Christian Sattler. 2019. Normalization by Evaluation for Call-By-Push-Value and Polarized Lambda Calculus. In *Proc. of the 21st Int. Conf. on Principles and Practice of Declarative Programming, PPDP 2019*. 3:1–3:12. https://doi.org/10.1145/3354166.3354168

Robert Atkey. 2018. The Syntax and Semantics of Quantitative Type Theory. In *33rd ACM/IEEE Symp. on Logic in Computer Science (LICS'18)*, Anuj Dawar and Erich Grädel (Eds.). ACM Press, 56–65. https://doi.org/10.1145/3209108.3209189

Robert Atkey and James Wood. 2019. Linear metatheory via linear algebra. In *25th Int. Conf. on Types for Proofs and Programs, TYPES 2019, Abstracts*, Marc Bezem (Ed.). http://www.ii.uib.no/~bezem/program.pdf

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus, See [Shao 2014], 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. In *Proc. of the 21st ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM Press, 476–489. https://doi.org/10.1145/2951913.2951939

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring, See [Shao 2014], 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

Suresh Jagannathan and Peter Sewell (Eds.). 2014. *Proc. of the 41st ACM Symp. on Principles of Programming Languages, POPL 2014*. ACM Press. http://dl.acm.org/citation.cfm?id=2535838

Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic foundations for effect-dependent optimisations. In *Proc. of the 39th ACM Symp. on Principles of Programming Languages, POPL 2012*, John Field and Michael Hicks (Eds.). ACM Press, 349–360. https://doi.org/10.1145/2103656.2103698

Shin-ya Katsumata. 2014. Parametric effect monads and semantics of effect systems, See [Jagannathan and Sewell 2014], 633–646. https://doi.org/10.1145/2535838.2535846

Paul Blain Levy. 2006. Call-by-push-value: Decomposing call-by-value and call-by-name. *J. Higher-Order and Symb. Comput.* 19, 4 (2006), 377–414. https://doi.org/10.1007/s10990-006-0480-6

Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World – Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lect. Notes in Comput. Sci.)*, Sam Lindley, Conor McBride, Philip W. Trinder,

and Donald Sannella (Eds.), Vol. 9600. Springer, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Dylan McDermott and Alan Mycroft. 2018. Call-by-need effects via coeffects. *Open Comput. Sci.* 8, 1 (2018), 93–108. https://doi.org/10.1515/comp-2018-0009

Dylan McDermott and Alan Mycroft. 2019. Extended Call-by-Push-Value: Reasoning About Effectful Programs and Evaluation Order. In *Proc. of the 28th European Symp. on Programming, ESOP 2019 (Lect. Notes in Comput. Sci.)*, Luís Caires (Ed.), Vol. 11423. Springer, 235–262. https://doi.org/10.1007/978-3-030-17184-1_9

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design, Recent Insight and Advances (Lect. Notes in Comput. Sci.)*, Ernst-Rüdiger Olderog and Bernhard Steffen (Eds.), Vol. 1710. Springer, 114–136. https://doi.org/10.1007/3-540-48092-7_6

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. of the ACM on Program. Lang.* 3, ICFP (2019), 110:1–110:30. https://doi.org/10.1145/3341714

Dominic Orchard, Philip Wadler, and Harley Eades III. 2020. Unifying graded and parameterised monads. In *Proc. 8th Wksh. on Mathematically Structured Functional Programming, MSFP 2020 (Electr. Proc. in Theor. Comp. Sci.)*, Max S. New and Sam Lindley (Eds.), Vol. 317. 18–38. https://doi.org/10.4204/EPTCS.317.2

Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. 2014. Coeffects: a calculus of context-dependent computation. In *Proc. of the 19th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM Press, 123–135. https://doi.org/10.1145/2628136.2628160

Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *Proc. of the 15th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM Press, 157–168. https://doi.org/10.1145/1863543.1863568

Ilya Sergey, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Modular, higher-order cardinality analysis in theory and practice, See [Jagannathan and Sewell 2014], 335–348. https://doi.org/10.1145/2535838.2535861

Zhong Shao (Ed.). 2014. *Proc. of the 23rd European Symp. on Programming, ESOP 2014.* Lect. Notes in Comput. Sci., Vol. 8410. Springer. https://doi.org/10.1007/978-3-642-54833-8

Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Sec.* 4, 2/3 (1996), 167–188. https://doi.org/10.3233/JCS-1996-42-304

Philip Wadler and Peter Thiemann. 2003. The marriage of effects and monads. *ACM Trans. Comput. Logic* 4, 1 (2003), 1–32. https://doi.org/10.1145/601775.601776

James Wood and Robert Atkey. 2020. A Linear Algebra Approach to Linear Metatheory. In *Linearity-TLLA 2020.* https://bentnib.org/linear-metatheory.html

Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching.* Ph.D. Dissertation. Carnegie Mellon University. http://software.imdea.org/~noam.zeilberger/thesis.pdf