Grading call-by-push-value, explicitly and implicitly

Dylan McDermott
University of Oxford, UK

Abstract

We present call-by-push-value with effects (CBPVE), a refinement of Levy's call-by-push-value (CBPV) calculus in which the types contain behavioural information about the effects of computations. CBPVE fits well into the existing literature on graded types and computational effects. We demonstrate this by providing graded call-by-value and call-by-name translations into CBPVE, and a semantics based on algebras of a graded monad.

CBPVE is designed as a standalone calculus, with explicit grade information in the syntax. We use it to study the assignment of graded types to the terms of an ungraded calculus such as CBPV, essentially treating the grades as implicit. To interpret such terms in a model that accounts for the grades, one has to prove a coherence result for the implicit grades. We show that, in the case of a graded monadic semantics, the necessary coherence result is false in general. To solve this problem, we show that a mild condition on the algebra of grades is enough to guarantee coherence, giving the first proof of a coherence result for grading, and hence also the first graded monadic semantics for CBPV computations.

2012 ACM Subject Classification Theory of computation → Categorical semantics; Theory of computation → Type structures

Keywords and phrases computational effect, effect system, call-by-push-value, graded monad

Digital Object Identifier 10.4230/LIPIcs.FSCD.2025.25

Funding This work was supported by EU Horizon Europe project TaRDIS (101093006).

Acknowledgements I am grateful to the anonymous referees for helpful comments.

1 Introduction

In the context of programming languages with computational effects, the following two developments have received a large amount of interest.

- *Graded type systems* are a general paradigm for tracking computational effects. Whenever some part of a program can have some observable effect, such as printing a value, or mutating some state, a graded type system would assign a *grade* to that part. The grade would record some information about the effects, e.g. what kind of value may be printed, or which part of the state may be mutated. This information can be useful for instance in verifying program transformations (e.g. [31, 3, 12]).
- Levy's *call-by-push-value* calculus [18] was developed to subsume both call-by-value and call-by-name. It has become one of the standard tools for studying computational effects (e.g. [2, 27]).

Despite this, the computational effects literature still does not contain any satisfactory treatment of a graded call-by-push-value calculus. Our primary goal is to provide one.

We introduce a calculus, called *call-by-push-value with effects* (CBPVE), based on call-by-push-value (CBPV), but with a graded type system. The design of CBPVE is based on the insight that in CBPV, one observes effects happening in computations of *returner types*. As a consequence, in CBPVE the grade annotations appear on returner types – and only on returner types. Our intention is for CBPVE to be the basis for studying graded computational effects in a CBPV-like calculus. We demonstrate its suitability for this purpose by showing that CBPVE subsumes graded call-by-value and call-by-name calculi, exactly

as CBPV subsumes the ungraded calculi. We also show that the monadic semantics for CBPV can be adapted into a graded monadic semantics for CBPVE; *graded monads* being the standard tool for modelling graded computational effects, following Katsumata [14].

Explicit and implicit When designing a graded type system, one has to make a choice. One can either treat (i) the grades as intrinsic to the calculus, so that the grades are baked into the syntax, and there is no separate ungraded calculus, or (ii) start with an ungraded calculus (like CBPV), and treat the ability to assign a particular grade to a computation as merely a property of that computation. In both cases, one has a *subgrading* rule (analogous to subtyping), and the most obvious difference between the two approaches lies in this rule. In the intrinsic approach, terms typically include explicit subgrading coercions, while in the extrinsic approach, they are necessarily implicit. Both options can be found in the literature, for instance, [12] has explicit coercions, while [9, 32, 16] have implicit coercions.

It is usual to think of grades as being purely descriptive, so that in particular they cannot influence the behaviour of a computation in any way; this view fits better into the extrinsic approach. However, the intrinsic approach is more convenient for the theory. In particular, if a calculus has implicit coercions, then to interpret it in a model that accounts for the grades, one should prove a *coherence* result, stating roughly that the interpretation of a computation cannot depend on the grades we assign. This is not necessary in the extrinsic approach. This problem of coherence has largely been neglected for graded type systems. In fact, as we show in Section 6 below, the coherence result is in general *false* for graded monads.

The second goal of this paper is to study the difference between the two approaches, and especially the coherence problem. We design CBPVE following the intrinsic approach: grades are baked into the calculus, and subgrading is explicit. We then show (Section 6) how to use CBPVE to overlay grades onto CBPV, providing a graded type system for CBPV. Since we do not have coherence in general, we cannot always interpret the latter using a graded monad. But we do not give up on our view of grades as purely descriptive: we exhibit a mild condition (Definition 11) on the grades that turns out to be strong enough to prove coherence for any graded monad.

Existing approaches CBPVE is not the first graded CBPV calculus to appear in the literature. Kammar and Plotkin [12] give a calculus designed to be interpreted using a family of monads (which can be seen as a graded monad). Their approach has been taken up in various other works [11, 32]. It is not entirely satisfactory as we explain in the following paragraph, which is why we develop a new calculus.

In [12, 11, 32], every computation is assigned a grade, unlike in CBPVE, where only computations of returner types are assigned grades. We argue that our approach is the correct one, since it is based on the fact that we observe the effects at returner types. Although the type system of CBPVE is superficially very different to that of [12, 11, 32], both approaches are sound, and they only have one small difference in expressiveness. The difference is that, in [12, 11, 32], one can only form a pair of computations of the same grade. This restriction has no theoretical motivation, and so it is not present in CBPVE. We can pair computations arbitrarily. This difference is minor, but should not be ignored. For instance, one of the characteristic properties of call-by-name is the currying isomorphism $(\tau \times \tau') \to \tau'' \cong \tau \to (\tau' \to \tau'')$. Requiring paired computations to have the same grade breaks this isomorphism. We discuss why in Section 4.2.

Contributions and outline The next section gives some necessary background on CBPV. The novel work is in the subsequent sections.

- Section 3 introduces our new calculus CBPVE, which we argue is the appropriate way of tracking effects in a CBPV-like calculus.

- Section 4 shows that CBPVE subsumes graded call-by-value and call-by-name calculi.
- Section 5 describes the graded monadic semantics of CBPVE.
- Section 6 shows how to use CBPVE to overlay grades on top of CBPV. We show that the desired coherence result is false in general, but that it does hold under a mild condition on the grades. This is the first proof of a coherence result for a graded type system.
- Section 7 discusses some related work, and compares with [12, 11, 32] in more detail.

2 Call-by-push-value (without grades)

We briefly describe the syntax of Levy's [18, 20] call-by-push-value (CBPV). The syntax of CBPV terms is stratified into two kinds: *values* V, W do not reduce, *computations* M, N might reduce (possibly with side-effects). The syntax of types is similarly stratified into *value types* A, B and computation types $\underline{C}, \underline{D}$.

$$\begin{array}{rcl} A, B & ::= & \mathbf{U}\underline{C} \mid \mathbf{1} \mid A_1 \times A_2 \mid \coprod_{i \in I} A_i \\ \underline{C}, \underline{D} & ::= & \mathbf{F}A \mid \prod_{i \in I} \underline{C}_i \mid A \to \underline{C} \\ V, W & ::= & x \mid \mathbf{thunk}\, M \mid () \mid (V_1, V_2) \mid \mathbf{inj}_i\, V \\ M, N & ::= & \mathbf{return}\, V \mid M\ \mathbf{to}\ x.\, N \mid \mathbf{do}\, y \leftarrow \mathsf{op}\, V\ \mathbf{then}\, M \\ & & \mid \lambda\{i.\, M_i\}_{i \in I} \mid i\text{`}M \mid \lambda x : A.\, M \mid V\text{`}M \\ & & \mid \mathbf{force}\, V \mid \mathbf{match}\, V\ \mathbf{with}\, (x_1, x_2)\ \mathbf{in}\, M \mid \mathbf{match}\, V\ \mathbf{with}\, \{\mathbf{inj}_i\, x_i.\, M_i\}_{i \in I} \end{array}$$

The value type $\mathbf{U}\underline{C}$ is the type of *thunks* of computations of type $\underline{C}$. Elements of $\mathbf{U}\underline{C}$ are introduced using **thunk**: the value $\mathbf{thunk}\, M$ is the suspension of the computation term M. The corresponding eliminator is **force**, which is the inverse of **thunk**. Value types also include a unit type $\mathbf{1}$, binary products, and finite sums (we require I to be finite). We write $\mathbf{0}$ for the empty type (the sum type with $I = \emptyset$).

The *returner type* $\mathbf{F}A$ has as elements computations that return elements of the value type A; these computations may have observable effects. Elements of $\mathbf{F}A$ are introduced by **return**; the computation $\mathbf{return}\, V$ immediately returns the value V (with no side-effects). Computations can be sequenced using $M\ \mathbf{to}\ x.\, N$. This first evaluates M (which is required to have returner type), and then evaluates N with x bound to the result of M. Computation types also include finite products; $\lambda\{i.\, M_i\}_{i \in I}$ is the tuple of computations whose ith element is M_i, and $i\text{`}M$ is the $i$th projection of $M$. Finally, we have function types $A \to \underline{C}$. Functions send values to computations, and are computations themselves. Function application is written $V\text{`}M$, where V is the argument and M is the function to apply.

A *ground type* is a value type that does not contain $\mathbf{U}$. We augment Levy's [18] syntax, by adding computations $\mathbf{do}\, y \leftarrow \mathsf{op}\, V\ \mathbf{then}\, M$. This computation performs an effectful operation op, with parameter V, and then continues as M. We assume a fixed set of *operation symbols* op, each equipped with two ground types, called the *parameter type* and the *arity*. We write $\mathsf{op} : A \rightsquigarrow B$ to indicate that op has parameter type A and arity B. For instance, we could have $\mathsf{raise} : \mathbf{1} \rightsquigarrow \mathbf{0}$ for raising an exception, $\mathsf{get} : \mathbf{1} \rightsquigarrow A$ and $\mathsf{put} : A \rightsquigarrow \mathbf{1}$ for interacting with a global state of type A, or $\mathsf{print}_m : \mathbf{1} \rightsquigarrow \mathbf{1}$ for printing a value m.

CBPV has two typing judgments: $\Gamma \vdash V : A$ for values and $\Gamma \vdash M : \underline{C}$ for computations. *Typing contexts* Γ are ordered lists of (variable, value type) pairs; we write $\cdot$ for the empty typing context. Figure 1 gives the typing rules.

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : \underline{C}}{\Gamma \vdash \mathbf{thunk}\, M : \mathbf{U}\underline{C}} \qquad \frac{}{\Gamma \vdash () : \mathbf{1}}$$

$$\frac{\Gamma \vdash V_1 : A_1 \quad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash V : A_i}{\Gamma \vdash \mathbf{inj}_i V : \coprod_{i \in I} A_i}$$

(a) Typing rules for values $\Gamma \vdash V : A$.

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathbf{return}\, V : \mathbf{F}A}$$

$$\frac{\Gamma \vdash M : \mathbf{F}A \qquad \Gamma, x : A \vdash N : \underline{C}}{\Gamma \vdash M \ \mathbf{to}\ x.\, N : \underline{C}}$$

$$\frac{\mathsf{op} \colon A \rightsquigarrow B \qquad \Gamma \vdash V : A \qquad \Gamma, y : B \vdash M : \underline{C}}{\Gamma \vdash \mathbf{do}\, y \leftarrow \mathsf{op}\, V\, \mathbf{then}\, M : \underline{C}}$$

(b) Typing rules for returner types and operations.

$$\frac{\{\Gamma \vdash M_i : \underline{C}_i\}_{i \in I}}{\Gamma \vdash \lambda\{i.\, M_i\}_{i \in I} : \prod_{i \in I} \underline{C}_i} \qquad \frac{\Gamma \vdash M : \prod_{i \in I} \underline{C}_i}{\Gamma \vdash i\text{`}M : \underline{C}_i}$$

$$\frac{\Gamma, x : A \vdash M : \underline{C}}{\Gamma \vdash \lambda x : A.\, M : A \to \underline{C}}$$

$$\frac{\Gamma \vdash V : A \qquad \Gamma \vdash M : A \to \underline{C}}{\Gamma \vdash V\text{`}M : \underline{C}}$$

$$\frac{\Gamma \vdash V : A_1 \times A_2 \quad \Gamma, x_1 : A_1, x_2 : A_2 \vdash M : \underline{C}}{\Gamma \vdash \mathbf{match}\, V\, \mathbf{with}\, (x_1, x_2)\, \mathbf{in}\, M : \underline{C}}$$

$$\frac{\Gamma \vdash V : \coprod_{i \in I} A_i \qquad \{\Gamma, x_i : A_i \vdash M_i : \underline{C}\}_{i \in I}}{\Gamma \vdash \mathbf{match}\, V\, \mathbf{with}\, \{\mathbf{inj}_i\, x_i.\, M_i\}_{i \in I} : \underline{C}}$$

$$\frac{\Gamma \vdash V : \mathbf{U}\underline{C}}{\Gamma \vdash \mathbf{force}\, V : \underline{C}}$$

(c) Typing rules for the remaining computations $\Gamma \vdash M : \underline{C}$.

Figure 1 Typing rules for call-by-push-value, without grades.

3 Grading call-by-push-value, explicitly

Now that we have given the type system of CBPV, we consider how to turn it into a graded type system. This section introduces our new calculus *call-by-push-value with effects* (CBPVE).

First we need to assume a given collection of grades; following [14], we assume they form an *ordered monoid*.

▶ **Definition 1.** *An* ordered monoid $(\mathbb{E}, \leq, \mathbf{1}, \cdot)$ *is a monoid* $(\mathbb{E}, \mathbf{1}, \cdot)$ *equipped with a preorder* $\leq$ *on* $\mathbb{E}$*, such that the multiplication* $\cdot$ *is monotone.*

The *grades* d, e are elements of the set $\mathbb{E}$. The order $\leq$ provides a notion of approximation of grades: $d \leq e$ means e is more restrictive than e. The *multiplication* $\cdot$ represents sequencing: $d \cdot e$ is the grade of running a computation of grade d followed by a computation of grade e. The grade $\mathbf{1}$ is for computations with no effects.

We require each operation to come with a grade $d \in \mathbb{E}$, the grade of performing the operation. Thus we assume a given *signature* in the following sense, similar to Kura's notion of graded signature [17]. Note that the ground types of CBPVE are identical to those of CBPV, since they do not contain any grade information.

▶ **Definition 2.** *A* graded operation signature Σ *is a set of* operation symbols $\mathsf{op} \colon A \rightsquigarrow_e B$*, each equipped with a grade* d*, and two ground types, the* parameter type A *and the* arity B*.*

For instance, given a set Σ of operation symbols, we could take $\mathbb{E}$ to be the powerset of Σ, ordered by inclusion, and with union for the multiplication, and then assign to each operation op the grade $\{\mathsf{op}\}$. The result would be a Gifford-style type-and-effect system [23] that tracks which operations a computation may use as it runs. Alternatively, we could take natural numbers with addition and the usual ordering, and $\mathsf{print}_m \colon \mathbf{1} \rightsquigarrow_1 \mathbf{1}$. Then "having grade e" would mean printing at most e many values; each execution of print contributing 1 to the grade.

$$\frac{\Gamma \vdash^{\mathtt{g}} V : A}{\Gamma \vdash^{\mathtt{g}} \mathbf{return}\, V : \mathbf{F}_{\mathbf{1}} A} \qquad \frac{\Gamma \vdash^{\mathtt{g}} M : \mathbf{F}_d A \qquad \Gamma, x : A \vdash^{\mathtt{g}} N : \underline{C}}{\Gamma \vdash^{\mathtt{g}} M\ \mathbf{to}\ x.\, N : \langle\!\langle d \rangle\!\rangle \underline{C}}$$

$$\frac{\mathsf{op} \colon A \rightsquigarrow_d B \qquad \Gamma \vdash^{\mathtt{g}} V : A \qquad \Gamma, y : B \vdash^{\mathtt{g}} M : \underline{C}}{\Gamma \vdash^{\mathtt{g}} \mathbf{do}\, y \leftarrow \mathsf{op}\, V\, \mathbf{then}\, M : \langle\!\langle d \rangle\!\rangle \underline{C}} \qquad \frac{\Gamma \vdash^{\mathtt{g}} M : \underline{C} \qquad \underline{C} <: \underline{D}}{\Gamma \vdash^{\mathtt{g}} \mathbf{coerce}_{\underline{D}}\, M : \underline{D}}$$

Figure 2 Typing rules for CBPVE returner types, operations, and coercions. These replace the rules of Figure 1b; the remaining CBPVE typing rules are those of Figures 1a and 1c.

Given an ordered monoid $\mathbb{E}$, we generate CBPVE types in the same way as for CBPV, except that each returner type is annotated with a grade $e \in \mathbb{E}$. Thus $\mathbf{F}_e A$ is the type of computations that return values of A, and which have grade e.

$$A, B \ ::= \ \mathbf{U}\underline{C} \mid \mathbf{1} \mid A_1 \times A_2 \mid \coprod_{i \in I} A_i \qquad \underline{C}, \underline{D} \ ::= \ \mathbf{F}_e A \mid \prod_{i \in I} \underline{C}_i \mid A \to \underline{C}$$

The ordered monoid structure of the grades $\mathbb{E}$ induces some structure on types, which we use to define the CBPVE typing judgements. The order $\leq$ on grades induces *subtyping* preorders $<:$ on both value types and computation types, defined inductively by lifting $\leq$ to returner types, and using the expected rules for the remaining type formers, including contravariance at function types:

$$\frac{\underline{C} <: \underline{D}}{\mathbf{U}\underline{C} <: \mathbf{U}\underline{D}} \qquad \frac{}{\mathbf{1} <: \mathbf{1}} \qquad \frac{A_1 <: B_1 \qquad A_2 <: B_2}{A_1 \times A_2 <: B_1 \times B_2} \qquad \frac{\{A_i <: B_i\}_{i \in I}}{\coprod_{i \in I} A_i <: \coprod_{i \in I} B_i}$$

$$\frac{d \leq e \qquad A <: B}{\mathbf{F}_d A <: \mathbf{F}_e B} \qquad \frac{\{\underline{C}_i <: \underline{D}_i\}_{i \in I}}{\prod_{i \in I} \underline{C}_i <: \prod_{i \in I} \underline{D}_i} \qquad \frac{B <: A \qquad \underline{C} <: \underline{D}}{A \to \underline{C} <: B \to \underline{D}}$$

The multiplication of the ordered monoid also lifts to computation types; for each grade d and computation type $\underline{C}$, we define a computation type $\langle\!\langle d \rangle\!\rangle \underline{C}$ as follows.

$$\langle\!\langle d \rangle\!\rangle (\mathbf{F}_e A) = \mathbf{F}_{d \cdot e} A \qquad \langle\!\langle d \rangle\!\rangle \left(\prod_{i \in I} \underline{C}_i\right) = \prod_{i \in I} \langle\!\langle d \rangle\!\rangle \underline{C}_i \qquad \langle\!\langle d \rangle\!\rangle (A \to \underline{C}) = A \to \langle\!\langle d \rangle\!\rangle \underline{C}$$

Informally, $\langle\!\langle d \rangle\!\rangle \underline{C}$ is the type of a computation that does something of grade d, and then continues as a computation of type $\underline{C}$. This defines an *action* of the ordered monoid of grades on the preorder of computation types, in the sense that $\langle\!\langle - \rangle\!\rangle$ is monotone (if $d \leq e$ and $\underline{C} <: \underline{D}$ then $\langle\!\langle d \rangle\!\rangle \underline{C} <: \langle\!\langle e \rangle\!\rangle \underline{D}$), and is unital ($\langle\!\langle \mathbf{1} \rangle\!\rangle \underline{C} = \underline{C}$) and associative ($\langle\!\langle d \rangle\!\rangle (\langle\!\langle d' \rangle\!\rangle \underline{C}) = \langle\!\langle d \cdot d' \rangle\!\rangle \underline{C}$).

We extend the syntax of computations by adding explicit subtype *coercions*; the syntax of computations and of values is otherwise unchanged from CBPV.

$$M ::= \cdots \mid \mathbf{coerce}_{\underline{D}}\, M$$

The typing judgements $\Gamma \vdash^{\mathtt{g}} V : A$ and $\Gamma \vdash^{\mathtt{g}} M : \underline{C}$ are also similar to CBPV, but we add the superscript $\mathtt{g}$ to distinguish them. The majority of the rules generating these are identical to those of CBPV; the new rules are given in Figure 2.

This concludes the basic syntax of our calculus CBPVE. When discussing the coherence result in Section 6, we will also need the equational theory $\equiv$ for CBPVE, which provides a notion of equality between computations of the same type. It is generated by the β- and η-laws for each type, *sequencing laws* (as in CBPV), and also coercion laws, which permit coercions to be moved around in terms. For space reasons, we list the laws only in the appendix (Figure 5), but none of them are surprising.

4 Subsuming call-by-value and call-by-name

CBPV subsumes both call-by-value and call-by-name, and it is important that we have an analogous fact for CBPVE. We show in this section that Levy's translations [18, 20] of *fine-grain* call-by-value, and of call-by-name, lift to the graded setting. For a proper subsumption result we need more than this, in particular subsumption of the CBV and CBN semantics and equational theories. These do not pose any particular difficulties, but we omit them for space reasons.

4.1 Fine grain call-by-value

We first consider fine-grain call-by-value [21], a call-by-value calculus that separates terms into values v and computations t. It is well-known how to add effects to fine-grain call-by-value. The only modifications we make to the syntax are to incorporate grades are to annotate each function type $\tau \to_e \tau'$ with a grade e, and to add coercions.

$$\begin{aligned} \tau &::= \mathbf{2} \mid \tau_1 \times \tau_2 \mid \tau \to_e \tau' \\ v, w &::= x \mid \mathbf{true} \mid \mathbf{false} \mid (v_1, v_2) \mid \lambda x : \tau. t \\ t, u &::= \mathbf{return}\, v \mid t \;\mathbf{to}\; x. u \mid \mathbf{do}\, y \leftarrow \mathsf{op}\, v \,\mathbf{then}\, t \mid \mathbf{coerce}_e\, t \\ &\quad \mid v\, w \mid \mathbf{if}\; v \;\mathbf{then}\; t_1 \;\mathbf{else}\; t_2 \mid \mathbf{match}\, v \,\mathbf{with}\, (x_1, x_2) \,\mathbf{in}\, t \end{aligned}$$

The typing judgements are $\Gamma \vdash v : \tau$ for values and $\Gamma \vdash t : \tau \,\&\, e$ for computations, where Γ is a list of (variable, type) pairs. Every computation is assigned a grade e. We assume a *graded CBV operation signature*, namely a collection of operations, each with a suitable typing $\mathsf{op} \colon \tau \rightsquigarrow_d \tau'$, where τ and τ' are ground types (they do not contain function types). The typing rules are as follows, where we omit the obvious rules for variables x, the boolean constants **true** and **false**, and pairs (v_1, v_2).

$$\frac{\Gamma, x : \tau \vdash t : \tau' \,\&\, e}{\Gamma \vdash \lambda x : \tau. t : \tau \to_e \tau'} \qquad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathbf{return}\, v : \tau \,\&\, \mathbf{1}} \qquad \frac{\Gamma \vdash t : \tau \,\&\, d \quad \Gamma, x : \tau \vdash u : \tau' \,\&\, e}{\Gamma \vdash t \;\mathbf{to}\; x. u : \tau' \,\&\, (d \cdot e)}$$

$$\frac{\mathsf{op} \colon \tau \rightsquigarrow_d \tau' \quad \Gamma \vdash v : \tau \quad \Gamma, y : \tau' \vdash t : \tau'' \,\&\, e}{\Gamma \vdash \mathbf{do}\, y \leftarrow \mathsf{op}\, v \,\mathbf{then}\, t : \tau'' \,\&\, d \cdot e} \qquad \frac{\Gamma \vdash t : \tau \,\&\, d \quad d \leq e}{\Gamma \vdash \mathbf{coerce}_e\, t : \tau \,\&\, e}$$

$$\frac{\Gamma \vdash v : \tau \to_e \tau' \quad \Gamma \vdash w : \tau}{\Gamma \vdash v\, w : \tau' \,\&\, e} \qquad \frac{\Gamma \vdash v : \mathbf{2} \quad \Gamma \vdash t_1 : \tau \,\&\, e \quad \Gamma \vdash t_2 : \tau \,\&\, e}{\Gamma \vdash \mathbf{if}\; v \;\mathbf{then}\; t_1 \;\mathbf{else}\; t_2 : \tau \,\&\, e}$$

$$\frac{\Gamma \vdash v : \tau_1 \times \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash t : \tau' \,\&\, e}{\Gamma \vdash \mathbf{match}\, v \,\mathbf{with}\, (x_1, x_2) \,\mathbf{in}\, t : \tau' \,\&\, e}$$

We translate each component of the syntax into CBPVE. Each type τ becomes a value type $(\!|\tau|\!)^{\mathrm{v}}$, and typing contexts are translated componentwise. Each operation $\mathsf{op} \colon \tau \rightsquigarrow_d \tau'$ becomes an operation $\mathsf{op} \colon (\!|\tau|\!)^{\mathrm{v}} \rightsquigarrow_d (\!|\tau'|\!)^{\mathrm{v}}$. Each value $\Gamma \vdash v : \tau$ becomes a value $(\!|\Gamma|\!)^{\mathrm{v}} \vdash^{\mathrm{g}} (\!|v|\!)^{\mathrm{v}} : (\!|\tau|\!)^{\mathrm{v}}$, and each computation $\Gamma \vdash t : \tau \,\&\, e$ becomes a computation $(\!|\Gamma|\!)^{\mathrm{v}} \vdash^{\mathrm{g}} (\!|t|\!)^{\mathrm{v}} : \mathbf{F}_e (\!|\tau|\!)^{\mathrm{v}}$. We omit most of the definition for the latter two, because the omitted parts are exactly the same as the standard translation [20].

$$(\!|\mathbf{2}|\!)^{\mathrm{v}} = \coprod_{i \in \{1,2\}} \mathbf{1} \qquad (\!|\tau_1 \times \tau_2|\!)^{\mathrm{v}} = (\!|\tau_1|\!)^{\mathrm{v}} \times (\!|\tau_2|\!)^{\mathrm{v}} \qquad (\!|\tau \to_e \tau'|\!)^{\mathrm{v}} = \mathbf{U}((\!|\tau|\!)^{\mathrm{v}} \to \mathbf{F}_e (\!|\tau'|\!)^{\mathrm{v}})$$

$$(\!|\mathbf{do}\, y \leftarrow \mathsf{op}\, v \,\mathbf{then}\, t|\!)^{\mathrm{v}} = \mathbf{do}\, y \leftarrow \mathsf{op}(\!|v|\!)^{\mathrm{v}} \,\mathbf{then}\, (\!|t|\!)^{\mathrm{v}}$$

$$(\!|\mathbf{coerce}_e\, t|\!)^{\mathrm{v}} = \mathbf{coerce}_{\mathbf{F}_e (\!|\tau|\!)^{\mathrm{v}}}\, (\!|t|\!)^{\mathrm{v}}$$

4.2 Call-by-name

We also show that CBPVE subsumes a graded call-by-name calculus. The design of this calculus is based on the same insight as the design of CBPVE, namely that we only observe effects at some computation types, and hence grades are only present at these types. In call-by-name, we only observe effects at ground types γ (the only ground type we include here is $\mathbf{2}$), and hence we annotate ground types with grades e. Call-by-name does not have a separate notion of value; there are only computations.

$$\begin{aligned} \tau &::= \gamma_e \mid \tau_1 \times \tau_2 \mid \tau \to \tau' \qquad \gamma ::= \mathbf{2} \\ t, u &::= x \mid \mathsf{op}(t) \mid \mathbf{coerce}_\tau\, t \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ t\ \mathbf{then}\ u_1\ \mathbf{else}\ u_2 \\ &\quad \mid (t_1, t_2) \mid \mathbf{fst}\, t \mid \mathbf{snd}\, t \mid \lambda x\colon \tau.\, t \mid t\, u \end{aligned}$$

As for CBPVE, there is again a subtyping relation $\tau <: \tau'$, and an action $\langle\!\langle d \rangle\!\rangle \tau$ of grades on types. The latter is given by $\langle\!\langle d \rangle\!\rangle \gamma_e = \gamma_{d \cdot e}$ for ground types, $\langle\!\langle d \rangle\!\rangle (\tau_1 \times \tau_2) = \langle\!\langle d \rangle\!\rangle \tau_1 \times \langle\!\langle d \rangle\!\rangle \tau_2$, and $\langle\!\langle d \rangle\!\rangle (\tau \to \tau') = \tau \to \langle\!\langle d \rangle\!\rangle \tau'$. The typing judgement has the form $\Gamma \vdash t : \tau$. The rules for variables, product types, and function types are identical to those for simply-typed lambda calculus, and the rules for **true** and **false** assign the type $\mathbf{2}_\mathbf{1}$. The remaining rules are as follows, where we again assume operations have suitable typing.

$$\frac{\mathsf{op}\colon \gamma \rightsquigarrow_e \gamma' \qquad \Gamma \vdash t : \gamma_d}{\Gamma \vdash \mathsf{op}(t) : \gamma'_{d \cdot e}} \qquad \frac{\Gamma \vdash t : \tau \qquad \tau <: \tau'}{\Gamma \vdash \mathbf{coerce}_{\tau'}\, t : \tau'} \qquad \frac{\Gamma \vdash t : \mathbf{2}_d \qquad \Gamma \vdash u_1 : \tau \qquad \Gamma \vdash u_2 : \tau}{\Gamma \vdash \mathbf{if}\ u\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : \langle\!\langle d \rangle\!\rangle \tau}$$

CBPVE subsumes our graded call-by-name calculus by mapping types τ to computation types, ground types γ to value types, typing contexts Γ to CBPVE typing contexts, and computations $\Gamma \vdash t : \tau$ to computations $(\!|\Gamma|\!)^{\mathrm{n}} \vdash (\!|t|\!)^{\mathrm{n}} : (\!|\tau|\!)^{\mathrm{n}}$ as follows. Again we omit much of the definition; the remainder is exactly the same usual translation of CBN into CBPV [18].

$$\begin{aligned} &(\!|\gamma_e|\!)^{\mathrm{n}} = \mathbf{F}_e (\!|\gamma|\!)^{\mathrm{n}} \quad (\!|\tau_1 \times \tau_2|\!)^{\mathrm{n}} = \textstyle\prod_{i \in \{1,2\}} (\!|\tau_i|\!)^{\mathrm{n}} \quad (\!|\tau \to \tau'|\!)^{\mathrm{n}} = \mathbf{U} (\!|\tau|\!)^{\mathrm{n}} \to (\!|\tau'|\!)^{\mathrm{n}} \\ &(\!|\mathbf{2}|\!)^{\mathrm{n}} = \textstyle\coprod_{i \in \{1,2\}} \mathbf{1} \qquad (\!|\cdot|\!)^{\mathrm{n}} = \cdot \qquad (\!|\Gamma, x : \tau|\!)^{\mathrm{n}} = (\!|\Gamma|\!)^{\mathrm{n}}, x : \mathbf{U} (\!|\tau|\!)^{\mathrm{n}} \\ &(\!|\mathsf{op}(t)|\!)^{\mathrm{n}} = (\!|t|\!)^{\mathrm{n}}\ \mathbf{to}\ x.\, \mathbf{do}\, y \leftarrow \mathsf{op}\, x\, \mathbf{then}\, \mathbf{return}\, y \qquad (\!|\mathbf{coerce}_\tau\, t|\!)^{\mathrm{n}} = \mathbf{coerce}_{(\!|\tau|\!)^{\mathrm{n}}} (\!|t|\!)^{\mathrm{n}} \end{aligned}$$

That $(\!|t|\!)^{\mathrm{n}}$ has the correct typing relies on the fact that $\tau <: \tau'$ implies $(\!|\tau|\!)^{\mathrm{n}} <: (\!|\tau'|\!)^{\mathrm{n}}$, and on $(\!|\langle\!\langle e \rangle\!\rangle \tau|\!)^{\mathrm{n}} = \langle\!\langle e \rangle\!\rangle (\!|\tau|\!)^{\mathrm{n}}$.

One of the characteristic features of call-by-name is the isomorphism $(\tau \times \tau') \to \tau'' \cong \tau \to (\tau' \to \tau'')$ given by uncurrying, which does not hold in call-by-value. This isomorphism holds in our graded CBN calculus up to suitable β- and η-laws. In particular, if $\Gamma \vdash t : (\tau \times \tau') \to \tau''$, then we have a computation $\Gamma \vdash (\lambda x : \tau.\, \lambda x' : \tau'.\, t\, (x, x')) : \tau \to (\tau' \to \tau'')$. If we had assigned a grade to each computation as in [12, 11, 32], and then forced elements of pairs to have the same grade, we would not have such an isomorphism. This provides some motivation for assigning grades only to returners in CBPVE.

5 Graded monad semantics

.

We give a categorical semantics for CBPVE, based on algebras[1] of a *graded monad* [5, 30, 14]. This is directly analogous to the semantics of CBPV using monad algebras [20]. By composing our CBPVE semantics with the fine grain call-by-value and the call-by-name translations, we therefore also obtain CBV and CBN semantics. In particular, we recover

the known graded monad interpretation of fine-grain call-by-value with grades [1]. Since coercions are explicit in CBPVE, there are no issues with coherence in the semantics.

We assume that the base category $\mathbb{C}$ of each model is bicartesian closed. That is, the base category $\mathbb{C}$ has finite products, finite coproducts, and exponentials $X \Rightarrow Y$. We write $\Lambda f \colon W \to X \Rightarrow Y$ for the currying of a morphism $f \colon W \times X \to Y$, and write $\Lambda^{-1} g$ for uncurrying. Such a category is necessarily *distributive*, meaning that for every family of morphisms $f_i \colon W \times X_i \to Y$ indexed by the elements i of a finite set I, there is a unique morphism $[f_i]_{i \in I} \colon W \times \coprod_i X_i \to Y$ such that $f_i = [f_i]_{i \in I} \circ (\mathrm{id}_W \times \mathrm{in}_i)$, where $\mathrm{in}_i \colon X_i \to \coprod_i X_i$ is the ith coprojection.

We continue to assume a given ordered monoid $(\mathbb{E}, \leq, \mathbf{1}, \cdot)$ of grades, the same ordered monoid used for the syntax.

▶ **Definition 3.** *A* graded object X *of a category* $\mathbb{C}$ *is a functor* $X \colon \mathbb{E} \to \mathbb{C}$*; that is, a family of objects* Xd *indexed by grades* $d \in \mathbb{E}$*, together with a* coercion *morphism* $e^* \colon Xd \to Xe$ *for each* $e \geq d$*, satisfying identity and composition laws.*

We need our graded monads to be *strong*, just as models of Moggi's monadic metalanguage [25] use a strong monad. Instead of presenting the strength as some extra on top of the notion of graded monad, it is more convenient for us to bake strength into the (Kleisli) extension operator of the graded monad. This is equivalent to the standard definition of strong graded monad [14], for the same reasons that we can bake the strength into the Kleisli extension of an ordinary strong monad (see for example [24]). Thus we go directly to our definition of strong graded monad.

▶ **Definition 4.** *A* strong graded monad T *on* $\mathbb{C}$ *consists of the following data.*

- *A graded object* TX *of* $\mathbb{C}$*, for each* $X \in \mathbb{C}$*;*
- *a* unit *morphism* $return_X \colon X \to TX\mathbf{1}$ *for each* $X \in \mathbb{C}$*;*
- *a* Kleisli extension *operator* $(-)^\dagger : \mathbb{C}(W \times X, TYe) \to \mathbb{C}(W \times TXd, TY(d \cdot e))$ *for each* $W, X, Y \in \mathbb{C}$ *and* $d, e \in \mathbb{E}$*.*

We require $(-)^\dagger$ *to be natural in* W*, and in both of the grades* d, e*. We also require that graded analogues of the three monad laws hold. These are as follows, using the canonical isomorphisms* $assoc_{W',W,X} \colon (W' \times W) \times X \cong W' \times (W \times X)$ *and* $lunit_X \colon 1 \times X \cong X$*.*

$$
\begin{aligned}
f^\dagger \circ (id_W \times return_X) &= f && \text{for all } f : W \times X \to TYe \\
(return_X \circ lunit_X)^\dagger &= lunit_{TXd} && \text{for all } X \in \mathbb{C}, d \in \mathbb{E} \\
(g^\dagger \circ (id_{W'} \times f) \circ assoc_{W',W,X})^\dagger &= g^\dagger \circ (id_{W'} \times f^\dagger) \circ assoc_{W',W,TXd} && \text{for all } \begin{array}{l} f : W \times X \to TYe \\ g : W' \times Y \to TZe' \\ d \in \mathbb{E} \end{array}
\end{aligned}
$$

To interpret computations, we need to assume an interpretation of each operation op in the graded monad. We summarize the requirements we have in the following definition.

▶ **Definition 5.** *A* graded model *consists of a bicartesian closed category* $\mathbb{C}$*, a strong graded monad* T *on* $\mathbb{C}$*, and a morphism* $\kappa_{\mathsf{op}} \colon [\![A]\!] \to T[\![B]\!]d$ *for each* $\mathsf{op} \colon A \rightsquigarrow_d B$*.*

For instance, if the grades are natural numbers and there is an operation $\mathsf{print}_w \colon \mathbf{1} \rightsquigarrow_1 \mathbf{1}$ for each $w \in W$, then one can take the graded *writer* monad $TXe = \mathrm{List}_e W \times X$ on **Set**,

[1] One may wish to consider other semantics for CBPVE, such as a notion of adjunction model analogous to Levy's [19] adjunction models for CBPV. Fujii et al.'s [8] decompositions of a graded monad into an adjunction equipped with an action provide a hint for what this would look like.

where $\mathrm{List}_e W$ is the set of lists of length at most e. For $\kappa_{\mathsf{print}_w} : \{\star\} \to \mathrm{List}_1 W \times \{\star\}$, we take the function that picks out the singleton list $[w]$.

We interpret computation types as (graded) *algebras* for T [8]. As above, we present the notion of algebra in extension form, and bake the strength directly into the extension operator; again this is equivalent to the usual definition. We also use a *parameterized* (cf. [7]) notion of morphism between algebras.[2] This specializes to the usual notion of morphism by taking $W = 1$.

▶ **Definition 6.** *Let T be a strong graded monad on a cartesian category $\mathbb{C}$. A* T-algebra *is a graded object Z of $\mathbb{C}$, called the* carrier*, equipped with an* extension *operator $(-)^\ddagger : \mathbb{C}(W \times X, Ze) \to \mathbb{C}(W \times TXd, Z(d \cdot e))$, for each $W, X \in \mathbb{C}$ and $d, e \in \mathbb{E}$, natural in W, d, e, and satisfying the following.*

$$f^\ddagger \circ (id_W \times return_X) = f \qquad \text{for all } f : W \times X \to Ze$$

$$(g^\ddagger \circ (id_{W'} \times f) \circ assoc)^\ddagger = g^\ddagger \circ (id_{W'} \times f^\dagger) \circ assoc \qquad \text{for all } \begin{array}{l} f : W \times X \to TYe \\ g : W' \times Y \to Ze' \end{array}$$

If $W \in C$ is an object, and Z, Z' are T-algebras, then a parameterized T-algebra morphism *$h : W \times Z \to Z'$ is a family of morphisms $h_e : W \times Ze \to Z'e$, natural in e and satisfying $(h_e \circ \langle \pi_1, f \rangle)^\ddagger = h_{d \cdot e} \circ \langle \pi_1, f^\ddagger \rangle$ for all $d \in \mathbb{E}$ and $f : W \times X \to Ze$.*

▶ **Example 7.** Every graded operation signature Σ induces a free strong graded monad T_Σ on **Set**. We give an explicit construction of this graded monad, based on [15]. First, for each ground type A, we define its interpretation as a set $[\![A]\!]$:

$$[\![\mathbf{1}]\!] = \{\star\} \quad [\![A_1 \times A_2]\!] = [\![A_1]\!] \times [\![A_2]\!] \quad [\![\coprod_{i \in I} A_i]\!] = \coprod_{i \in I} [\![A_i]\!]$$

The elements t of the graded set TX are generated by elements x of X, which can be thought of as variables, the operation symbols op of Σ, and coercions, as follows. They are quotiented by smallest congruence relation $\approx$ generated by the three axioms below, where $t \in TXd$ and $k : [\![B]\!] \to TXe$.

$$\frac{x \in X}{x \in TX\mathbf{1}} \qquad \frac{\mathsf{op} : A \rightsquigarrow_d B \qquad a \in [\![A]\!] \qquad k : [\![B]\!] \to TXe}{\mathsf{op}(a; k) \in TX(d \cdot e)} \qquad \frac{t \in TXd \qquad d \leq e}{e^* t \in TXe}$$

$$t \approx d^* t \qquad e^*(d^* t) \approx e^* t \qquad \mathsf{op}(a; b \mapsto e'^*(kb)) \approx (d \cdot e')^*(\mathsf{op}(a; k))$$

The unit functions $return_X : X \to TX\mathbf{1}$ are given by the variables x, and Kleisli extension is given by substitution. This graded monad forms a graded model, by equipping it with the functions $\kappa_{\mathsf{op}} : [\![A]\!] \to T_\Sigma [\![B]\!]$ given by $\kappa_{\mathsf{op}}(a) = \mathsf{op}(a; b \mapsto b)$.

A T_Σ-algebra is equivalently a graded set Z, equipped with a function $[\![\mathsf{op}]\!] : [\![A]\!] \times ([\![B]\!] \Rightarrow Ze) \to Z(d \cdot e)$ for each $\mathsf{op} : A \rightsquigarrow_d B$ and $e \in \mathbb{E}$, natural in e. Given these functions, the corresponding extension operator is given by $f^\ddagger(w, t) = [\![t]\!](x \mapsto f(w, x))$, where for each $t \in TXd$, the function $[\![t]\!] : (X \Rightarrow Ze) \to Z(d \cdot e)$ is defined by

$$[\![x]\!]\rho = \rho(x) \qquad [\![\mathsf{op}(a; k)]\!]\rho = [\![\mathsf{op}]\!](a, b \mapsto [\![kb]\!]\rho) \qquad [\![e^* t]\!]\rho = e^*([\![t]\!]\rho)$$

A parameterized morphism $h : W \times Z \to Z'$ is equivalently a family of functions that preserves the interpretation of each operator, in the sense that $h_{d \cdot e}(w, [\![\mathsf{op}]\!](a, f)) = [\![\mathsf{op}]\!](a, b \mapsto h_e(w, kb))$.

[2] Parameterization is also present in Levy's locally indexed perspective on models of CBPV [19], and taking a locally indexed perspective on graded monads would likely be useful. We do not do so here simply because it requires setting up more definitions.

We will need the following constructions on T-algebras to interpret CBPVE. The first three are graded adaptations of the T-algebra constructions used to interpret CBPV with a strong monad. The final one is specific to the graded setting; it is the action of $\mathbb{E}$ on T-algebras that plays a crucial role in the theory of graded monads [8].

- The free T-algebra $F_T X$ on an object $X \in \mathbb{C}$ has the graded object TX as its carrier, and $(-)^\dagger$ as its extension operator.
- The finite product $\prod_i Z_i$ of T-algebras Z_i has carrier $(\prod_i Z_i)e = \prod_i (Z_i e)$. The extension operator is unique such that the projection morphisms $\pi_j \colon \prod_i (Z_i e) \to Z_j e$ form algebra morphisms $\prod_i Z_i \to Z_j$.
- The power $X \Rightarrow Z$ of a T-algebra Z by an object $X \in \mathbb{C}$ has as its carrier the graded object $(X \Rightarrow Z)e = X \Rightarrow Ze$ given by exponentials in $\mathbb{C}$. The extension operator is unique such that the evaluation morphisms $ev \colon X \times (X \Rightarrow Ze) \to Ze$ form a parameterized algebra morphism $X \times (X \Rightarrow Z) \to Z$.
- The T-algebra $e * Z$, where Z is a T-algebra and e is a grade, has carrier $(e*Z)e' = Z(e' \cdot e)$; the extension operator is given by specializing that of Z.

Under the above definitions, the extension operator for an algebra Z sends each $\mathbb{C}$-morphism $f \colon W \times X \to Ze$ to a parameterized T-algebra morphism $f^\ddagger \colon W \times F_T X \to e * Z$.

The interpretations of value types as objects of $\underline{C}$ and of computation types as T-algebras are defined in Figure 3a. It follows from the definition that $[\![\langle\!\langle e \rangle\!\rangle \underline{C}]\!] = e * [\![\underline{C}]\!]$. We also interpret typing contexts as objects $[\![\Gamma]\!]$ of $\mathbb{C}$, using finite products. To interpret the subtyping rules, we also define $\mathbb{C}$-morphisms $[\![A <: B]\!]$ and T-algebra morphisms $[\![\underline{C} <: \underline{D}]\!]$, as shown in Figure 3b.

Given a graded model as in Definition 5, the interpretations of values as morphisms $[\![V]\!] \colon [\![\Gamma]\!] \to [\![A]\!]$ and of computations as morphisms $[\![M]\!] \colon [\![\Gamma]\!] \to [\![\underline{C}]\!]\mathbf{1}$ are defined in Figure 3c. Both of these are morphisms in $\mathbb{C}$. Except for the typing, and for the interpretation of coercions, the definition is exactly what one would expect for CBPV. We do not have to do anything special to account for grading.

The main result of this section is soundness of the equational theory defined in Figure 5 (in the appendix).

▶ **Theorem 8.** *If $M \equiv M'$ then $[\![M]\!] = [\![M']\!]$, and if $V \equiv V'$ then $[\![V]\!] = [\![V']\!]$.*

Proof. By induction on the derivations of $M \equiv M'$ and of $V \equiv V'$. Verifying the β-, η- and sequencing laws is standard. The first few coercion laws need identity and composition laws for the interpretation of subtyping, along with the fact that $[\![\langle\!\langle e \rangle\!\rangle \underline{C} <: \langle\!\langle e \rangle\!\rangle \underline{D}]\!] = e * [\![\underline{C} <: \underline{D}]\!]$; these are easy to show. The remaining coercion laws either follow from the definition of the subtyping morphisms, or are immediate from the interpretation of computations. ◀

6 Graded call-by-push-value, implicitly

The discussion of the previous sections treats CBPVE as an entirely separate calculus to CBPV, with its own syntax of computations. The alternative approach, as discussed in the introduction, is to view the ability to assign particular grades as a property of a computation. In this view, we retain the original syntax of CBPV values V and computations M, but have judgements $\Gamma \vdash^{\mathtt{i}} V : \underline{C}$ and $\Gamma \vdash^{\mathtt{i}} M : \underline{C}$, where Γ, A, and $\underline{C}$ are contexts and types from CBPVE. These judgements express stronger properties than the typing judgements of ordinary CBPV. The primary difference with this approach is that there are no explicit

$$[\![\mathbf{U}\underline{C}]\!] = [\![\underline{C}]\!]\mathbf{1} \quad [\![\mathbf{1}]\!] = \{\star\} \quad [\![A_1 \times A_2]\!] = [\![A_1]\!] \times [\![A_2]\!] \quad \left[\!\!\left[\coprod_{i\in I} A_i\right]\!\!\right] = \coprod_i [\![A_i]\!]$$
$$[\![\mathbf{F}_e A]\!] = e * F_T[\![A]\!] \quad \left[\!\!\left[\prod_{i\in I} \underline{C}_i\right]\!\!\right] = \prod_i [\![\underline{C}_i]\!] \quad [\![A \to \underline{C}]\!] = [\![A]\!] \Rightarrow [\![\underline{C}]\!]$$
$$[\![\cdot]\!] = \{\star\} \quad [\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times [\![A]\!]$$

(a) Interpretation of value types A as objects $[\![A]\!]$ of $\mathbb{C}$, of computation types $\underline{C}$ as T-algebras $[\![\underline{C}]\!]$, and of typing contexts Γ as objects $[\![\Gamma]\!]$ of $\mathbb{C}$.

$$[\![\mathbf{U}\underline{C} <: \mathbf{U}\underline{D}]\!] = [\![\underline{C} <: \underline{D}]\!]_{\mathbf{1}} \qquad [\![\mathbf{1} <: \mathbf{1}]\!] = id$$
$$[\![A_1 \times A_2 <: B_1 \times B_2]\!] = [\![A_1 <: B_1]\!] \times [\![A_2 <: B_2]\!] \quad \left[\!\!\left[\coprod_{i\in I} A_i <: \coprod_{i\in I} B_i\right]\!\!\right] = \coprod_i [\![A_i <: B_i]\!]$$
$$[\![\mathbf{F}_d A <: \mathbf{F}_e B]\!]_{e'} = (e' \cdot e)^* \circ (\mathit{return} \circ [\![A <: B]\!])^\dagger$$
$$\left[\!\!\left[\prod_{i\in I} \underline{C}_i <: \prod_{i\in I} \underline{D}_i\right]\!\!\right]_e = \prod_i [\![\underline{C}_i <: \underline{D}_i]\!]_e$$
$$[\![A \to \underline{C} <: B \to \underline{D}]\!]_e = [\![B <: A]\!] \Rightarrow [\![\underline{C} <: \underline{D}]\!]_e$$

(b) Interpretation of value subtyping as morphisms $[\![A <: B]\!] \colon [\![A]\!] \to [\![B]\!]$, and of computation subtyping as T-algebra morphisms $[\![\underline{C} <: \underline{D}]\!] \colon [\![\underline{C}]\!] \to [\![\underline{D}]\!]$.

$$[\![x]\!] = \pi_x \quad [\![\mathbf{thunk}\, M]\!] = [\![M]\!] \quad [\![()]\!] = \langle\rangle_{[\![\Gamma]\!]} \quad [\![(V_1, V_2)]\!] = \langle [\![V_1]\!], [\![V_2]\!] \rangle \quad [\![\mathbf{inj}_i\, V]\!] = \mathrm{in}_i \circ [\![V]\!]$$

$$[\![\mathbf{return}\, V]\!] = \mathit{return} \circ [\![V]\!] \qquad [\![M\ \mathbf{to}\ x.\, N]\!] = [\![N]\!]^\ddagger \circ \langle \mathit{id}, [\![M]\!] \rangle$$
$$[\![\mathbf{do}\, y \leftarrow \mathsf{op}\, V\, \mathbf{then}\, M]\!] = [\![M]\!]^\ddagger \circ \langle \mathit{id}, \kappa_{\mathsf{op}} \circ [\![V]\!] \rangle \qquad [\![\mathbf{coerce}_{\underline{D}}\, M]\!] = [\![\underline{C} <: \underline{D}]\!]_{\mathbf{1}} \circ [\![M]\!]$$
$$[\![\lambda\{i.\, M_i\}_{i\in I}]\!] = \langle [\![M_i]\!] \rangle_{i\in I} \quad [\![i\text{`}M]\!] = \pi_i \circ [\![M]\!]$$
$$[\![\lambda x : A.\, M]\!] = \Lambda[\![M]\!]$$
$$[\![V\text{`}M]\!] = \Lambda^{-1}[\![M]\!] \circ \langle \mathit{id}, [\![V]\!] \rangle$$
$$[\![\mathbf{match}\, V\, \mathbf{with}\, (x, y)\, \mathbf{in}\, M]\!] = [\![M]\!] \circ \mathit{assoc}^{-1} \circ \langle \mathit{id}, [\![V]\!] \rangle$$
$$[\![\mathbf{match}\, V\, \mathbf{with}\, \{\mathbf{inj}_i\, x_i.\, M_i\}_{i\in I}]\!] = [[\![M_i]\!]_i \circ \langle \mathit{id}, [\![V]\!] \rangle$$
$$[\![\mathbf{force}\, V]\!] = [\![V]\!]$$

(c) Interpretation of values $\Gamma \vdash^{\mathtt{g}} V : A$ as morphisms $[\![V]\!] \colon [\![\Gamma]\!] \to [\![A]\!]$ of $\mathbb{C}$ and of computations $\Gamma \vdash^{\mathtt{g}} M : \underline{C}$ as morphisms $[\![M]\!] \colon [\![\Gamma]\!] \to [\![\underline{C}]\!]\mathbf{1}$ of $\mathbb{C}$. The morphism $\pi_x \colon [\![\Gamma]\!] \to [\![A]\!]$ is the projection corresponding to the variable x, where $(x : A) \in \Gamma$.

Figure 3 Graded monad semantics of CBPVE

coercions in the syntax of computations, instead there is an admissible rule

$$\frac{\Gamma \vdash^{\mathtt{i}} M : \underline{C} \qquad \underline{C} <: \underline{D}}{\Gamma \vdash^{\mathtt{i}} M : \underline{D}}$$

We can define these judgements using the CBPVE typing judgements. For a CBPVE computation M, we let $\lfloor M \rfloor$ be the CBPV computation that arises by erasing all grade information, with $\lfloor \mathbf{coerce}_{\underline{D}}\, M \rfloor = \lfloor M \rfloor$; we similarly define $\lfloor - \rfloor$ for all of the other parts of the CBPVE syntax. Then we can view a CBPVE computation $\Gamma \vdash^{\mathtt{g}} M' : \underline{C}$ such that $\lfloor M' \rfloor = M$ as a witness that the typing judgement $\Gamma \vdash^{\mathtt{i}} M : \underline{C}$ holds, as in the following definition.

▶ **Definition 9.** *We write $\Gamma \vdash^{\mathtt{i}} M : \underline{C}$ when there exists some M' such that $\Gamma \vdash^{\mathtt{g}} M' : \underline{C}$ and $\lfloor M' \rfloor = M$. We define $\Gamma \vdash^{\mathtt{i}} V : A$ similarly.*

One can then read off typing rules that generate $\Gamma \vdash^{\mathtt{i}} M : \underline{C}$, including the implicit coercion rule above, but we will not state these rules explicitly.

If $\Gamma \vdash^{\mathtt{i}} M : \underline{C}$ holds, then we might expect to be able to interpret the CBPV computation $[\![M]\!]$ as a morphism $[\![M]\!] \colon [\![\Gamma]\!] \to [\![\underline{C}]\!]\mathbf{1}$ in any graded model, where $[\![\Gamma]\!]$ and $[\![\underline{C}]\!]$ are as defined in the previous section. The way to do this is to leverage the interpretation of CBPVE computations, defining

$$[\![M]\!] = [\![M']\!] \quad \text{where } M' \text{ witnesses } \Gamma \vdash^{\mathtt{i}} M : \underline{C}$$

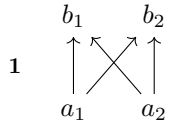
The problem is that, in general, there are several different witnesses M'; for $[\![M]\!]$ to be well-defined, we need all of them to have identical interpretations. Thus, we would like the graded model to be *coherent*, in the sense that

$$\lfloor M_1 \rfloor = \lfloor M_2 \rfloor \text{ implies } [\![M_1]\!] = [\![M_2]\!] \quad \text{for all } \Gamma \vdash^{\mathtt{g}} M_1 : \underline{C} \text{ and } \Gamma \vdash^{\mathtt{g}} M_2 : \underline{C}$$

and similarly for interpretations of values.

Sadly, it turns out that coherence is *false* in general, as the following example demonstrates.

▶ **Example 10.** Consider the following 5-element ordered monoid:



$$\mathbf{1} \cdot e = e = e \cdot \mathbf{1}$$
$$d \cdot e = b_2 = e \cdot d \quad (\mathbf{1} \notin \{d, e\})$$

Let $\Gamma = x : \mathbf{U}(\mathbf{F}_{b_2}\mathbf{2}), y_1 : \mathbf{U}(\mathbf{F}_{a_1}\mathbf{0}), y_2 : \mathbf{U}(\mathbf{F}_{a_2}\mathbf{0})$, where $\mathbf{2} = \coprod_{i \in \{1,2\}} \mathbf{1}$. Our counterexample to coherence consists of the following two computations $\Gamma \vdash^{\mathtt{g}} M_i : \mathbf{F}_{b_2}\mathbf{0}$ $(i \in \{1, 2\})$.

$$M_i = \mathbf{force}\, x \,\mathbf{to}\, z.\, \mathbf{match}\, z \,\mathbf{with}\, \{\mathbf{inj}_j\, w.\, \mathbf{coerce}_{\mathbf{F}_{b_i}\mathbf{0}}\, (\mathbf{force}\, y_j)\}_{j \in \{1,2\}}$$

These two computations satisfy $\lfloor M_1 \rfloor = \lfloor M_2 \rfloor$, and while they differ in the type $\mathbf{F}_{b_i}\mathbf{0}$ of the coercion, M_1 and M_2 have a common type $\mathbf{F}_{b_2}\mathbf{0} = \langle\!\langle b_2 \rangle\!\rangle(\mathbf{F}_{b_1}\mathbf{0}) = \langle\!\langle b_2 \rangle\!\rangle(\mathbf{F}_{b_2}\mathbf{0})$. Nevertheless, they can be distinguished by a graded model. Before showing how to do this, we note that if one were to add an empty ground type to the call-by-name calculus (or replace $\mathbf{0}$ with $\mathbf{2}$ above), then computations M_1 and M_2 would be in the image of the call-by-name translation. Thus, the same coherence problem is already present in call-by-name. We can also adjust the counterexample to one that involves translations of call-by-value computations, by replacing the typing context with $\Gamma = x : \mathbf{U}(\mathbf{1} \to \mathbf{F}_{b_2}\mathbf{2}), y_1 : \mathbf{U}(\mathbf{1} \to \mathbf{F}_{a_1}\mathbf{0}), y_2 : \mathbf{U}(\mathbf{1} \to \mathbf{F}_{a_2}\mathbf{0})$; and thus coherence is also an issue when grading call-by-value.

The rest of this example is devoted to showing that the terms M_1 and M_2 can be distinguished by a graded model. Let Σ be the graded operation signature consisting of three operation symbols

$$\mathsf{op}_1 \colon \mathbf{1} \rightsquigarrow_{a_1} \mathbf{0} \quad \mathsf{op}_2 \colon \mathbf{1} \rightsquigarrow_{a_2} \mathbf{0} \quad \mathsf{op}_3 \colon \mathbf{1} \rightsquigarrow_{b_2} \mathbf{2}$$

and consider the free graded monad T_Σ on **Set** (Example 7). The computations M_i are interpreted as functions $[\![M_i]\!] \colon T_\Sigma[\![\mathbf{2}]\!]b_2 \times T_\Sigma 0 a_1 \times T_\Sigma 0 a_2 \to T_\Sigma 0 b_2$. The set $T_\Sigma[\![\mathbf{2}]\!]b_2$ has an element $\mathsf{op}_3(\star; x \mapsto x)$, and the two sets $T_\Sigma 0 a_i$ each have a unique element $\mathsf{op}_i(\star; \_)$. Applying the functions $[\![M_i]\!]$ to these yield the following two elements of $T_\Sigma 0 b_2$.

$$t_i = \mathsf{op}_3(\star; j \mapsto b_i^*(\mathsf{op}_j(\star; \_))) \quad \text{where } [\![\mathbf{2}]\!] = \{1, 2\}$$

To conclude that $[\![M_1]\!] \neq [\![M_2]\!]$, it is then enough to show that $t_1 \neq t_2$. This is not quite trivial, because the construction of the free graded monad T_Σ involves a quotient, by an

equivalence relation $\approx$. We show that t_1 and t_2 are distinct by exhibiting a T_Σ-algebra Z such that $[\![t_1]\!](\_) \neq [\![t_2]\!](\_)$, where $\_$ is the unique function $0 \to Zb_2$. The carrier of Z is the graded set given by

$$Z\mathbf{1} = \{\} \quad Za_1 = Zb_1 = \{1\} \quad Za_2 = \{0\} \quad Zb_2 = \{0, 1\}$$

The coercion functions are inclusions, except for ${b_1}^* \colon Za_2 \to Zb_1$, which is the unique function $\{0\} \to \{1\}$. The operator symbols are interpreted as follows, where $\oplus$ is addition modulo 2.

$$[\![\mathsf{op}_1]\!](\_, \_) = 1 \quad [\![\mathsf{op}_2]\!](\_, \_) = 0 \quad [\![\mathsf{op}_3]\!](\_, (x, y)) = x \oplus y$$

Verifying that this is a T_Σ-algebra is trivial, and we can calculate that this algebra does distinguish between t_1 and t_2, as required: $[\![t_1]\!](\_) = 1 \oplus 1 = 0 \neq 1 = 1 \oplus 0 = [\![t_2]\!](\_)$.

The good news is that there are many cases in which the above interpretation of $\Gamma \vdash^{\mathtt{i}} M : \underline{C}$ does work. In particular, we show that, if we assume the following condition on the ordered monoid of grades, then coherence *does* hold, for every graded model.

▶ **Definition 11.** *An ordered monoid* $\mathbb{E}$ *has* left-cancellative upper bounds *if, whenever*

$$d \cdot e_1 \leq d' \geq d \cdot e_2$$

there exists some e' *such that* $e_1 \leq e' \geq e_2$ *and* $d \cdot e' \leq d'$.

This condition is a mild one. For instance, it is common for grades to have least upper bounds (cf. [26]), and for multiplication $\cdot$ to distribute over least upper bounds from the left ($d \cdot (e_1 \sqcup e_2) = (d \cdot e_1) \sqcup (e \cdot e_2)$). This is enough to guarantee that $\mathbb{E}$ has left-cancellative upper bounds, because if we have $d \cdot e_1 \leq d' \geq d \cdot e_2$, then we can take $e' = e_1 \sqcup e_2$.

Having left-cancellative upper bounds guarantees coherence, and hence that the interpretation of $\Gamma \vdash^{\mathtt{i}} M : \underline{C}$ is well-defined.

▶ **Theorem 12.** *Assume that* $\mathbb{E}$ *has left-cancellative upper bounds. For all computations* $\Gamma \vdash M_i : \underline{C}$ *such that* $\lfloor M_1 \rfloor = \lfloor M_2 \rfloor$*, we have* $M_1 \equiv M_2$*. It follows that* $[\![M_1]\!] = [\![M_2]\!]$ *for every graded model.*

Proof. We defer the proof of $M_1 \equiv M_2$ to Section 6.1 below. That the interpretations are equal follows by Theorem 8. ◀

For instance, consider what happens when we adjust our counterexample by adding an upper bound.

▶ **Example 13.** For our counterexample to coherence (Example 10), the ordered monoid does not have left-cancellative upper bounds: we have $b_2 \cdot b_1 \leq b_2 \geq b_2 \cdot b_2$, but b_1 and b_2 do not have an upper bound. Suppose that we were to add an upper bound b', with $b' \cdot e = b_2 = e \cdot b'$. This adjusted ordered monoid does have left-cancellative upper bounds. Accordingly, we can show that $M_1 \equiv M_2$ with the adjusted ordered monoid: for each i we have

$$\begin{aligned} M_i &\equiv \mathbf{coerce}_{\mathbf{F}_{b_2 \cdot b'}\mathbf{0}}\, M_i \\ &\equiv \mathbf{force}\, x\ \mathbf{to}\ z.\, \mathbf{match}\, z\, \mathbf{with}\, \{\mathbf{inj}_j\, w.\, \mathbf{coerce}_{\mathbf{F}_{b'}\mathbf{0}}\, (\mathbf{coerce}_{\mathbf{F}_{b_i}\mathbf{0}}\, (\mathbf{force}\, y_j))\}_{j \in \{1,2\}} \\ &\equiv \mathbf{force}\, x\ \mathbf{to}\ z.\, \mathbf{match}\, z\, \mathbf{with}\, \{\mathbf{inj}_j\, w.\, \mathbf{coerce}_{\mathbf{F}_{b'}\mathbf{0}}\, (\mathbf{force}\, y_j)\}_{j \in \{1,2\}} \end{aligned}$$

In the above example, we show $M_1 \equiv M_2$ using only the coercion laws of Figure 5d. We emphasize however that coherence is not simply a matter of moving coercions around; as the following example demonstrates, we need to employ the other laws in some cases.

▶ **Example 14.** Let $\mathbb{E}$ be the positive integers with multiplication and the discrete ordering, and suppose that the graded operation signature contains an operation symbol $\mathsf{raise} \colon \mathbf{1} \rightsquigarrow_1 \mathbf{0}$, for raising an exception. For every type $\underline{C}$, we have the following closed computation $\mathbf{raise}_{\underline{C}} : \underline{C}$.

$$\mathbf{raise}_{\underline{C}} = \mathbf{do}\, y \leftarrow \mathsf{raise}()\, \mathbf{then}\, \mathbf{match}\, y\, \mathbf{with}\, \{\}$$

Thus, for all positive integers i, j, we have a closed computation $M_{i,j} : \mathbf{F}_{i \cdot j}\mathbf{0}$:

$$M_{i,j} = \mathbf{raise}_{\mathbf{F}_i\mathbf{0}}\ \mathbf{to}\ x.\, \mathbf{raise}_{\mathbf{F}_j\mathbf{0}}$$

The computations $M_{i,j}$ and $M_{j,i}$ have the same type, and satisfy $\lfloor M_{i,j} \rfloor = \lfloor M_{j,i} \rfloor$, but (for $i \neq j$), we clearly cannot show $M_{i,j} \equiv M_{j,i}$ using the coercion laws of Figure 5d alone, because the ordering on $\mathbb{E}$ is discrete. It is true that $M_{i,j} \equiv M_{j,i}$ however, using one of the sequencing laws and the η-law for sum types.

$$M_{i,j} \;\equiv\; \mathbf{do}\, y \leftarrow \mathsf{raise}()\, \mathbf{then}\, (\mathbf{match}\, y\, \mathbf{with}\, \{\}\ \mathbf{to}\ x.\, \mathbf{raise}_{\mathbf{F}_j\mathbf{0}}) \;\equiv\; \mathbf{raise}_{\mathbf{F}_{i \cdot j}\mathbf{0}}$$

This example also illustrates one of the main difficulties in the proof of coherence. Even though $M_{i,j}$ and $M_{j,i}$ have the same type, they involve subterms $\mathbf{raise}_{\mathbf{F}_i\mathbf{0}}$ and $\mathbf{raise}_{\mathbf{F}_j\mathbf{0}}$ of incomparable types, at the same positions. In our proof of coherence, we are forced to compare terms of incomparable types. We explain how we do this in Section 6.1 below.

6.1 Coherence proof

The basic idea behind the proof of Theorem 12 is to compare CBPVE terms $\Gamma \vdash^{\mathtt{g}} M_i : \underline{C}_i$, where $\lfloor \underline{C}_1 \rfloor = \lfloor \underline{C}_2 \rfloor$ using a logical relation. The definition of the logical relation is based on $\top\top$-*lifting* [22]: we compare the terms M_1 and M_2 by looking at what happens in related *continuations*, which, in our case, are also CBPVE computations. Since we wish to prove a result ($M_1 \equiv M_2$) about open terms, we employ logical relations *of varying arity* [10], which are indexed by typing contexts Γ. Varying-arity $\top\top$-lifting has previously been applied to calculi with sum types by Katsumata [13].

The logical relation consists of a relation $\mathcal{R}[\![\underline{D}_1, \underline{D}_2]\!]_\Gamma$ between CBPVE computations $\Gamma \vdash^{\mathtt{g}} M_i : \underline{D}_i$, for each CBPVE typing context Γ and pair of CBPVE computation types with $\lfloor \underline{D}_1 \rfloor = \lfloor \underline{D}_2 \rfloor$. These are defined mutually inductively on $\lfloor \underline{D}_i \rfloor$, together with analogous relations $\mathcal{R}[\![B_1, B_2]\!]_\Gamma$ between CBPVE values.

We write $\Gamma' \triangleright \Gamma$ to mean that Γ is a sublist of Γ', so that every computation $\Gamma \vdash^{\mathtt{g}} M : \underline{C}$ has a weakening $\Gamma' \vdash^{\mathtt{g}} M : \underline{C}$, and similarly for values. All of the relations we define are closed under weakening, so for instance, if $\Gamma' \triangleright \Gamma$ and $(M_1, M_2) \in \mathcal{R}[\![\underline{D}_1, \underline{D}_2]\!]_\Gamma$, then $(M_1, M_2) \in \mathcal{R}[\![\underline{D}_1, \underline{D}_2]\!]_{\Gamma'}$. If $\Gamma \vdash^{\mathtt{g}} M_i : \mathbf{F}_{d_i} A_i$ and $\Gamma, x : A_i \vdash^{\mathtt{g}} K_i : \underline{D}_i$, then we write $(M_1, M_2) \bot (K_1, K_2)$ when

$$\forall \underline{D} :> \langle\!\langle d_1 \rangle\!\rangle \underline{D}_1, \langle\!\langle d_2 \rangle\!\rangle \underline{D}_2.\ \mathbf{coerce}_{\underline{D}}\, (M_1\ \mathbf{to}\ x.\, K_1) \equiv \mathbf{coerce}_{\underline{D}}\, (M_2\ \mathbf{to}\ x.\, K_2)$$

The definition of the logical relation is given in Figure 4. The logical relation also involves relations $\mathcal{R}[\![A_1, A_2]\!]^\top_\Gamma$ between *continuations*, which in our case are computations $\Gamma, x : A_i \vdash^{\mathtt{g}} K_i : \underline{D}_i$.

The crucial fact about our logical relation is that, if we restrict to computations M_i that happen to have the same type, it exactly matches $\equiv$:

$(M_1, M_2) \in \mathcal{R}[\![\underline{D}_1, \underline{D}_2]\!]_\Gamma$ if:

$\forall \Gamma' \triangleright \Gamma, (K_1, K_2) \in \mathcal{R}[\![A_1, A_2]\!]^\top_{\Gamma'}. (M_1, M_2) \bot (K_1, K_2)$ for $\underline{D}_i = \mathbf{F}_{d_i} A_i$

$\forall j \in J. (j\text{`}M_1, j\text{`}M_2) \in \mathcal{R}[\![\underline{C}_{1,j}, \underline{C}_{2,j}]\!]_\Gamma\}$ for $\underline{D}_i = \prod_{j \in J} \underline{C}_{i,j}$

$\forall \Gamma' \triangleright \Gamma, (V_1, V_2) \in \mathcal{R}[\![A_1, A_2]\!]_{\Gamma'}. (V_1\text{`}M_1, V_2\text{`}M_2) \in \mathcal{R}[\![\underline{C}_1, \underline{C}_2]\!]_{\Gamma'}$ for $\underline{D}_i = A_i \to \underline{C}_i$

$(K_1, K_2) \in \mathcal{R}[\![A_1, A_2]\!]^\top_\Gamma$ if:

$\forall \Gamma' \triangleright \Gamma, (V_1, V_2) \in \mathcal{R}[\![A_1, A_2]\!]_{\Gamma'}. (\mathbf{return}\, V_1, \mathbf{return}\, V_2) \bot (K_1, K_2)$

$(V_1, V_2) \in \mathcal{R}[\![B_1, B_2]\!]_\Gamma$ if:

$(\mathbf{force}\, V_1, \mathbf{force}\, V_2) \in \mathcal{R}[\![\underline{C}_1, \underline{C}_2]\!]_\Gamma$ for $B_i = \mathbf{U}\underline{C_i}$

(always true) for $B_i = \mathbf{1}$

$\forall \Gamma' \triangleright \Gamma, \{\Gamma', x_1 : A_{i,1}, x_2 : A_{i,2} \vdash^{\mathtt{g}} M_i : \underline{D}\}_{i \in \{1,2\}}.$
$(\forall \Gamma'' \triangleright \Gamma', \{(W_{1,j}, W_{2,j}) \in \mathcal{R}[\![A_{1,j}, A_{2,j}]\!]_{\Gamma''}\}_{j \in \{1,2\}}. M_1[x_j \mapsto W_{1,j}] \equiv M_2[x_j \mapsto W_{2,j}])$
$\Rightarrow$ $\mathbf{match}\, V_1\, \mathbf{with}\, (x_1, x_2)\, \mathbf{in}\, M_1 \equiv \mathbf{match}\, V_2\, \mathbf{with}\, (x_1, x_2)\, \mathbf{in}\, M_2$
for $B_i = A_{i,1} \times A_{i,2}$

$\forall \Gamma' \triangleright \Gamma, \{\Gamma', x_j : A_{i,j} \vdash^{\mathtt{g}} M_{i,j} : \underline{D}\}_{i,j}.$
$(\forall j \in J, \Gamma'' \triangleright \Gamma', (W_1, W_2) \in \mathcal{R}[\![A_{1,j}, A_{2,j}]\!]_{\Gamma''}. M_1[x_j \mapsto W_1] \equiv M_2[x_j \mapsto W_2])$
$\Rightarrow$ $\mathbf{match}\, V_1\, \mathbf{with}\, \{\mathbf{inj}_j\, x_j. M_{1,j}\}_{j \in J} \equiv \mathbf{match}\, V_2\, \mathbf{with}\, \{\mathbf{inj}_j\, x_j. M_{2,j}\}_{j \in J}$
for $B_i = \coprod_{j \in J} A_{i,j}$

Figure 4 The logical relation used in the proof of coherence

▶ **Lemma 15.** *Assume that $\mathbb{E}$ has left-cancellative upper bounds.*

1. *If $\Gamma \vdash^{\mathtt{g}} M_i : \underline{C}$ for $i \in \{1, 2\}$, then $(M_1, M_2) \in \mathcal{R}[\![\underline{C}, \underline{C}]\!]_\Gamma$ iff $M_1 \equiv M_2$.*
2. *If $\Gamma \vdash^{\mathtt{g}} V_i : A$ for $i \in \{1, 2\}$, then $(V_1, V_2) \in \mathcal{R}[\![A, A]\!]_\Gamma$ iff, for all $\Gamma' \triangleright \Gamma$ and $\Gamma', x : A \vdash^{\mathtt{g}} M : \underline{C}$, we have $M[x \mapsto V_1] \equiv M[x \mapsto V_2]$.*

Proof. By mutual induction on $\underline{C}$ and A. We only give the case for returner types $\mathbf{F}_d A$, which demonstrates where we use our assumption on $\mathbb{E}$. If $(M_1, M_2) \in \mathcal{R}[\![\mathbf{F}_d A, \mathbf{F}_d A]\!]_\Gamma$, then we certainly have $(M_1, M_2) \bot (\mathbf{return}\, x, \mathbf{return}\, x)$ in context $\Gamma, x : A$. This implies $M_1 \equiv M_2$. For the other direction, assume that $M_1 \equiv M_2$. If $(K_1, K_2) \in \mathcal{R}[\![A, A]\!]^\top_{\Gamma'}$ with $\Gamma' \triangleright \Gamma$, and $\underline{D} :> \langle\!\langle d \rangle\!\rangle \underline{D}_i$, then there is some $\underline{C} :> \underline{D}_1, \underline{D}_2$ with $\langle\!\langle d \rangle\!\rangle \underline{C} <: \underline{D}$, by our assumption on $\mathbb{E}$. Since $(\Gamma', w : A) \triangleright \Gamma'$, we have $\mathbf{coerce}_{\underline{C}}\, (\mathbf{return}\, w\, \mathbf{to}\, x. K_1) \equiv \mathbf{coerce}_{\underline{C}}\, (\mathbf{return}\, w\, \mathbf{to}\, x. K_2)$ and so $\mathbf{coerce}_{\underline{C}}\, K_1 \equiv \mathbf{coerce}_{\underline{C}}\, K_2$. Since $M_1 \equiv M_2$, it follows that $\mathbf{coerce}_{\underline{D}}\, (M_1\, \mathbf{to}\, x. K_1) \equiv \mathbf{coerce}_{\underline{D}}\, (M_2\, \mathbf{to}\, x. K_2)$. ◀

▶ **Lemma 16** (Fundamental lemma)**.** *Assume that $\mathbb{E}$ has left-cancellative upper bounds, and let $\Delta_i = x_1 : B_{i,1}, \ldots, x_n : B_{i,n}$ be two typing contexts such that $\lfloor \Delta_1 \rfloor = \lfloor \Delta_2 \rfloor$. For all $\{(W_{1,j}, W_{2,j}) \in \mathcal{R}[\![B_{1,j}, B_{2,j}]\!]_\Gamma\}_{j=1,\ldots,n}$, we have the following.*

1. *If $\Delta_i \vdash^{\mathtt{g}} M_i : \underline{C}_i$ are two computations such that $\lfloor M_1 \rfloor = \lfloor M_2 \rfloor$, then $(M_1[x_j \mapsto W_{1,j}]_j, M_2[x_j \mapsto W_{2,j}]_j) \in \mathcal{R}[\![\underline{C}_1, \underline{C}_2]\!]_\Gamma$.*
2. *If $\Delta_i \vdash^{\mathtt{g}} V_i : A_i$ are two values such that $\lfloor V_1 \rfloor = \lfloor V_2 \rfloor$, then $(V_1[x_j \mapsto W_{1,j}]_j, V_2[x_j \mapsto W_{2,j}]_j) \in \mathcal{R}[\![A_1, A_2]\!]_\Gamma$.*

Proof. By induction on the structure of the terms M_i and V_i. If one of the terms M_i is a coercion, then we use the fact that the logical relation is closed under coercions in each

component of the pair of computations separately. If neither is a coercion, then they must involve the same term former. We handle each separately, and most of the cases are similar to other logical relations proofs. The case for **do** however, involves our assumption on $\mathbb{E}$. ◀

The above two lemmas together are enough to show the missing step in the proof of Theorem 12 above, namely that $\lfloor M_1 \rfloor = \lfloor M_2 \rfloor$ implies $M_1 \equiv M_2$. By Lemma 15, for every variable $(x : B) \in \Gamma$ we have $(x, x) \in \mathcal{R}[\![B, B]\!]_\Gamma$, so that Lemma 16 implies $(M_1, M_2) \in \mathcal{R}[\![\underline{C}, \underline{C}]\!]_\Gamma$. Thus Lemma 15 implies $M_1 \equiv M_2$ as required.

7 Related work

Grading CBPV As mentioned in the introduction, we are not the first to add grades to CBPV. Various authors have used graded calculi based on CBPV for specific purposes. For instance, Kammar et al. [11] use a Gifford-style effect system to add effect handlers [29] to CBPV, while Torczon et al. [32] use a general ordered monoid of grades, and also add *coeffects* [28]. All of these assign a grade to every computation, and thus have a typing judgement of the form $\Gamma \vdash M : \underline{C}\&e$. To give a type to **thunk** M, they then also annotate thunk types with grades e. We can embed all of this into CBPVE, by defining $(\!|\mathbf{U}_e\underline{C}|\!) = \mathbf{U}(\langle\!\langle e \rangle\!\rangle \underline{C})$, and mapping $\Gamma \vdash M : \underline{C}\&e$ to $(\!|\Gamma|\!) \vdash^{\mathtt{g}} (\!|M|\!) : \langle\!\langle e \rangle\!\rangle (\!|\underline{C}|\!)$. But since [12, 11, 32] require computations in tuples to have uniform grades, CBPVE is strictly more general. Moreover, unlike in [12, 11, 32], the assignment of grades in CBPVE matches the fact that we only observe effects at returner types.

Coherence in graded calculi While the coherence problem for subtyping is well-known e.g. in the context of recursive types [6], and logical relations techniques have been applied to proofs of coherence [4] in that context, the problem has not been seriously considered in the context of grading. For instance, [9] give a calculus with implicit coercions, but then only claim their semantics can interpret typing *derivations*; they do not claim an interpretation of *terms*. The only existing approach to interpreting the terms of an implicit graded calculus is the refinement approach proposed by Katsumata [14]. In this approach one first interprets the ungraded syntax (using a strong monad), and then defines the interpretation of the graded calculus as the lifting of this along a fibration. One has to assume an interpretation of the ungraded calculus, which seems unnecessarily strong in light of our coherence result.

8 Conclusions

Call-by-push-value is established as a central calculus in the study of computational effects. The purpose of CBPVE is to play the same role as CBPV, but for *graded* computational effects. CBPVE can be seen as a graded calculus that refines CBPV, the syntax being augmented with explicit effect information. Alternatively, CBPVE can be seen an inference system that can be overlaid on top of CBPV, as in the implicit approach (Definition 9) – at least when the appropriate coherence result holds. While coherence does not hold in general, a mild condition on the ordered monoid of grades suffices (Theorem 12). This paper does not provide a complete treatment of CBPVE, notable omissions being a discussion of the operational semantics, and of other notions of model, adjunction models in particular. We hope to correct these omissions in a subsequent paper, but for now the results of the present paper should provide a useful foundation for the study of graded computational effects.

References

1 Danel Ahman. When programs have to watch paint dry. In *FoSSaCS*, pages 1–23, 2023.
2 Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Logical methods in computer science*, 10, 2014.
3 Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 129–140. ACM, 1998. URL: http://doi.acm.org/10.1145/289423.289435, doi:10.1145/289423.289435.
4 Dariusz Biernacki and Piotr Polesiuk. Logical relations for coherence of effect subtyping. *Logical methods in computer science*, 14, 2018.
5 Francis Borceux, George Janelidze, and G Max Kelly. Internal object actions. *Comment. Math. Univ. Carolin.*, 46(2):235–255, 2005.
6 Val Breazu-Tannen, Thierry Coquand, Carl A Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and computation*, 93(1):172–221, 1991.
7 Marcelo Fiore and Philip Saville. List objects with algebraic structure. In Dale Miller, editor, *2nd Int. Conference on Formal Structures for Computation and Deduction, FSCD 2017*, volume 84 of *Leibniz Int. Proc. in Informatics*, pages 16:1–16:18. Dagstuhl Publishing, 2017. doi:10.4230/lipics.fscd.2017.16.
8 Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. Towards a formal theory of graded monads. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 513–530. Springer, 2016.
9 Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. Combining effects and coeffects via grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 476–489. ACM, 2016. URL: http://doi.acm.org/10.1145/2951913.2951939, doi:10.1145/2951913.2951939.
10 Achim Jung and Jerzy Tiuryn. A new characterization of lambda definability. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 245–257. Springer, 1993. doi:10.1007/BFb0037110.
11 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. *ACM SIGPLAN Notices*, 48(9):145–158, 2013.
12 Ohad Kammar and Gordon D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 349–360. ACM, 2012. URL: http://doi.acm.org/10.1145/2103656.2103698, doi:10.1145/2103656.2103698.
13 Shin-ya Katsumata. A characterisation of lambda definability with sums via $\top\top$-closure operators. In *Computer Science Logic: 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings 22*, pages 278–292. Springer, 2008.
14 Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proc. of 41st Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 633–645. ACM Press, New York, 2014. doi:10.1145/2535838.2535846.
15 Shin-ya Katsumata, Dylan McDermott, Tarmo Uustalu, and Nicolas Wu. Flexible presentations of graded monads. *Proceedings of the ACM on Programming Languages*, 6(ICFP):902–930, 2022.
16 Ariel E Kellison and Justin Hsu. Numerical fuzz: A type system for rounding error analysis. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1954–1978, 2024.
17 Satoshi Kura. Graded algebraic theories. In *International Conference on Foundations of Software Science and Computation Structures*, pages 401–421. Springer, 2020.
18 Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, pages 228–243. Springer, 1999. doi:10.1007/3-540-48959-2_17.

19 Paul Blain Levy. Adjunction models for call-by-push-value with stacks. *Electronic Notes in Theoretical Computer Science*, 69:248–271, 2003. CTCS'02, Category Theory and Computer Science. doi:10.1016/S1571-0661(04)80568-1.

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

21 Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and computation*, 185(2):182–210, 2003.

22 Sam Lindley and Ian Stark. Reducibility and $\top\top$-lifting for computation types. In *Typed Lambda Calculi and Applications: 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005.*, pages 262–277. Springer, 2005.

23 J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–57. ACM, 1988. URL: http://doi.acm.org/10.1145/73560.73564, doi:10.1145/73560.73564.

24 Dylan McDermott and Tarmo Uustalu. What makes a strong monad? In *Proceedings Ninth Workshop on Mathematically Structured Functional Programming (to appear)*. Open Publishing Association, 2022.

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

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

27 Max S New, Daniel R Licata, and Amal Ahmed. Gradual type theory. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–31, 2019.

28 Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: A calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 123–135. ACM, 2014. URL: http://doi.acm.org/10.1145/2628136.2628160, doi:10.1145/2628136.2628160.

29 Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

30 A.L. Smirnov. Graded monads and rings of polynomials. *J. Math. Sci.*, 151(3):3032–3051, 2008. doi:10.1007/s10958-008-9013-7.

31 Andrew Tolmach. Optimizing ML using a hierarchy of monadic types. In Xavier Leroy and Atsushi Ohori, editors, *Types in Compilation*, pages 97–115. Springer, 1998.

32 Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. Effects and coeffects in call-by-push-value. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):1108–1134, 2024.

A The CBPVE equational theory

The axioms genering the CBPVE equational theory $\equiv$ are listed in Figure 5. Each axiom is subject to the evident typing constraints, for instance, $V \equiv ()$ requires V to have type $\mathbf{1}$. In addition to the axioms, we generate $\equiv$ by the obvious congruence rules, plus reflexivity, symmetry, and transitivity.

$$
\begin{aligned}
\mathbf{return}\, V\ \mathbf{to}\ x.\, M &\equiv M[x \mapsto V] \\
j`\lambda\{i.\, M_i\}_{i\in I} &\equiv M_j \\
V`(\lambda x : A.\, M) &\equiv M[x \mapsto V] \\
\mathbf{force}\,(\mathbf{thunk}\, M) &\equiv M \\
\mathbf{match}\,(V_1, V_2)\,\mathbf{with}\,(x_1, x_2)\,\mathbf{in}\, M &\equiv M[x_1 \mapsto V_1, x_2 \mapsto V_2] \\
\mathbf{match}\,\mathbf{inj}_j\, V\,\mathbf{with}\,\{\mathbf{inj}_i\, x_i.\, M_i\}_{i\in I} &\equiv M_j[x_j \mapsto V]
\end{aligned}
$$

(a) The β-laws.

$$
\begin{aligned}
V &\equiv \mathbf{thunk}\,(\mathbf{force}\, V) \\
V &\equiv () \\
M[x \mapsto V] &\equiv \mathbf{match}\, V\,\mathbf{with}\,(x_1, x_2)\,\mathbf{in}\, M[x \mapsto (x_1, x_2)] \\
M[x \mapsto V] &\equiv \mathbf{match}\, V\,\mathbf{with}\,\{\mathbf{inj}_i\, x_i.\, M[x \mapsto \mathbf{inj}_i\, x_i]\}_{i\in I} \\
M &\equiv M\ \mathbf{to}\ x.\, \mathbf{return}\, x \\
M &\equiv \lambda\{i.\, i`M\}_i \\
M &\equiv \lambda x : A.\, x`M
\end{aligned}
$$

(b) The η-laws. The second law only applies when V has type $\mathbf{1}$.

$$
\begin{aligned}
(M_1\ \mathbf{to}\ x.\, M_2)\ \mathbf{to}\ y.\, M_3 &\equiv M_1\ \mathbf{to}\ x.\, (M_2\ \mathbf{to}\ y.\, M_3) \\
(\mathbf{do}\, x \leftarrow \mathsf{op}\, V\,\mathbf{then}\, M)\ \mathbf{to}\ y.\, M' &\equiv \mathbf{do}\, x \leftarrow \mathsf{op}\, V\,\mathbf{then}\,(M\ \mathbf{to}\ y.\, M') \\
\lambda\{i.\, M\ \mathbf{to}\ x.\, N_i\}_{i\in I} &\equiv M\ \mathbf{to}\ x.\, \lambda\{i.\, N_i\}_{i\in I} \\
\lambda y : A.\, M\ \mathbf{to}\ x.\, N &\equiv M\ \mathbf{to}\ x.\, \lambda y : A.\, N
\end{aligned}
$$

(c) The sequencing laws.

$$
\begin{aligned}
M &\equiv \mathbf{coerce}_{\underline{C}}\, M \\
\mathbf{coerce}_{\underline{D}}\,(\mathbf{coerce}_{\underline{C}}\, M) &\equiv \mathbf{coerce}_{\underline{D}}\, M \\
(\mathbf{coerce}_{\mathbf{F}_e A}\, M)\ \mathbf{to}\ x.\,(\mathbf{coerce}_{\underline{D}}\, N) &\equiv \mathbf{coerce}_{\langle\!\langle e \rangle\!\rangle \underline{D}}\,(M\ \mathbf{to}\ x.\, N) \\
\mathbf{do}\, y \leftarrow \mathsf{op}\, V\,\mathbf{then}\,(\mathbf{coerce}_{\underline{D}}\, N) &\equiv \mathbf{coerce}_{\langle\!\langle d \rangle\!\rangle \underline{D}}\,(\mathbf{do}\, y \leftarrow \mathsf{op}\, V\,\mathbf{then}\, N) \\
i`(\mathbf{coerce}_{\prod_{i\in I} \underline{D}_i}\, M) &\equiv \mathbf{coerce}_{\underline{D}_i}\,(i`M) \\
V`(\mathbf{coerce}_{B \to \underline{D}}\, M) &\equiv \mathbf{coerce}_{\underline{D}}\,(V`M) \\
\mathbf{match}\, V\,\mathbf{with}\,(x_1, x_2)\,\mathbf{in}\,(\mathbf{coerce}_{\underline{D}}\, M) &\equiv \mathbf{coerce}_{\underline{D}}\,(\mathbf{match}\, V\,\mathbf{with}\,(x_1, x_2)\,\mathbf{in}\, M) \\
\mathbf{match}\, V\,\mathbf{with}\,\{\mathbf{inj}_i\, x_i.\,(\mathbf{coerce}_{\underline{D}} M_i)\}_{i\in I} &\equiv \mathbf{coerce}_{\underline{D}}\,(\mathbf{match}\, V\,\mathbf{with}\,\{\mathbf{inj}_i\, x_i.M_i\}_{i\in I})
\end{aligned}
$$

(d) The coercion laws. The first two laws are for reflexivity and transitivity; the first only applies when M has type $\underline{C}$.

Figure 5 (Typed) equations between CBPVE values and computations.