Effects and Coeffects in Call-by-Push-Value

CASSIA TORCZON, University of Pennsylvania, USA
EMMANUEL SUÁREZ ACEVEDO, University of Pennsylvania, USA
SHUBH AGRAWAL, University of Michigan, USA
JOEY VELEZ-GINORIO, University of Pennsylvania, USA
STEPHANIE WEIRICH, University of Pennsylvania, USA

Effect and coeffect tracking integrate many types of compile-time analysis, such as cost, liveness, or dataflow, directly into a language's type system. In this paper, we investigate the addition of effect and coeffect tracking to the type system of call-by-push-value (CBPV), a computational model useful in compilation for its isolation of effects and for its ability to cleanly express both call-by-name and call-by-value computations. Our main result is *effect-and-coeffect soundness*, which asserts that the type system accurately bounds the effects that the program may trigger during execution and accurately tracks the demands that the program may make on its environment. This result holds for two different dynamic semantics: a generic one that can be adapted for different coeffects and one that is adapted for reasoning about resource usage. In particular, the second semantics discards the evaluation of unused values and pure computations while ensuring that effectful computations are always evaluated, even if their results are not required. Our results have been mechanized using the Coq proof assistant.

CCS Concepts: • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: Types, CBPV, Effects, Coeffects

ACM Reference Format:
Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. 2024. Effects and Coeffects in Call-by-Push-Value. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 310 (October 2024), 27 pages. https://doi.org/10.1145/3689750

1 Introduction

Computations interact with the world in which they run. Sometimes the computation does something the world can observe, known as an *effect* [Lucassen and Gifford 1988], and sometimes computations demand something that the world must provide, known as a *coeffect* [Brunel et al. 2014; Orchard and Eades III 2022; Petricek et al. 2014]. For example, running a computation might take time (a clock ticking is an effect) and might require resources (using input parameters is a coeffect).

Some programming languages track effects and coeffects statically. Frank [Convent et al. 2020], Koka [Leijen 2014], and the Verse functional logic language [Verse development team 2023] do this for effects such as state, exceptions, divergence, and failure; Linear Haskell [Bernardy et al. 2017] does this for a resource management coeffect, while Agda and Idris 2 [Brady 2021] do this for a relevancy coeffect. The Effekt language [Brachthäuser et al. 2022] both tracks effects statically and

Authors' Contact Information: Cassia Torczon, University of Pennsylvania, Philadelphia, USA, ctorczon@seas.upenn.edu; Emmanuel Suárez Acevedo, University of Pennsylvania, Philadelphia, USA, emsu@seas.upenn.edu; Shubh Agrawal, University of Michigan, Ann Arbor, USA, shbhgrwl@umich.edu; Joey Velez-Ginorio, University of Pennsylvania, Philadelphia, USA, joeyv@seas.upenn.edu; Stephanie Weirich, University of Pennsylvania, Philadelphia, USA, sweirich@seas.upenn.edu.

ACM 2475-1421/2024/10-ART310
https://doi.org/10.1145/3689750

uses a limited form of coeffect tracking to ensure that effect handlers are well-scoped. Finally, the Granule language [Orchard et al. 2019] uses monads and comonads graded by abstract structures to track various effects and coeffects in a flexible and expressive system.

We would like to update the type systems of existing languages with effect and coeffect tracking by annotating their existing type systems. However, in contrast to systems that use monads and comonads to isolate effectful and coeffectful code from the rest of the language, we need an approach that is descriptive and that does not restrict programmers in how they structure their code.

Because effectful computation depends on evaluation order, precisely tracking effects works best in a language that makes its "ambient monad" explicit, such as Moggi's computational lambda calculus [Moggi 1989] and fine-grained CBV [Levy et al. 2003]. These systems separate inert "values" from executable "computations" and include "return" and "let" constructs to sequence evaluation. This "ambient monad" is part of the structure of the language itself; all computations are monadic.

Levy's Call-By-Push-Value (CBPV) [Levy 2003b] is a calculus that makes both the ambient computational monad *and comonad* explicit. As above, it separates values from computations and uses "return" and "let" constructs to track how computations manipulate values. However, CBPV also includes thunks, which temporarily suspend computations and treat them as values, for the opposite purpose; as a result all computations are also comonadic. In CBPV, then, we can annotate these existing structures directly to track effects and coeffects, instead of adding new features to the language.

CBPV is a low-level language and is appropriate for use as a compiler intermediate representation [Garbuzov et al. 2018; Rizkallah et al. 2018]. Its distinction between values and computations allows CBPV to work with strict and nonstrict language features explicitly, enabling it to model both call-by-value and call-by-name languages with the same facility. Adding effects and coeffects to CBPV would enrich this intermediate representation to support program optimizations; for example, to justify dead code elimination for pure code whose coeffect annotations mark it as unused.

The ability of CBPV to model both CBV and CBN also lets us observe how evaluation order changes the way a program alters and makes demands on the world. Levy characterizes the difference between values and computations with the slogan: "a value *is*, a computation *does*." [Levy 2003b] Our interpretation of this slogan is that only computations may contain effectful subcomponents—values must be pure throughout. Conversely, coeffects describe the demands a program makes on its inputs, which are always values in CBPV.

CBPV uses separate types for values and computations. Values have *positive* types (for which we use the metavariable A), while computations have *negative* types (for which we use B). These two forms are connected via an adjunction: the thunk type $\mathbf{U}\,B$ suspends a computation as an inert value, and the type of return $\mathbf{F}\,A$ creates a fine-grained structure similar to monadic bind that threads values through computations. Due to the structure of the adjunction, the combination $\mathbf{U}\,(\mathbf{F}\,A)$ forms a monad and the combination $\mathbf{F}\,(\mathbf{U}\,B)$ forms a comonad [Levy 2003a].

The duality between values and computations gives CBPV its power, and it is reflected in the structures we use to statically track effects and coeffects. For effects, we add effect information ϕ to the thunk type $\mathbf{U}_\phi\ B$, recording the latent effect of suspended computations. Similarly, to track coeffects, we add coeffect information q to the returner type $\mathbf{F}_q\ A$, describing the demands subsequent computation is allowed to make on the returned value. With this augmentation, we will show that the types $\mathbf{U}_\phi\ (\mathbf{F}\,A)$ and $\mathbf{F}_q\ (\mathbf{U}\,B)$ can encode the graded monads and comonads associated with effect and coeffect tracking.

Following this duality, this paper begins with two mirrored halves and then combines them. The first part (Section 2) extends CBPV with effect tracking and shows how we can recover the graded monad by grading the thunk type with latent effects. The second part (Section 3) extends

CBPV with coeffect tracking and recovers a graded comonad by grading the returner type with latent coeffects; we also discuss modifications to the system for resource tracking with coeffects (Section 4). Finally, we combine the two systems and explore their interaction (Section 5). This paper is best read in color: effects ϕ appear in red and coeffects q in blue. Without these colorful annotations, the type system and semantics are the standard rules of CBPV.

Along the way, we prove the following results about our extensions.

- We prove *effect soundness* for our effect-annotated extension of CBPV, demonstrating that the type-and-effect system accurately bounds what happens at runtime. To do so, we define an environment-based big-step operational semantics for CBPV instrumented to precisely track effects during evaluation, and we use a logical relation to prove our soundness theorem. (Section 2.3)
- We prove that the standard translations from call-by-value (CBV) and call-by-name (CBN) lambda calculi to CBPV are *type-and-effect preserving*. Starting with a well-typed CBV or (monadic) CBN program, we can produce a well-typed CBPV program with the same effects as the source program. (Section 2.4)
- We prove *coeffect soundness* for a coeffect-annotated extension of CBPV, demonstrating that the type-and-coeffect system accurately tracks the demands a program may make on its inputs. We do so using an environment-based big-step operational semantics for CBPV, where the environment has been instrumented to track coeffects during evaluation. (Section 3.1)
- We observe that our generic coeffect-tracking operational semantics behavior has strange implications when reasoning about resource usage. Therefore, we adapt the rules of our semantics so that it does not demand resources for discarded values, providing a better model of how the program uses its inputs in this coeffect. (Section 4)
- We prove that the standard translations from both CBN and CBV to CBPV are *type-and-coeffect* preserving for this updated coeffect system. Starting with a well-typed CBN or CBV program, we can produce a well-typed CBPV program with the same coeffects. (Section 4.1)
- We combine the 'tick' effect and resource tracking coeffect together into the same CBPV type system and prove combined versions of the results from each: *type-and-effect-and-coeffect soundness* and *type-and-effect-and-coeffect preservation* of the standard translations from CBV and CBN. We extend this system with a new rule that does not demand resources for unused *computations*, when they are effect-free. Finally, we prove that our discarding semantics produces the same result and has the same effects as our general semantics, justifying the soundness of our resource accounting semantics. (Section 5)

We are not the first to extend CBPV with effect tracking and our type system is most similar to Kammar and Plotkin [2012] and Forster et al. [2017]. However, all other definitions and results of this paper are novel. In particular, we have found little work that explores the interaction between CBPV and coeffects. Furthermore, while we are able use the standard translations to interpret CBV and CBN in CBPV, designing the effect and coeffect systems so that these translations "just work" is a contribution of this paper. Our approach to effect-and-coeffect soundness also differs from prior work—we employ a novel environment-based big-step semantics for CBPV that leads to short and straightforward proofs.

For simplicity, the effect systems in this paper only track clock effects, and the coeffect systems only count variable usages. As a result, we do not explore more sophisticated interactions between other forms of effects and coeffects, such as local and global state [Nanevski 2003], or between information flow and nondeterminism, or between usage analysis and errors [Gaboardi et al. 2016].

The results of this paper have been formalized in Coq and are available online[1] and archived on Zenodo [Torczon et al. 2024b]. This document includes hyperlinks that connect each definition and theorem to the appropriate source file in the mechanized proofs. For space, some parts of our mechanization have been elided from this paper, but full details are available in an extended version [Torczon et al. 2024a].

2 Call-by-Push-Value (CBPV) and Effect Tracking

In this section, we extend the type system of CBPV with effect tracking. Our modifications to the base system, which are limited to reasoning about effect annotations ϕ, are marked in red.

CBPV syntactically separates terms into *values* V, inhabiting positive types A, and *computations* M, inhabiting negative types B, as shown by the following grammar.

$$
\begin{array}{llll}
\textit{value types} & A & ::= & \mathtt{unit} \mid \mathbf{U}_\phi\ B \mid A_1 \times A_2 \\
\textit{computation types} & B & ::= & A \to B \mid \mathbf{F}\ A \mid B_1 \,\&\, B_2 \\
\\
\textit{values} & V & ::= & x \mid \{M\}\ () \mid \mid (V_1, V_2) \\
\textit{computations} & M & ::= & \lambda x.M \mid M\ V \mid V! \mid \mathbf{let}\ (x_1, x_2) = V\ \mathbf{in}\ N \\
 & & \mid & \langle M_1, M_2 \rangle \mid M.1 \mid M.2 \mid \mathbf{return}\ V \mid x \leftarrow M\ \mathbf{in}\ N \mid \mathbf{tick}
\end{array}
$$

Values in CBPV mostly correspond to the values found in a call-by-value typed functional language, such as unit and positive products of values. CBPV values also include suspended computations, called *thunks* and written $\{M\}$. (Variables always represent values, so they are always declared with value types in the context.)

Computations in CBPV include abstractions ($\lambda x.M$), applications ($M\ V$), elimination (*forcing*) of thunks ($V!$), and positive product elimination ($\mathbf{let}\ (x_1, x_2) = V\ \mathbf{in}\ N$). In addition to positive products, CBPV also includes negative products, of type $B_1 \,\&\, B_2$. These are introduced with a pair of computations $\langle M_1, M_2 \rangle$ and eliminated by projecting either the first or second component, *i.e.* $M.1$ or $M.2$.

Values can be threaded through computations. The $\mathbf{return}\ V$ form injects a value into a trivial computation. In the "letin" construct, written $x \leftarrow M\ \mathbf{in}\ N$, the first subcomputation must evaluate to the form $\mathbf{return}\ V$, and the second computation can then reference V. An advantage of CBPV is that this bind-like method of threading values through computations makes it readily extensible with effectful language features. Levy [2003b, 2006, 2022] demonstrates how to add nontermination, nondeterminism, errors, I/O, state, and control effects to CBPV. In each case, Levy extends the language with new computations and modifies the operational semantics to account for the new features.

For simplicity, we describe a single effect in this paper, the **tick** computation. This effect advances a virtual clock in the operational semantics, simulating the cost of the program.

2.1 CBPV: Type-and-Effect System

Our type-and-effect system for CBPV is shown in Figure 1. Under some typing context Γ, this system assigns a value type to values ($\Gamma \vdash_{\mathit{eff}} V : A$) and both a computation type and effect to computations ($\Gamma \vdash_{\mathit{eff}} M :^{\phi} B$), where ϕ is an upper bound on the effects that could occur during the evaluation of M. The judgement for values does not need an effect annotation because values are pure. In rule EFF-THUNK, the thunk type $\mathbf{U}_\phi\ B$ records the latent effect of a suspended computation.

Following Katsumata [2014], our system models effects using an arbitrary *preordered monoid*. This gives us an identity element ε, an associative combining operation $\phi_1 \cdot \phi_2$, and a preorder

[1] https://github.com/plclub/cbpv-effects-coeffects/

$\boxed{\Gamma \vdash_{\mathit{eff}} V : A}$ *(value effect typing)*

$$\textsc{eff-var}\ \frac{x : A \in \Gamma}{\Gamma \vdash_{\mathit{eff}} x : A} \qquad \textsc{eff-thunk}\ \frac{\Gamma \vdash_{\mathit{eff}} M :^{\phi} B}{\Gamma \vdash_{\mathit{eff}} \{M\} : \mathbf{U}_{\phi} B} \qquad \textsc{eff-unit}\ \frac{}{\Gamma \vdash_{\mathit{eff}} () : \mathbf{unit}}$$

$$\textsc{eff-pair}\ \frac{\Gamma \vdash_{\mathit{eff}} V_1 : A_1 \quad \Gamma \vdash_{\mathit{eff}} V_2 : A_2}{\Gamma \vdash_{\mathit{eff}} (V_1, V_2) : A_1 \times A_2}$$

$\boxed{\Gamma \vdash_{\mathit{eff}} M :^{\phi} B}$ *(computation effect typing)*

$$\textsc{eff-abs}\ \frac{\Gamma, x : A \vdash_{\mathit{eff}} M :^{\phi} B}{\Gamma \vdash_{\mathit{eff}} \lambda x.M :^{\phi} A \to B} \qquad \textsc{eff-app}\ \frac{\Gamma \vdash_{\mathit{eff}} M :^{\phi} A \to B \quad \Gamma \vdash_{\mathit{eff}} V : A}{\Gamma \vdash_{\mathit{eff}} M\, V :^{\phi} B} \qquad \textsc{eff-force}\ \frac{\Gamma \vdash_{\mathit{eff}} V : \mathbf{U}_{\phi} B}{\Gamma \vdash_{\mathit{eff}} V! :^{\phi} B}$$

$$\textsc{eff-ret}\ \frac{\Gamma \vdash_{\mathit{eff}} V : A}{\Gamma \vdash_{\mathit{eff}} \mathbf{return}\, V :^{\varepsilon} \mathbf{F}\, A} \qquad \textsc{eff-letin}\ \frac{\Gamma \vdash_{\mathit{eff}} M :^{\phi_1} \mathbf{F}\, A \quad \Gamma, x : A \vdash_{\mathit{eff}} N :^{\phi_2} B}{\Gamma \vdash_{\mathit{eff}} x \leftarrow M\, \mathbf{in}\, N :^{\phi_1 \cdot \phi_2} B}$$

$$\textsc{eff-splitt}\ \frac{\Gamma \vdash_{\mathit{eff}} V : A_1 \times A_2 \quad \Gamma, x_1 : A_1, x_2 : A_2 \vdash_{\mathit{eff}} N :^{\phi} B}{\Gamma \vdash_{\mathit{eff}} \mathbf{let}\, (x_1, x_2) = V\, \mathbf{in}\, N :^{\phi} B}$$

$$\textsc{eff-cpair}\ \frac{\Gamma \vdash_{\mathit{eff}} M_1 :^{\phi} B_1 \quad \Gamma \vdash_{\mathit{eff}} M_2 :^{\phi} B_2}{\Gamma \vdash_{\mathit{eff}} \langle M_1, M_2 \rangle :^{\phi} B_1 \,\&\, B_2} \qquad \textsc{eff-fst}\ \frac{\Gamma \vdash_{\mathit{eff}} M :^{\phi} B_1 \,\&\, B_2}{\Gamma \vdash_{\mathit{eff}} M.1 :^{\phi} B_1} \qquad \textsc{eff-snd}\ \frac{\Gamma \vdash_{\mathit{eff}} M :^{\phi} B_1 \,\&\, B_2}{\Gamma \vdash_{\mathit{eff}} M.2 :^{\phi} B_2}$$

$$\textsc{eff-tick}\ \frac{}{\Gamma \vdash_{\mathit{eff}} \mathbf{tick} :^{\mathrm{Tick}} \mathbf{F}\, \mathbf{unit}} \qquad \textsc{eff-sub}\ \frac{\Gamma \vdash_{\mathit{eff}} M :^{\phi_1} B \quad \phi_1 \leq_{\mathit{eff}} \phi_2}{\Gamma \vdash_{\mathit{eff}} M :^{\phi_2} B}$$

Fig. 1. CBPV typing and effect tracking

relation $\leq_{\mathit{eff}}$ that respects the operation. We also include a primitive effect Tick produced by the **tick** computation. However, the only parts of the system that are specific to this effect are the rules for **tick**, which is our *only* effectful computation. All other rules are presented generically and are adaptable to other effects and effectful computations (e.g. a Read effect produced by a **read** computation).

Concretely, we could use the natural number monoid with the usual ordering, 0 as the identity element ε, and addition as the combining operation to have our type system perform a cost analysis. Using 1 as our model of the Tick effect, the system would statically bound the number of **tick**s that are evaluated. For example, the type system would tell us that the computation $\langle \mathbf{tick}, y \leftarrow \mathbf{tick}\, \mathbf{in}\, \mathbf{tick} \rangle$ advances the clock at most twice. If the first component of the pair is projected, the type system overapproximates the effect produced during execution. Note that to track other behaviors with our type system, we need only change our preordered monoid accordingly (e.g. we could track possible effects with the power set monoid ordered by set inclusion).

Rules EFF-RET and EFF-LETIN motivate the choice of a monoid structure. Returning a value has no effect, so the effect of **return** V should always be ε. Rule EFF-LETIN must combine effects because $x \leftarrow M\, \mathbf{in}\, N$ is the only computation in our system with two subcomputations, both of which may

$\boxed{\rho \vdash V \Downarrow W}$ *(Value closing)*

$$\textsc{eval-val-var}\ \frac{x \mapsto W \in \rho}{\rho \vdash x \Downarrow W} \qquad \textsc{eval-val-unit}\ \frac{}{\rho \vdash () \Downarrow ()} \qquad \textsc{eval-val-thunk}\ \frac{}{\rho \vdash \{M\} \Downarrow \mathbf{clo}(\rho, \{M\})}$$

$$\textsc{eval-val-vpair}\ \frac{\rho \vdash V_1 \Downarrow W_1 \quad \rho \vdash V_2 \Downarrow W_2}{\rho \vdash (V_1, V_2) \Downarrow (W_1, W_2)}$$

Fig. 2. Operational semantics of CBPV with effect tracking

be effectful. Finally, because return and letin satisfy identity and associativity properties as the building blocks of the CBPV monad, we need these same properties in our effect structure.

Rule EFF-SUB allows for imprecision in the type system. That is, an effect annotation ϕ on the type of a program indicates that the program will have *at most* ϕ as its effect; it may have less. If the type system determines that the computation will complete within 5 ticks, it is also sound, but less precise, for it to say that it will complete within 7 ticks. Choosing the discrete ordering (i.e. using equality for $\leq_{\mathit{eff}}$) forces the type system to track effects precisely. Note that to allow the discrete ordering, we do not assume $\varepsilon \leq_{\mathit{eff}} \phi$ from the effect structure. In other words, the type system does not need ε to be the least effect, only an identity element for the combining operation.

This imprecision allows more programs to type check. In a program with branching, different branches may have different effects. For example, in rule EFF-CPAIR, only one side of a computational pair will ever be evaluated. However, for soundness, both computations must be typed with the same effect (which may be an overapproximation due to subeffecting).

Unlike in effect systems for the λ-calculus, the latent effects of function bodies are not recorded in function types. Instead, they are propagated to the conclusion of rule EFF-ABS. This makes sense because abstractions are not values in CBPV. From an operational sense, they are computations that pop the argument off the stack before continuing [Levy 2003b].

2.2 Instrumented Operational Semantics and Effect Soundness

We next define a big-step, *environment-based* operational semantics for CBPV. Here, an environment, ρ, is a mapping from variables to *closed values*, W, and can be thought of as a sequence of delayed substitutions. Closed values include closures, *i.e.* suspended computations paired with closing environments, as well as unit and positive products.

$$\begin{array}{llll} \textit{environments} & \rho & ::= & \emptyset \mid \rho, x \mapsto W \\ \textit{closed values} & W & ::= & () \mid \mathbf{clo}(\rho, \{M\}) \mid (W_1, W_2) \end{array}$$

This semantics is new but straightforward. Past presentations of CBPV define its operational behavior using small-step, big-step, or stack-based semantics, but all the ones we have found use immediate substitution [Levy 2022]. We choose an environment-based big-step semantics for two reasons. First, the big-step structure corresponds closely to the structure of the type system; there is only one rule of the operational semantics for each rule of the type system. Together with the use of environments, this semantics eliminates the need for substitution lemmas, leading to a remarkably straightforward soundness proof (Section 2.3). Second, the environment lets us track the demands that computations make on their inputs in our coeffect soundness proof (Section 3.1). For example, with resource usage, we can include annotations in the environment that count how many times the program accesses each variable during computation, mirroring the annotations in the context in the type system. A substitution-based semantics does not support this instrumentation.

$\boxed{\rho \vdash_{\mathit{eff}} M \Downarrow T \# \phi}$ *(Computation rules)*

EVAL-EFF-COMP-ABS

$$\frac{}{\rho \vdash_{\mathit{eff}} \lambda x.M \Downarrow \mathbf{clo}(\rho, \lambda x.M) \# \varepsilon}$$

EVAL-EFF-COMP-APP-ABS

$$\frac{\rho \vdash_{\mathit{eff}} M \Downarrow \mathbf{clo}(\rho', \lambda x.M') \# \phi_1 \qquad \rho \vdash V \Downarrow W \qquad \rho', x \mapsto W \vdash_{\mathit{eff}} M' \Downarrow T \# \phi_2}{\rho \vdash_{\mathit{eff}} M\, V \Downarrow T \# \phi_1 \cdot \phi_2}$$

EVAL-EFF-COMP-FORCE-THUNK

$$\frac{\rho \vdash V \Downarrow \mathbf{clo}(\rho', \{M\}) \qquad \rho' \vdash_{\mathit{eff}} M \Downarrow T \# \phi}{\rho \vdash_{\mathit{eff}} V! \Downarrow T \# \phi}$$

EVAL-EFF-COMP-RETURN

$$\frac{\rho \vdash V \Downarrow W}{\rho \vdash_{\mathit{eff}} \mathbf{return}\, V \Downarrow \mathbf{return}\, W \# \varepsilon}$$

EVAL-EFF-COMP-LETIN-RET

$$\frac{\rho \vdash_{\mathit{eff}} M \Downarrow \mathbf{return}\, W \# \phi_1 \qquad \rho, x \mapsto W \vdash_{\mathit{eff}} N \Downarrow T \# \phi_2}{\rho \vdash_{\mathit{eff}} x \leftarrow M\, \mathbf{in}\, N \Downarrow T \# \phi_1 \cdot \phi_2}$$

EVAL-EFF-COMP-SPLIT

$$\frac{\rho \vdash V \Downarrow (W_1, W_2) \qquad \rho, x_1 \mapsto W_1, x_2 \mapsto W_2 \vdash_{\mathit{eff}} N \Downarrow T \# \phi}{\rho \vdash_{\mathit{eff}} \mathbf{let}\, (x_1, x_2) = V\, \mathbf{in}\, N \Downarrow T \# \phi}$$

EVAL-EFF-COMP-CPAIR

$$\frac{}{\rho \vdash_{\mathit{eff}} \langle M_1, M_2 \rangle \Downarrow \mathbf{clo}(\rho, \langle M_1, M_2 \rangle) \# \varepsilon}$$

EVAL-EFF-COMP-FST

$$\frac{\rho \vdash_{\mathit{eff}} M \Downarrow \mathbf{clo}(\rho', \langle M_1, M_2 \rangle) \# \phi_1 \qquad \rho' \vdash_{\mathit{eff}} M_1 \Downarrow T \# \phi_2}{\rho \vdash_{\mathit{eff}} M.1 \Downarrow T \# \phi_1 \cdot \phi_2}$$

EVAL-EFF-COMP-SND

$$\frac{\rho \vdash_{\mathit{eff}} M \Downarrow \mathbf{clo}(\rho', \langle M_1, M_2 \rangle) \# \phi_1 \qquad \rho' \vdash_{\mathit{eff}} M_2 \Downarrow T \# \phi_2}{\rho \vdash_{\mathit{eff}} M.2 \Downarrow T \# \phi_1 \cdot \phi_2}$$

Fig. 3. Operational semantics of CBPV with effect tracking

Figure 3 shows the definition of the operational semantics. This semantics consists of two relations. The first relation, written $\rho \vdash V \Downarrow W$, uses the provided environment ρ to "evaluate" a value V to a closed value W. This operation is essentially a substitution operation in that it replaces each variable found in the value with its definition in the environment.

The second relation, written $\rho \vdash_{\mathit{eff}} M \Downarrow T \# \phi$, shows how computations evaluate to *closed terminal computations*, T. Closed terminals are computations that cannot step any further, such as returned (closed) values and suspended abstractions and pairs. The effect annotation ϕ on this relation counts the number of ticks that occur during evaluation of M. While suspended abstractions and pairs resemble closures, they are not first class. Instead, they provide a convenient notation describing the propagation of the environment during evaluation.

$$\textit{closed terminals} \quad T \quad ::= \quad \mathbf{return}\, W \mid \mathbf{clo}(\rho, \lambda x.M) \mid \mathbf{clo}(\rho, \langle M_1, M_2 \rangle)$$

The operational semantics of the **tick** computation is trivial—it merely produces a unit value and a single **Tick** effect. Other computations either produce no effect (as in rule EVAL-EFF-COMP-ABS) or combine the effects of their subcomponents (as in rule EVAL-EFF-COMP-APP-ABS). As in the type-and-effect system, the only rule that is specific to the **Tick** effect is the rule for **tick**. All other effects in these rules are parameterized over the input monoid.

While the type system allows for imprecision, the operational semantics precisely tracks the effects of computation.

2.3 Type-and-Effect Soundness

We state our effect soundness theorem as follows: closed, well-typed computations of type $\mathbf{F}\,A$ return closed values and produce effects that are bounded by the type system.

Theorem 2.1 (Effect soundness). *If* $\varnothing \vdash_{\mathit{eff}} M \ :^{\phi} \ \mathbf{F}\,A$ *then* $\emptyset \vdash_{\mathit{eff}} M \Downarrow \mathbf{return}\ W \,\#\, \phi_1$ *where* $\phi_1 \leq_{\mathit{eff}} \phi$.

The proof is simple and based on the following logical relation, consisting of three functions defined mutually over the structure of types: closed values $\mathcal{W}[\![A]\!]$, closed terminal computations $\mathcal{T}[\![B]\!]^{\phi}$, and computations tupled with environments $\mathcal{M}[\![B]\!]^{\phi}$.

Definition 2.2 (CBPV with Effects: Logical Relation).

$$
\begin{array}{lcll}
\mathcal{W}[\![\mathbf{U}_{\phi}\ B]\!] & = & \{\ \mathbf{clo}(\rho, \{M\}) & \mid (\rho, M) \in \mathcal{M}[\![B]\!]^{\phi}\ \} \\
\mathcal{W}[\![\mathbf{unit}]\!] & = & \{\ ()\ \} & \\
\mathcal{W}[\![A_1 \times A_2]\!] & = & \{\ (W_1, W_2) & \mid W_1 \in \mathcal{W}[\![A_1]\!] \textit{ and } W_2 \in \mathcal{W}[\![A_2]\!]\ \} \\
 & & & \\
\mathcal{T}[\![\mathbf{F}\,A]\!]^{\phi} & = & \{\ \mathbf{return}\ W & \mid W \in \mathcal{W}[\![A]\!] \textit{ and } \phi \equiv \varepsilon\ \} \\
\mathcal{T}[\![A \to B]\!]^{\phi} & = & \{\ \mathbf{clo}(\rho, \lambda x.M) & \mid \textit{for all } W \in \mathcal{W}[\![A]\!],\ ((\rho, x \mapsto W), M) \in \mathcal{M}[\![B]\!]^{\phi}\ \} \\
\mathcal{T}[\![B_1\ \&\ B_2]\!]^{\phi} & = & \{\ \mathbf{clo}(\rho, \langle M_1, M_2\rangle) & \mid (\rho, M_1) \in \mathcal{M}[\![B_1]\!]^{\phi} \textit{ and } (\rho, M_2) \in \mathcal{M}[\![B_2]\!]^{\phi}\ \} \\
 & & & \\
\mathcal{M}[\![B]\!]^{\phi} & = & \{\ (\rho, M) & \mid \rho \vdash_{\mathit{eff}} M \Downarrow T \,\#\, \phi_1 \textit{ and } T \in \mathcal{T}[\![B]\!]^{\phi_2} \textit{and } \phi_1 \cdot \phi_2 \leq_{\mathit{eff}} \phi\ \}
\end{array}
$$

We use this relation to define semantic typing for environments, values, and computations.

Definition 2.3 (CBPV with Effects: Semantic Typing).

$$
\begin{array}{lcl}
\Gamma \vDash \rho & = & x : A \in \Gamma \textit{ implies } x \mapsto W \in \rho \textit{ and } W \in \mathcal{W}[\![A]\!] \\
\Gamma \vDash_{\mathit{eff}} V : A & = & \Gamma \vDash \rho \textit{ implies } \rho \vdash V \Downarrow W \textit{ and } W \in \mathcal{W}[\![A]\!] \\
\Gamma \vDash_{\mathit{eff}} M \ :^{\phi} \ B & = & \Gamma \vDash \rho \textit{ implies } (\rho, M) \in \mathcal{M}[\![B]\!]^{\phi}
\end{array}
$$

Using these definitions, we can prove semantic typing lemmas corresponding to each of the syntactic typing rules shown in Figure 1. These proofs require our assumptions about the monoidal structure of effects: that ε is an identity element for the associative combining operation.

With these lemmas, we show the fundamental lemma as a straightforward induction.

Lemma 2.4 (Fundamental Lemma: effect soundness).

(1) *If* $\Gamma \vdash_{\mathit{eff}} V : A$ *then* $\Gamma \vDash_{\mathit{eff}} V : A$.

(2) *If* $\Gamma \vdash_{\mathit{eff}} M \ :^{\phi} \ B$ *then* $\Gamma \vDash_{\mathit{eff}} M \ :^{\phi} \ B$.

The effect soundness theorem (2.1) follows from the second clause of this lemma, after instantiating Γ with the empty context and B with $\mathbf{F}\,A$. Unfolding the definition of $\varnothing \vDash_{\mathit{eff}} M \ :^{\phi} \ \mathbf{F}\,A$ gives us some ϕ_1 and ϕ_2 such that $\emptyset \vdash_{\mathit{eff}} M \Downarrow T \,\#\, \phi_1$ and $T \in \mathcal{T}[\![\mathbf{F}\,A]\!]^{\phi_2}$ and $\phi_1 \cdot \phi_2 \leq_{\mathit{eff}} \phi$. Further unfolding definitions means that T must be $\mathbf{return}\ W$, ϕ_2 must be ε, and thus $\phi_1 \leq_{\mathit{eff}} \phi$.

2.4 Type-and-Effect Preserving Translations

Levy [2006] provides translations from call-by-value (CBV) and call-by-name (CBN) λ-calculi to CBPV and shows that those translations preserve types, denotational semantics, and (substitution-based) big-step operational semantics. We show here that those translations also preserve effects.

For the CBV translation, we start with a λ-calculus that has a simple type-and-effect system, loosely based on Lucassen and Gifford [1988]. However, as few CBN languages directly include effects, for the CBN translation we start with a simply-typed λ-calculus that encapsulates effects using a *graded monad*. Furthermore, we show that we can use this same monad with the CBV translation because effects are encapsulated.

2.4.1 CBV Type-and-Effect System. The simple CBV language with effect tracking in this subsection features the same **tick** term and **Tick** effect as before, along with the usual forms of the λ-calculus.[2]

$$\textsc{lam-eff-var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash_{\mathit{eff}} x :^{\varepsilon} \tau} \qquad \textsc{lam-eff-abs}\ \frac{\Gamma, x : \tau_1 \vdash_{\mathit{eff}} e :^{\phi} \tau_2}{\Gamma \vdash_{\mathit{eff}} \lambda x.e :^{\varepsilon} \tau_1 \xrightarrow{\phi} \tau_2}$$

$$\textsc{lam-eff-app}\ \frac{\Gamma \vdash_{\mathit{eff}} e_1 :^{\phi_1} \tau_1 \xrightarrow{\phi_3} \tau_2 \qquad \Gamma \vdash_{\mathit{eff}} e_2 :^{\phi_2} \tau_1}{\Gamma \vdash_{\mathit{eff}} e_1\ e_2 :^{\phi_1 \cdot \phi_2 \cdot \phi_3} \tau_2} \qquad \textsc{lam-eff-tick}\ \frac{}{\Gamma \vdash_{\mathit{eff}} \mathbf{tick} :^{\mathrm{Tick}} \mathbf{unit}}$$

Function types, written $\tau_1 \xrightarrow{\phi} \tau_2$, are annotated with *latent effects*, which occur when the function is called. In the application rule rule LAM-EFF-APP, this latent effect is combined with ϕ_1, the effects that occur when evaluating the function e_1 to a λ expression, and ϕ_2, the effects that occur when evaluating the argument to a value.

The CBV type and term translations follow directly from Levy [2022]. Besides adding a case for the **tick** expression, the only change that we make is moving the latent effect from the function type to the thunk type. All other cases are exactly as in prior work. Because of this, we only show the translation for function types and the **tick** expression.

Type translation

$$[\![\tau_1 \xrightarrow{\phi} \tau_2]\!]_{\mathrm{v}} = \mathbf{U}_{\phi}\ ([\![\tau_1]\!]_{\mathrm{v}} \to \mathbf{F}\,[\![\tau_2]\!]_{\mathrm{v}})$$

Term translation

$$[\![\mathbf{tick}]\!]_{\mathrm{v}} = \mathbf{tick}$$

This translation preserves types and effects from the source language.

Lemma 2.5 (CBV translation is type correct). *If* $\Gamma \vdash_{\mathit{eff}} e :^{\phi} \tau$ *then* $[\![\Gamma]\!]_v \vdash_{\mathit{eff}} [\![e]\!]_v :^{\phi} \mathbf{F}\,[\![\tau]\!]_v$.

This result is easy to prove, reassuring us that our effect system design is correct: we can use CBPV to encode the well-studied type-and-effect systems developed over the past 40 years.

2.4.2 Graded Monads. CBPV is designed to serve as a convenient translation target for both CBV and CBN languages. However, in CBN languages, effects are usually[3] tracked using parametric effect monads, also known as *graded monads* [Katsumata 2014; Orchard and Petricek 2014; Smirnov 2008; Wadler and Thiemann 2003]. Therefore, here we translate a CBN language with graded monads to CBPV. Our source language for this translation is the simply-typed λ-calculus with unit and products, together with a graded monadic type $\mathbf{T}_{\phi}\ \tau$, the monadic operations **return** and **bind**, and the **tick** operation, with a monadic type. To account for imprecision, we include an explicit type coercion, written **coerce** e for the graded monad.

$$\textsc{lam-mon-return}\ \frac{\Gamma \vdash_{\mathit{mon}} e : \tau}{\Gamma \vdash_{\mathit{mon}} \mathbf{return}\ e : \mathbf{T}_{\varepsilon}\ \tau} \qquad \textsc{lam-mon-bind}\ \frac{\Gamma \vdash_{\mathit{mon}} e_1 : \mathbf{T}_{\phi_1}\ \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\mathit{mon}} e_2 : \mathbf{T}_{\phi_2}\ \tau_2}{\Gamma \vdash_{\mathit{mon}} \mathbf{bind}\ x = e_1\ \mathbf{in}\ e_2 : \mathbf{T}_{\phi_1 \cdot \phi_2}\ \tau_2}$$

[2] For space, we elide the typing rules for unit and products. These rules are available in the extended version [Torczon et al. 2024a]. [3] Instead of graded monads, we could also consider a translation from call-by-name language that does not encapsulate effects, such as the one defined by McDermott and Mycroft [2018].

$$\textsc{lam-mon-tick}\ \frac{}{\Gamma \vdash_{mon} \mathbf{tick} : \mathbf{T}_{\mathrm{Tick}}\ \mathbf{unit}} \qquad \textsc{lam-mon-coerce}\ \frac{\Gamma \vdash_{mon} e : \mathbf{T}_{\phi_1}\ \tau \qquad \phi_1 \leq_{eff} \phi_2}{\Gamma \vdash_{mon} \mathbf{coerce}\ e : \mathbf{T}_{\phi_2}\ \tau}$$

Below, we extend Levy's translation of the CBN λ-calculus to include the graded monad. The translation of the core language is as in prior work and all effects are isolated to the monadic type, so we only show the monadic portion in the figure.

Type translation

$$[\![\mathbf{T}_{\phi}\ \tau]\!]_{\mathrm{N}} = \mathbf{F}\ (\mathbf{U}_{\phi}\ \mathbf{F}\ (\mathbf{U}_{\varepsilon}\ [\![\tau]\!]_{\mathrm{N}}))$$

Term translation

$$\begin{aligned}
[\![\mathbf{return}\ e]\!]_{\mathrm{N}} &= \mathbf{return}\ \{\mathbf{return}\ \{[\![e]\!]_{\mathrm{N}}\}\} \\
[\![\mathbf{bind}\ x = e_1\ \mathbf{in}\ e_2]\!]_{\mathrm{N}} &= \mathbf{return}\ \{x \leftarrow (y \leftarrow [\![e_1]\!]_{\mathrm{N}}\ \mathbf{in}\ y!)\ \mathbf{in}\ z \leftarrow [\![e_2]\!]_{\mathrm{N}}\ \mathbf{in}\ z!\} \\
[\![\mathbf{coerce}\ e]\!]_{\mathrm{N}} &= \mathbf{return}\ \{x \leftarrow [\![e]\!]_{\mathrm{N}}\ \mathbf{in}\ x!\} \\
[\![\mathbf{tick}]\!]_{\mathrm{N}} &= \mathbf{return}\ \{x \leftarrow \mathbf{tick}\ \mathbf{in}\ \mathbf{return}\ \{\mathbf{return}\ x\}\}
\end{aligned}$$

This translation preserves types (with embedded effects) from the source language. Note that, because the monadic type marks effectful code, the translation produces CBPV computations that can be checked with the "pure" effect ε.

Lemma 2.6 (CBN translation is type correct). *If* $\Gamma \vdash_{mon} e : \tau$ *then* $[\![\Gamma]\!]_N \vdash_{eff} [\![e]\!]_N :^{\varepsilon} [\![\tau]\!]_N$.

One difficulty of this translation is that the monadic type in the CBPV adjunction is **U F**. This type is a value type, and the standard CBN translation produces terms with computation types. Therefore. to use **U F** as the monad in our CBN translation, we need to bracket it: on the outside by **F** to form a computation type, and then on the inside by **U** to construct the value type that the monad expects. This bracketing produces an awkward translation of the monadic operations with doubled thunking. This awkwardness is due to the presence of the monad in the source language; it is a separate structure from the ambient monad of the computation language.

3 CBPV and Coeffects (Version 1: General Semantics)

Next, we construct a parallel extension of CBPV augmented with *coeffect* tracking. Figure 4 lists the typing rules, with coeffect annotations in blue. Coeffect systems are designed for reasoning about how programs use their inputs, so we annotate variables at their binding sites and in the context.

Coeffects annotations consist of *grades* q taken from a *preordered semiring*. This structure provides an addition operation $q_1 + q_2$, an additive identity element 0, a multiplication operation $q_1 \cdot q_2$, a multiplicative identity 1, and a reflexive and transitive binary relation $\leq_{co}$ that respects addition and multiplication. (The preorder does not have to be the one defined by the addition operation.) The need for a semiring rather than a monoid arises from the fact that any value may be bound to a variable that may then be used multiple times, requiring a notion of coeffect multiplication.

Similarly to the previous section, our type system in this section is general across coeffects and can be specialized via the choice of semiring and preorder. For example, if we are only concerned with relevance analysis (i.e. determining which of a functions inputs are relevant to computation) then we might use a semiring with two elements: 0 marks inputs that are known to be unused and 1 is for elements that may or may not be needed. Or, in the case of information flow, then we might use a semiring where 0 marks secret inputs and 1 marks public information; only the latter may influence the result of the computation.

We would also like to use coeffects to track resource usage. However, as we discuss in detail below, this general semantics does not provide a satisfying account of resource usage and requires

$\boxed{\gamma \cdot \Gamma \vdash_{coeff} V : A}$ *(value coeffect typing)*

$$\textsc{coeff-var}\ \frac{}{\overline{0} \cdot \Gamma_1, x :^{1} A, \overline{0} \cdot \Gamma_2 \vdash_{coeff} x : A} \qquad \textsc{coeff-thunk}\ \frac{\gamma \cdot \Gamma \vdash_{coeff} M : B}{\gamma \cdot \Gamma \vdash_{coeff} \{M\} : \mathbf{U}\, B} \qquad \textsc{coeff-unit}\ \frac{}{\overline{0} \cdot \Gamma \vdash_{coeff} () : \mathbf{unit}}$$

$$\textsc{coeff-pair}\ \frac{\gamma_1 \cdot \Gamma \vdash_{coeff} V_1 : A_1 \qquad \gamma_2 \cdot \Gamma \vdash_{coeff} V_2 : A_2}{\gamma_1 + \gamma_2 \cdot \Gamma \vdash_{coeff} (V_1, V_2) : A_1 \times A_2} \qquad \textsc{coeff-vsub}\ \frac{\gamma' \cdot \Gamma \vdash_{coeff} V : A \qquad \gamma \leq_{co} \gamma'}{\gamma \cdot \Gamma \vdash_{coeff} V : A}$$

$\boxed{\gamma \cdot \Gamma \vdash_{coeff} M : B}$ *(computation coeffect typing)*

$$\textsc{coeff-abs}\ \frac{\gamma \cdot \Gamma, x :^{q} A \vdash_{coeff} M : B \qquad q' \leq_{co} q}{\gamma \cdot \Gamma \vdash_{coeff} \lambda x^{q}.M : A^{q'} \to B} \qquad \textsc{coeff-app}\ \frac{\gamma_1 \cdot \Gamma \vdash_{coeff} M : A^{q} \to B \qquad \gamma_2 \cdot \Gamma \vdash_{coeff} V : A}{\gamma_1 + (q \cdot \gamma_2) \cdot \Gamma \vdash_{coeff} M\, V : B} \qquad \textsc{coeff-force}\ \frac{\gamma \cdot \Gamma \vdash_{coeff} V : \mathbf{U}\, B}{\gamma \cdot \Gamma \vdash_{coeff} V! : B}$$

$$\textsc{coeff-split}\ \frac{\gamma_1 \cdot \Gamma \vdash_{coeff} V : A_1 \times A_2 \qquad \gamma_2 \cdot \Gamma, x_1 :^{q} A_1, x_2 :^{q} A_2 \vdash_{coeff} N : B}{(q \cdot \gamma_1) + \gamma_2 \cdot \Gamma \vdash_{coeff} \mathbf{case}_{q}\ V\ \mathbf{of}\ (x_1, x_2)\ \to\ N : B} \qquad \textsc{coeff-ret}\ \frac{\gamma \cdot \Gamma \vdash_{coeff} V : A}{q \cdot \gamma \cdot \Gamma \vdash_{coeff} \mathbf{return}_{q}\ V : \mathbf{F}_{q}\, A}$$

$$\textsc{coeff-letin}\ \frac{\gamma_1 \cdot \Gamma \vdash_{coeff} M : \mathbf{F}_{q_1} A \qquad \gamma_2 \cdot \Gamma, x :^{q_1 \cdot q_2} A \vdash_{coeff} N : B}{(q_2 \cdot \gamma_1) + \gamma_2 \cdot \Gamma \vdash_{coeff} x \leftarrow^{q_2} M\ \mathbf{in}\ N : B} \qquad \textsc{coeff-cpair}\ \frac{\gamma \cdot \Gamma \vdash_{coeff} M_1 : B_1 \qquad \gamma \cdot \Gamma \vdash_{coeff} M_2 : B_2}{\gamma \cdot \Gamma \vdash_{coeff} \langle M_1, M_2 \rangle : B_1 \,\&\, B_2}$$

$$\textsc{coeff-fst}\ \frac{\gamma \cdot \Gamma \vdash_{coeff} M : B_1 \,\&\, B_2}{\gamma \cdot \Gamma \vdash_{coeff} M.1 : B_1} \qquad \textsc{coeff-snd}\ \frac{\gamma \cdot \Gamma \vdash_{coeff} M : B_1 \,\&\, B_2}{\gamma \cdot \Gamma \vdash_{coeff} M.2 : B_2} \qquad \textsc{coeff-csub}\ \frac{\gamma' \cdot \Gamma \vdash_{coeff} M : B \qquad \gamma \leq_{co} \gamma'}{\gamma \cdot \Gamma \vdash_{coeff} M : B}$$

Fig. 4. CBPV type system with coeffect tracking

further refinement in the next section. Therefore, we first describe the general semantics in terms of the resource usage coeffect, so that we can prepare for this discussion.

In the case of resource usage, grades bound the *uses* of variables, as in bounded linear logic, and come from the natural number semiring with the usual addition and multiplication operators. The additive and multiplicative identity elements of this semiring mark 0 and at most 1 (affine) use of a variable respectively, and the addition and multiplication semiring operations calculate the total number of times each variable is used in the program.

As in many systems for bounded linear logic, $q_1 \leq_{co} q_2$ indicates that q_1 is *less precise* or *less restrictive* than q_2. When counting variable usage, this has the opposite order from the usual one—we have $3 \leq_{co} 2$ because allowing at most 3 uses is less restrictive than at most 2. With other coeffects, such as security levels, this ordering has a more intuitive interpretation: a higher grade corresponds to a higher security level, which is more restrictive than a low security level.

Like the effect system with subeffecting, this type system includes a rule for *subcoeffecting*: if a judgment holds with some annotation q_2 on a variable in the context, then it is also derivable with

any $q_1 \leq_{co} q_2$. For example, we can weaken a judgment that a computation makes zero (0) uses of some variable to observe at most one use (affine) or any other number. This corresponds to the usual weakening lemma from typed λ-calculi.

Again, as in the effect section, including a preorder with the semiring allows for imprecision, needed when analyzing branching computations. For example, if one branch requires 1 use of a variable x, but the other branch requires 0 uses, the system will record that the program must have the resources to use x at least once, because $1 \leq_{co} 0$, in a semiring where 1 corresponds to affine usage. This relation is dual to the preorder's role in the effect system—if one branch ticks once and the other does not tick, then the system will record at most one tick. In both cases, replacing the ordering with the discrete preorder means that the type system must be precise and would reject both of these examples.

The type system uses a *grade vector* γ, a comma-separated list of grades, to represent the annotations for the variables in a typing context. When combined with a typing context Γ, written $\gamma \cdot \Gamma$, the grade vector must have the same length as Γ. We extend a combined grade vector and typing context simultaneously with the notation $\gamma \cdot \Gamma, x :^q A$, equivalent to $(\gamma, q) \cdot (\Gamma, x : A)$.

The grade vector written $\overline{0}$ contains only zeros and is used where its length can be inferred from context. Grade vectors of the same length can be added together pointwise, written $\gamma_1 + \gamma_2$, and compared pointwise, written $\gamma_1 \leq_{co} \gamma_2$. Grade vectors can also be pointwise scaled, written $q \cdot \gamma$.

The basis of this system is rule COEFF-VAR. When introducing a variable x, the context must grade x with 1, indicating that it is used once. No other variables in the context should affect the typing judgement, so they must have grade 0. Similarly, the unit value $()$ can make no demands on the environment, so rule COEFF-UNIT requires that all variables in the typing context be graded 0.

In rule COEFF-THUNK and rule COEFF-FORCE, there is a single subterm that makes exactly the same demands on its environment as the term in the conclusion, so we use the same grade vector in the conclusion and the premise.

In other rules, the term in the conclusion has multiple subterms, so we combine the demands made by each. In rule COEFF-PAIR, the subterms both get evaluated and do not directly interact, so we combine their grade vectors via simple pointwise addition. Conversely, with negative products, the two subterms must use the same resources, so we use the same grade vector in each premise and the conclusion. Intuitively, this is because we can only ever project out one subterm from a computation pair (see rule COEFF-FST and rule COEFF-SND), so the projected term will make all the same demands on the environment as the pair.

In rule COEFF-ABS, we know from the premise that M will require a grade of q on x, so we store that grade as an annotation on x in the term syntax. For flexibility, we allow the annotation in the type, q', to be a less precise approximation of q. (This expressiveness is useful for the translation results in the next section. Note that subcoeffecting is not sufficient as it cannot allow the annotation on the λ to differ from the annotation on the function type.) Both the premise and the conclusion make the same demands on the variables in Γ, so γ is otherwise the same in both.

In some rules, we must combine the grade vectors of subterms using both scaling and addition. For example, in rule COEFF-APP, γ_1 denotes the demands the operator M makes on the environment, and γ_2 denotes the demands the argument V makes. M has type $A^q \to B$, indicating that when it is reduced to some terminal $\lambda x^{q'}.M'$, then M' will require x to have a grade of q', where $q \leq_{co} q'$. This means we must scale γ_1 by q before adding it to γ_2 to calculate the total demands that $M\ V$ makes on its environment.

In the effect system, we annotate the type $\mathbf{U}_\phi\ B$ with the effect of the suspended computation. In the coeffect system, we dually annotate the returner type $\mathbf{F}_q\ A$. In our resource usage example, the q indicates that we require enough resources from the environment to produce q copies of a value.

For example, $\mathbf{return}_3\ V$ indicates that we require the resources to create 3 copies of V. Therefore, rule COEFF-RET scales the demands needed to create V by q.

In rule COEFF-LETIN, M has returner type $\mathbf{F}_{q_1}\ A$, and its result value has been scaled by q_1. However, the expression includes another scaling annotation q_2, that allows duplication of the computation M itself. If γ_1 denotes the demands M makes on its environment, $q_1 \cdot q_2$ denotes the grade N requires x to have, and γ_2 denotes the demands N makes from the rest of the environment, then we need $q_2 \cdot \gamma_1 + \gamma_2$ to type the entire term.

The scaling annotations in $\mathbf{return}_q\ V$ and $x \leftarrow^q M\ \mathbf{in}\ N$ increase the expressiveness of the language and are required for the translation of a CBV λ-calculus to CBPV described in Section 4.1.2. Because CBV is strict, when translating an application, we must use a let binding to evaluate the translated argument before applying the translated function to it. However, the function may require a particular grade q on its argument, so we must be able to scale this computation. Similarly, to translate the graded CBV comonadic type, we need to be able to duplicate values.

The two subsumption rules COEFF-VSUB and COEFF-CSUB allow for subcoeffecting.

3.1 General Instrumented Operational Semantics and Coeffect Soundness

Next, we develop an instrumented operational semantics, shown in Figure 5, that tracks coeffects using an environment ρ, which maps variables to closed values, and a grade vector γ of equal length, which implicitly maps variables to their coeffects. As before, we extend both a grade vector and corresponding environment simultaneously with the notation $\gamma \cdot \rho,\ x \mapsto^q W$, equivalent to $(\gamma, q) \cdot (\rho, x \mapsto W)$.

We also use W as a metavariable for *closed* values and T as a metavariable for *closed terminal* computations. However, closed terminals include coeffects here. They have the form $\mathbf{return}_q\, W$, $\mathbf{clo}(\gamma \cdot \rho, \lambda x^q.M)$, or $\mathbf{clo}(\gamma \cdot \rho, \langle M_1, M_2 \rangle)$, where $\mathbf{clo}(\gamma \cdot \rho, M)$ denotes the *closure* of M under $\gamma \cdot \rho$. The grade vector in the closure indicates the demands on the variables used by M.

Unlike our instrumented operational semantics for effects, which calculates the exact effect of a computation, this semantics cannot track coeffects with precision. For example, suppose we have a term $\lambda x^1.M$ where M is a computation that both branches on its argument and uses it in at exactly one branch, such as $\mathbf{case}_1\ x\ \mathbf{of}\ \mathbf{inl}\ x_1 \to\ \mathbf{return}\, x; \mathbf{inr}\ x_2 \to\ \mathbf{return}\, \mathbf{inr}\, ()$. [4] What should this step to? If provided with an argument of the form $\mathbf{inl}\, y$, it should step to $\mathbf{clo}(x \mapsto^1 \mathbf{inl}\, y, \lambda x^1.M)$. If provided with an argument of the form $\mathbf{inr}\, y$, it should step to $\mathbf{clo}(x \mapsto^0 \mathbf{inr}\, y, \lambda x^1.M)$. But, if this term is the entire program, it is not clear what it should step to. In general, depending on the argument, the body of a function $\lambda x^q.M$ may require a different exact grade on x; all we know from the typing judgement is that q must be a bound on that usage. We cannot write a precise rule for evaluating abstractions to their closed terminal forms, because we do not have access to the argument yet when doing that evaluation.

Therefore, as in the typing rules, the operational semantics also includes rules for subcoeffecting, rules EVAL-COEFF-VAL-VSUB and EVAL-COEFF-COMP-CSUB. These rules say that if we can step a term with grades given by γ attached to the environment, then we can step it with γ' for any $\gamma' \leq_{co} \gamma$, i.e., any less precise accounting.

As in the semantics for CBPV without coeffects, we define "evaluation" of values using the given environment (see Figure 5). These rules mirror the typing rules: rule EVAL-COEFF-VAL-VAR requires the evaluating variable to have 1 as its corresponding grade and all other variables to have 0; rule EVAL-COEFF-VAL-UNIT requires that every variable be graded with 0; rule EVAL-COEFF-VAL-THUNK simply includes the grade vector in the closure along with the environment, and rule EVAL-COEFF-VAL-VPAIR sums the grades needed to evaluate subterms to their closures.

[4] The system in our extended version [Torczon et al. 2024a] and Coq development includes sums.

$\boxed{\gamma \cdot \rho \vdash_{coeff} V \Downarrow W}$ *(Value rules)*

EVAL-COEFF-VAL-VAR

$$\overline{\overline{0}_1 \cdot \rho_1,\ x \mapsto^1 W,\ \overline{0}_2 \cdot \rho_2 \vdash_{coeff} x \Downarrow W}$$

EVAL-COEFF-VAL-UNIT

$$\overline{\overline{0} \cdot \rho \vdash_{coeff} () \Downarrow ()}$$

EVAL-COEFF-VAL-THUNK

$$\frac{}{\gamma \cdot \rho \vdash_{coeff} \{M\} \Downarrow \mathbf{clo}(\gamma \cdot \rho, \{M\})}$$

EVAL-COEFF-VAL-VPAIR

$$\frac{\gamma_1 \cdot \rho \vdash_{coeff} V_1 \Downarrow W_1 \qquad \gamma_2 \cdot \rho \vdash_{coeff} V_2 \Downarrow W_2}{\gamma_1 + \gamma_2 \cdot \rho \vdash_{coeff} (V_1, V_2) \Downarrow (W_1, W_2)}$$

EVAL-COEFF-VAL-VSUB

$$\frac{\gamma_1 \cdot \rho \vdash_{coeff} V \Downarrow W \qquad \gamma_2 \leq_{co} \gamma_1}{\gamma_2 \cdot \rho \vdash_{coeff} V \Downarrow W}$$

$\boxed{\gamma \cdot \rho \vdash_{coeff} M \Downarrow T}$ *(Computation rules)*

EVAL-COEFF-COMP-ABS

$$\frac{q' \leq_{co} q}{\gamma \cdot \rho \vdash_{coeff} \lambda x^q.M \Downarrow \mathbf{clo}(\gamma \cdot \rho, \lambda x^{q'}.M)}$$

EVAL-COEFF-COMP-CPAIR

$$\frac{}{\gamma \cdot \rho \vdash_{coeff} \langle M_1, M_2 \rangle \Downarrow \mathbf{clo}(\gamma \cdot \rho, \langle M_1, M_2 \rangle)}$$

EVAL-COEFF-COMP-APP-ABS

$$\frac{\gamma_1 \cdot \rho \vdash_{coeff} M \Downarrow \mathbf{clo}(\gamma' \cdot \rho', \lambda x^q.M') \quad \gamma_2 \cdot \rho \vdash_{coeff} V \Downarrow W \quad \gamma' \cdot \rho',\ x \mapsto^q W \vdash_{coeff} M' \Downarrow T \qquad \gamma \equiv \gamma_1 + q \cdot \gamma_2}{\gamma \cdot \rho \vdash_{coeff} M\ V \Downarrow T}$$

EVAL-COEFF-COMP-SPLIT

$$\frac{\gamma_1 \cdot \rho \vdash_{coeff} V \Downarrow (W_1, W_2) \quad \gamma_2 \cdot \rho,\ x_1 \mapsto^q W_1,\ x_2 \mapsto^q W_2 \vdash_{coeff} N \Downarrow T \quad \gamma \equiv q \cdot \gamma_1 + \gamma_2}{\gamma \cdot \rho \vdash_{coeff} \mathbf{case}_q\ V\ \mathbf{of}\ (x_1, x_2)\ \rightarrow\ N \Downarrow T}$$

EVAL-COEFF-COMP-RETURN

$$\frac{\gamma \cdot \rho \vdash_{coeff} V \Downarrow W}{q \cdot \gamma \cdot \rho \vdash_{coeff} \mathbf{return}_q\ V \Downarrow \mathbf{return}_q W}$$

EVAL-COEFF-COMP-FORCE-THUNK

$$\frac{\gamma \cdot \rho \vdash_{coeff} V \Downarrow \mathbf{clo}(\gamma' \cdot \rho', \{M\}) \qquad \gamma' \cdot \rho' \vdash_{coeff} M \Downarrow T}{\gamma \cdot \rho \vdash_{coeff} V! \Downarrow T}$$

EVAL-COEFF-COMP-LETIN-RET

$$\frac{\gamma_1 \cdot \rho \vdash_{coeff} M \Downarrow \mathbf{return}_{q_1} W \quad \gamma_2 \cdot \rho,\ x \mapsto^{q_1 \cdot q_2} W \vdash_{coeff} N \Downarrow T}{q_2 \cdot \gamma_1 + \gamma_2 \cdot \rho \vdash_{coeff} x \leftarrow^{q_2} M\ \mathbf{in}\ N \Downarrow T}$$

EVAL-COEFF-COMP-FST

$$\frac{\gamma \cdot \rho \vdash_{coeff} M \Downarrow \mathbf{clo}(\gamma' \cdot \rho', \langle M_1, M_2 \rangle) \quad \gamma' \cdot \rho' \vdash_{coeff} M_1 \Downarrow T}{\gamma \cdot \rho \vdash_{coeff} M.1 \Downarrow T}$$

EVAL-COEFF-COMP-SND

$$\frac{\gamma \cdot \rho \vdash_{coeff} M \Downarrow \mathbf{clo}(\gamma' \cdot \rho', \langle M_1, M_2 \rangle) \qquad \gamma' \cdot \rho' \vdash_{coeff} M_2 \Downarrow T}{\gamma \cdot \rho \vdash_{coeff} M.2 \Downarrow T}$$

EVAL-COEFF-COMP-CSUB

$$\frac{\gamma' \cdot \rho \vdash_{coeff} M \Downarrow T \qquad \gamma \leq_{co} \gamma'}{\gamma \cdot \rho \vdash_{coeff} M \Downarrow T}$$

Fig. 5. Instrumented operational semantics

Figure 5 also shows computations. Rules EVAL-COEFF-COMP-ABS, EVAL-COEFF-COMP-FORCE-THUNK, EVAL-COEFF-COMP-CPAIR, EVAL-COEFF-COMP-FST, and EVAL-COEFF-COMP-SND are largely the same as before, just with the inclusion of grade vectors along with environments.

In rule EVAL-COEFF-COMP-RETURN, we scale the grade needed to evaluate the subterm to its closure by q. In the elimination rules EVAL-COEFF-COMP-APP-ABS, EVAL-COEFF-COMP-LETIN-RET, and

EVAL-COEFF-COMP-SPLIT, if we are eliminating a value V and binding it to a variable x with a grade q for use in some computation M, we must scale the grade vector needed to evaluate V by q before adding it to the grade vector needed to continue with M, as in the typing rules.

We prove a coeffect soundness theorem stating that if a term is well-typed with some grade vector γ, then given γ and some environment ρ that provides values of the correct type for all free variables, it can evaluate to a terminal. Because both values and computations make demands on their inputs, we state this property for both. We formalize the requirement on ρ as $\Gamma \vDash \rho$ in our logical relation below, and this theorem follows immediately from the fundamental lemma.

THEOREM 3.1 (COEFFECT SOUNDNESS). *Let Γ be a context and ρ an environment mapping all variables in the domain of Γ to closed values of the expected type, such that $\Gamma \vDash \rho$. Then:*

(1) *If $\gamma \cdot \Gamma \vdash_{coeff} V : A$ then $\gamma \cdot \rho \vdash_{coeff} V \Downarrow W$ for some closed value W.*
(2) *If $\gamma \cdot \Gamma \vdash_{coeff} M : B$ then $\gamma \cdot \rho \vdash_{coeff} M \Downarrow T$ for some closed terminal computation T.*

The proof of coeffect soundness is similar to the proof of effect soundness, and requires a similar logical relation.

Definition 3.2 (CBPV with General Coeffects: Logical Relation).

$$
\begin{array}{lcll}
\mathcal{W}[\![\mathbf{U}\, B]\!] & = & \{\ \mathbf{clo}(\gamma \cdot \rho, \{M\}) & \mid (\gamma \cdot \rho, M) \in \mathcal{M}[\![B]\!]\ \} \\
\mathcal{W}[\![\mathbf{unit}]\!] & = & \{\ ()\ \} & \\
\mathcal{W}[\![A_1 \times A_2]\!] & = & \{\ (W_1, W_2) & \mid W_1 \in \mathcal{W}[\![A_1]\!] \text{ and } W_2 \in \mathcal{W}[\![A_2]\!]\} \\
 & & & \\
\mathcal{T}[\![\mathbf{F}_q\, A]\!] & = & \{\ \mathbf{return}_q\, W & \mid W \in \mathcal{W}[\![A]\!]\ \} \\
\mathcal{T}[\![A^q \to B]\!] & = & \{\ \mathbf{clo}(\gamma \cdot \rho, \lambda x^q.M) & \mid \text{for all } W \in \mathcal{W}[\![A]\!], ((\gamma \cdot \rho,\ x \mapsto^q W), M) \in \mathcal{M}[\![B]\!]\ \} \\
\mathcal{T}[\![B_1\ \&\ B_2]\!] & = & \{\ \mathbf{clo}(\gamma \cdot \rho, \langle M_1, M_2 \rangle) & \mid (\gamma \cdot \rho, M_1) \in \mathcal{M}[\![B_1]\!] \text{ and } (\gamma \cdot \rho, M_2) \in \mathcal{M}[\![B_2]\!]\ \} \\
 & & & \\
\textit{Closures} & & & \\
\mathcal{V}[\![A]\!] & = & \{\ (\gamma \cdot \rho, V) & \mid \gamma \cdot \rho \vdash_{coeff} V \Downarrow W \text{ and } W \in \mathcal{W}[\![A]\!]\ \} \\
\mathcal{M}[\![B]\!] & = & \{\ (\gamma \cdot \rho, M) & \mid \gamma \cdot \rho \vdash_{coeff} M \Downarrow T \text{ and } T \in \mathcal{T}[\![B]\!]\ \}
\end{array}
$$

Definition 3.3 (CBPV with General Coeffects: Semantic Typing).

$$
\begin{array}{lcl}
\Gamma \vDash \rho & = & x : A \in \Gamma \text{ implies exists } W,\ x \mapsto W \in \rho \text{ and } W \in \mathcal{W}[\![A]\!] \\
\gamma \cdot \Gamma \vDash_{coeff} V : A & = & \text{for all } \rho, \Gamma \vDash \rho \text{ implies exists } W,\ \gamma \cdot \rho \vdash_{coeff} V \Downarrow W \text{ and } W \in \mathcal{W}[\![A]\!] \\
\gamma \cdot \Gamma \vDash_{coeff} M : B & = & \text{for all } \rho, \Gamma \vDash \rho \text{ implies } (\gamma \cdot \rho, M) \in \mathcal{M}[\![B]\!]
\end{array}
$$

We can now state the fundamental lemma, which derives the soundness theorem as a corollary.

THEOREM 3.4 (FUNDAMENTAL LEMMA: COEFFECT SOUNDNESS). *For all γ, Γ, if $\gamma \cdot \Gamma \vdash_{coeff} V : A$ then $\gamma \cdot \Gamma \vDash_{coeff} V : A$, and for all γ, Γ, if $\gamma \cdot \Gamma \vdash_{coeff} M : B$ then $\gamma \cdot \Gamma \vDash_{coeff} M : B$.*

We can show 3.1 by unfolding the definitions of $\gamma \cdot \Gamma \vDash_{coeff} V : A$ and $\gamma \cdot \Gamma \vDash_{coeff} M : B$, which give us the desired evaluations.

3.2 A Strange Semantics?

The operational semantics and soundness proof in this section work for any instantiation of the coeffect semiring. However, this semantics has strange implications for the resource usage coeffect. Here, the soundness theorem should say that if $\gamma \cdot \rho \vdash_{coeff} M \Downarrow T$, then the evaluation of M used its variables at most the number of times indicated by γ. If γ says that a variable x has grade 0, then there should never be a use of rule EVAL-COEFF-VAL-VAR with the variable x.

$\boxed{\gamma \cdot \Gamma \vdash_{lin} M : B}$ *(Modified typing rule)*

$$\textsc{lin-letin}\quad \frac{q' \;=\; q_2 \parallel 1 \qquad \gamma_1 \cdot \Gamma \vdash_{lin} M : \mathbf{F}_{q_1}\ A \qquad \gamma_2 \cdot \Gamma, x :^{q_1 \cdot q'} A \vdash_{lin} N : B}{(q' \cdot \gamma_1) + \gamma_2 \cdot \Gamma \vdash_{lin} x \leftarrow^{q_2} M \textbf{ in } N : B}$$

$\boxed{\gamma \cdot \rho \vdash_{lin} M \Downarrow T}$ *(New and modified computation rules)*

$$\textsc{eval-lin-comp-app-abs-zero}\quad \frac{\gamma \cdot \rho \vdash_{lin} M \Downarrow \textbf{clo}(\gamma' \cdot \rho', \lambda x^0.M') \qquad (\gamma' \cdot \rho'), (x \mapsto^0 \lightning) \vdash_{lin} M' \Downarrow T}{\gamma \cdot \rho \vdash_{lin} M\ V \Downarrow T}$$

$$\textsc{eval-lin-comp-ret-zero}\quad \frac{}{\overline{0} \cdot \rho \vdash_{lin} \textbf{return}_0\ V \Downarrow \textbf{return}_0 \lightning}$$

$$\textsc{eval-lin-comp-split-zero}\quad \frac{\gamma \cdot \rho,\ x_1 \mapsto^0 \lightning,\ x_2 \mapsto^0 \lightning \vdash_{lin} N \Downarrow T}{\gamma \cdot \rho \vdash_{lin} \textbf{case}_0\ V\ \textbf{of}\ (x_1, x_2)\ \rightarrow\ N \Downarrow T}$$

$$\textsc{eval-lin-comp-letin-ret}\quad \frac{q' \;=\; q_2 \parallel 1 \qquad \gamma_1 \cdot \rho \vdash_{lin} M \Downarrow \textbf{return}_{q_1} W \qquad \gamma_2 \cdot \rho,\ x \mapsto^{q_1 \cdot q'} W \vdash_{lin} N \Downarrow T}{q' \cdot \gamma_1 + \gamma_2 \cdot \rho \vdash_{lin} x \leftarrow^{q_2} M \textbf{ in } N \Downarrow T}$$

Fig. 6. Typing rules and instrumented operational semantics for resource tracking

But, on closer examination of the operational semantics, this is not exactly what this soundness theorem implies. Consider the following example:

$$x :^0 \mathbf{U}\,(\mathbf{F}\,\mathbf{unit}) \vdash_{coeff} z_1 \leftarrow^0 x! \textbf{ in return}_1\ () : \mathbf{F}_1\ \mathbf{unit}$$

$x!$ does not contribute to the final result, and the resources used in its evaluation are accordingly multiplied by 0 when we calculate the grade for x in the context. However, our semantics evaluates x once here using rule EVAL-COEFF-VAL-VAR, violating the principle we described above.

More generally, we encounter this issue with any rule in the operational semantics that scales resources based on some annotation in the terms. For example, in rule EVAL-COEFF-COMP-APP-ABS, the resources used by the evaluation of the argument γ_2 are scaled by q, the grade on the function argument. The total resources of the application γ must equal this scaled vector plus γ_1, the resources used to evaluate the function – *i.e.*, we must have $\gamma \equiv \gamma_1 + q \cdot \gamma_2$. What if q is 0? The resources needed to compute the argument are then not accounted for in γ. This suggests that we should not evaluate the argument at all in this case, so we need to adjust our operational semantics.

4 CBPV and Coeffects (Version 2: Resource Tracking)

In this section, we discuss how, with a few additional axioms, we can modify our instrumented operational semantics and type system to produce a better model for resource tracking. Our goal is to ensure that we never evaluate values and computations without including their resource usage in the final count. The modifications that we discuss here are summarized in Figure 6. We use the judgements $\gamma \cdot \Gamma \vdash_{lin} M : B$ and $\gamma \cdot \Gamma \vdash_{lin} V : A$ to refer to the modified typing rules of this section and $\gamma \cdot \rho \vdash_{lin} M \Downarrow T$ to refer to the modified operational semantics, highlighting the connection between resource usage coeffects and bounded linear logic.

First, we axiomatize that the semiring is nontrivial. If $1 \;=\; 0$, resource tracking via grades is meaningless, and our general semantics degenerates to standard CBPV. Second, we require that

if $0 \leq_{co} q_1 + q_2$, then $q_1 = 0$ and $q_2 = 0$. If either subterm in a value pair requires nonzero resources, we should not be able to evaluate the pair with no resources. Finally, for similar reasons, we require that there be no nonzero zero divisors in the semiring, *i.e.*, if $0 = q_1 \cdot q_2$, then $q_1 = 0$ or $q_2 = 0$. Semirings that satisfy these additional constraints include natural numbers, with their usual or discrete orderings, or the $\{0, 1, \omega\}$ semiring that tracks whether inputs are unused, only used linearly, or with any usage. Note that 1 is incomparable to 0 in this semiring.

In this system, the 0 grade denotes that the corresponding variable is inaccessible, i.e., used 0 times, so anywhere we eliminate a value and bind it to an inaccessible variable (or return a value with grade 0), we require special treatment. Rules EVAL-COEFF-COMP-APP-ABS, EVAL-COEFF-COMP-RETURN, and EVAL-COEFF-COMP-SPLIT all have this property, so we modify these rules to require that the relevant grade be nonzero. We also add new rules that apply when the grade *is* zero. These rules, shown in Figure 6, discard the unused value V without evaluating it and use a new, untyped, closed value $\lightning$ in place of the result of evaluating V. Because values are pure, discarding an unused value does not alter any effects of the program.

However, rule EVAL-COEFF-COMP-LETIN-RET requires special consideration. Unlike in the rules above, which discard values, this rule discards a *computation* – but because that computation could be effectful, this could change the semantics in unintended ways. Following related work [Dal Lago and Gavazzo 2022; Gavazzo 2018], we reconcile this by adding a notion of $q \parallel 1$, which is equivalent to q when q is nonzero and 1 otherwise. We continue to allow the syntax of the term itself to contain any q_2, but the rest of the typing rule refers to $q_2 \parallel 1$ instead. (All other typing rules stay the same.) The evaluation rule, rule EVAL-LIN-COMP-LETIN-RET, follows the same pattern (see Figure 6). Note that this modified evaluation rule introduces a new source of imprecision: we may consume resources to evaluate code without ever using its result, making our final resource accounting more of an overapproximation.

With these modifications, we update our logical relation with a special case for zero resources below. For brevity we show only the changes.

Definition 4.1 (CBPV with Resource Coeffects: Logical Relation).

Closed graded values

$$\begin{array}{lcl} \mathcal{W}_0[\![A]\!] & = & \{\ \lightning\ \} \\ \mathcal{W}_q[\![A]\!] & = & \mathcal{W}[\![A]\!] \text{ when } q \neq 0 \end{array}$$

Closed terminals

$$\begin{array}{lcl} \mathcal{T}[\![\mathbf{F}_q\ A]\!] & = & \{\ \mathbf{return}_q\, W \quad \mid W \in \mathcal{W}_q[\![A]\!]\ \} \\ \mathcal{T}[\![A^q \to B]\!] & = & \{\ \mathbf{clo}(\gamma\cdot\rho, \lambda x^q.M) \quad \mid \textit{forall } W \in \mathcal{W}_q[\![A]\!], \\ & & \qquad ((\gamma\cdot\rho,\ x \mapsto^q W), M) \in \mathcal{M}[\![B]\!]\ \} \end{array}$$

Furthermore, we update our semantic typing relation for environments to also include a special case for zero; in this case the environment need not have a closed value for that variable. (The remaining definitions do not change other than to use the resource accounting operational semantics. In particular, $\mathcal{V}[\![A]\!]$ still requires the resulting closed value to be in $\mathcal{W}[\![A]\!]$.)

Definition 4.2 (CBPV with Resource Coeffects: Semantic Typing).

$$\gamma \cdot \Gamma \vDash \rho \quad = \quad x :^q A \in \gamma\cdot\Gamma \textit{ implies } q = 0 \textit{ or } (x \mapsto W \in \rho \textit{ and } W \in \mathcal{W}[\![A]\!])$$

With these updates, we again prove the fundamental theorem. As in the previous section, if we unfold the definitions above, this theorem gives us exactly the soundness theorem we would like.

THEOREM 4.3 (FUNDAMENTAL LEMMA: RESOURCE SOUNDNESS). *For all γ, Γ, if $\gamma\cdot\Gamma \vdash_{lin} V : A$, then $\gamma \cdot \Gamma \vDash_{lin} V : A$, and for all γ, Γ, if $\gamma\cdot\Gamma \vdash_{lin} M : B$, then $\gamma \cdot \Gamma \vDash_{lin} M : B$.*

$$\textsc{cbncoeff-app}\quad \frac{\gamma_1 \cdot \Gamma \vdash_{cbncoeff} e_1 : \tau_1^{q} \to \tau_2 \qquad \gamma_2 \cdot \Gamma \vdash_{cbncoeff} e_2 : \tau_1}{\gamma_1 + q \cdot \gamma_2 \cdot \Gamma \vdash_{cbncoeff} e_1\ e_2 : \tau_2}$$

$$\textsc{cbncoeff-box}\quad \frac{\gamma_1 \cdot \Gamma \vdash_{cbncoeff} e : \tau}{q \cdot \gamma_1 \cdot \Gamma \vdash_{cbncoeff} \mathbf{box}_q\ e : \Box_q\ \tau}$$

$$\textsc{cbncoeff-unbox}\quad \frac{q' = q_2 \parallel 1 \qquad \gamma_1 \cdot \Gamma \vdash_{cbncoeff} e_1 : \Box_{q_1}\ \tau \qquad \gamma_2 \cdot \Gamma, x :^{q_1 \cdot q'} \tau \vdash_{cbncoeff} e_2 : \tau'}{q' \cdot \gamma_1 + \gamma_2 \cdot \Gamma \vdash_{cbncoeff} \mathbf{unbox}_{q_2}\ x = e_1\ \mathbf{in}\ e_2 : \tau'}$$

Fig. 7. CBN with coeffect tracking

$$\textsc{cbvcoeff-app}\quad \frac{q' = q \parallel 1 \qquad \gamma_1 \cdot \Gamma \vdash_{cbvcoeff} e_1 : \tau_1^{q'} \to \tau_2 \qquad \gamma_2 \cdot \Gamma \vdash_{cbvcoeff} e_2 : \tau_1}{\gamma_1 + q' \cdot \gamma_2 \cdot \Gamma \vdash_{cbvcoeff} e_1\ ^{q} e_2 : \tau_2}$$

$$\textsc{cbvcoeff-box}\quad \frac{q' = q \parallel 1 \qquad \gamma_1 \cdot \Gamma \vdash_{cbvcoeff} e : \tau}{q' \cdot \gamma_1 \cdot \Gamma \vdash_{cbvcoeff} \mathbf{box}_q\ e : \Box_{q'}\ \tau}$$

$$\textsc{cbvcoeff-unbox}\quad \frac{q' = q_2 \parallel 1 \qquad \gamma_1 \cdot \Gamma \vdash_{cbvcoeff} e_1 : \Box_{q_1}\ \tau \qquad \gamma_2 \cdot \Gamma, x :^{q_1 \cdot q'} \tau \vdash_{cbvcoeff} e_2 : \tau'}{q' \cdot \gamma_1 + \gamma_2 \cdot \Gamma \vdash_{cbvcoeff} \mathbf{unbox}_{q_2}\ x = e_1\ \mathbf{in}\ e_2 : \tau'}$$

Fig. 8. CBV with coeffect tracking

We can also use this theorem to reason about unused variables. For example, suppose we type check some computation M in the context of an inaccessible variable x. Instantiating the theorem above with this context assures us that evaluation succeeds even when variables are mapped to $\lightning$ in the environment.

Corollary 4.4 (Inaccessible variable example). *For all M and B, if $x :^{0} A \vdash_{lin} M : B$, then there exists some T, such that $x \mapsto^{0} \lightning \vdash_{lin} M \Downarrow T$.*

Because the operational semantics does not include any rules for evaluating $\lightning$, we can conclude that 0-marked variables are never used by the operational semantics. Furthermore, there are no assumptions about the structure of $\lightning$ values, so we can discard them during computation.

4.1 Translation Soundness

As with effects, we explore the translation of coeffect-aware CBN and CBV λ-calculi to CBPV. As in our CBPV extension with coeffects, the source type systems are parameterized by a pre-ordered semiring structure of coeffects and combine the typing context with γ, a vector of coeffect annotations that describe the demands on each variable.

The type-and-coeffect system that we consider as the starting point of our CBN translation is adapted from the simple type system of Choudhury et al. [2021] and is similar to the system developed by Abel and Bernardy [2020]. The differences between this source language and the related work are minor. The design of our CBV language is inspired by Dal Lago and Gavazzo [2022]. To make the comparison clear, we present it as a standard CBV lambda calculus instead of fine-grained CBV. Other changes to the language include the introduction of subcoeffecting, allowing functions to take q copies of their argument instead of one (and annotating applications with q), and replacing $q \wedge 1$ with $q \parallel 1$ to force the evaluation of subterms. (We choose $q \parallel 1$ over $q \wedge 1$ to avoid requiring the existence of $q \wedge 1$ as an axiom of the semiring. The difference is minor.)

The rules for the CBN version of the system appear in Figure 7; the rules for the CBV version are in Figure 8. Most of the rules parallel those of the corresponding terms in CBPV; for brevity, then, we show only the rules for application, boxing and unboxing here. The two languages differ in the application rule. In CBV, we annotate applications with the number of times the function uses its argument. Because the argument will always be evaluated once in CBV, if q is zero, we force it to be one.

These latter two terms introduce and eliminate the modal type $\Box_q\ \tau$. The introduction form requires grade q on its argument. When we unbox the argument, the second subterm has access to it with grade $q_1 \cdot q_2$. The q_1 comes from when the box was created, and the q_2 comes from the unboxing term, as in let bindings in CBPV. In CBV, we use **letin** in both translations, so we include $q \parallel 1$ in both rules in an analogous way to its use in rule COEFF-LETIN. In CBN, we use **letin** in the translation of **unbox** but not **box**, so we can drop $q \parallel 1$ from the typing rule for box. This imprecision makes sense in the source languages for the same reason it makes sense in CBPV: because we are combining effects and coeffects, we sometimes need to evaluate subterms for their effects even if the results of those subterms are never used.

4.1.1 Call-by-Name Translation. We first consider a call-by-name translation to CBPV. For brevity, we show just the translation of function and box types on the left below and the translation of applications and the **box** and **unbox** terms on the right.

$$\begin{array}{ll} [\![\tau_1^{\,q} \to \tau_2]\!]_{\mathrm{N}} & = (\mathbf{U}\,[\![\tau_1]\!]_{\mathrm{N}})^q \to [\![\tau_2]\!]_{\mathrm{N}} \\ [\![\Box_q\ \tau]\!]_{\mathrm{N}} & = \mathbf{F}_q\ (\mathbf{U}\,[\![\tau]\!]_{\mathrm{N}}) \end{array} \qquad \begin{array}{ll} [\![e_1\ e_2]\!]_{\mathrm{N}} & = [\![e_1]\!]_{\mathrm{N}}\ \{[\![e_2]\!]_{\mathrm{N}}\} \\ [\![\mathbf{box}_q\ e]\!]_{\mathrm{N}} & = \mathbf{return}_q\ \{[\![e]\!]_{\mathrm{N}}\} \\ [\![\mathbf{unbox}_q\ x = e_1\ \mathbf{in}\ e_2]\!]_{\mathrm{N}} & = x \leftarrow^q [\![e_1]\!]_{\mathrm{N}}\ \mathbf{in}\ [\![e_2]\!]_{\mathrm{N}} \end{array}$$

In this translation, the coeffect on the λ-calculus function type translates directly to the coeffect on the CBPV function type. Furthermore, the modal type $\Box_q\ \tau$ is a graded comonad, so it can be translated to the comonad in CBPV, adding the grade to the returner type.

The CBN translation of λ terms is as usual. However, the translation of the box introduction and elimination forms follows from the definition of the CBPV comonadic type. To create a box, we return the thunked translation of the expression. To eliminate a box, we use **letin** to move the thunk to the environment.

4.1.2 Call-by-Value Translation. Next, we define a corresponding CBV translation to CBPV. For brevity, we again show only the translation of function and graded modal types and of applications and the **box** and **unbox** terms.

$$\begin{array}{ll} [\![\tau_1^{\,q} \to \tau_2]\!]_{\mathrm{V}} & = \mathbf{U}\,([\![\tau_1]\!]_{\mathrm{V}}^q \to \mathbf{F}_1\ [\![\tau_2]\!]_{\mathrm{V}}) \\ [\![\Box_q\ \tau]\!]_{\mathrm{V}} & = \mathbf{U}\,(\mathbf{F}_q\ [\![\tau]\!]_{\mathrm{V}}) \end{array}$$

$$\begin{array}{ll} [\![e_1\,{}^q\,e_2]\!]_{\mathrm{V}} & = x \leftarrow^1 [\![e_1]\!]_{\mathrm{V}}\ \mathbf{in}\ y \leftarrow^q [\![e_2]\!]_{\mathrm{V}}\ \mathbf{in}\ x!\ y \\ [\![\mathbf{box}_q\ e]\!]_{\mathrm{V}} & = x \leftarrow^q [\![e]\!]_{\mathrm{V}}\ \mathbf{in}\ \mathbf{return}_1\ \{\mathbf{return}_{q\,\parallel\,1}\ x\} \\ [\![\mathbf{unbox}_q\ x = e_1\ \mathbf{in}\ e_2]\!]_{\mathrm{V}} & = y \leftarrow^q [\![e_1]\!]_{\mathrm{V}}\ \mathbf{in}\ x \leftarrow^q y!\ \mathbf{in}\ [\![e_2]\!]_{\mathrm{V}} \end{array}$$

As above, we propagate the coeffect from the λ-calculus function type directly to the CBPV function type. Similarly, we propagate the grade in the modal type to the inner returner type and let binding in CBPV.

For applications, we use let bindings to access the translations of the function and the argument. The argument is not thunked in translation, so it is strict, but the function is thunked in translation, so we must force it before applying it. **box** is also strict in CBV, so its translation first evaluates its argument. The rest of the translation follows its type definition. In CBPV, the computation

$x \leftarrow^{1} M$ **in return**$_1$ x is equivalent to M, but the computation $x \leftarrow^{q} M$ **in return**$_q$ x corresponds to duplicating M q times in a resource usage coeffect. This propagation of the grade is exactly the feature that we need to translate the **box** term. Like the CBN translation of the modal type, in the CBV translation, the comonadic type is difficult to access. In this translation, **box** must include an extra thunk that is forced in the translation of the **unbox** term, giving us access to the comonadic type **F U**. We must also use the annotation capability of **letin** (twice) to mirror the annotation in the source language. The correctness proofs for both the CBN and CBV translations follow from the corresponding proofs of the combined system (taking the trivial effect).

5 Combined Effects and Coeffects

Next, we present a system that tracks both effects and coeffects, by combining the effect system of Section 2 with the resource usage system of Section 4, and adding one new rule.

Definition 5.1 (Combined type system). The judgements $\gamma \cdot \Gamma \vdash_{full} V : A$ and $\gamma \cdot \Gamma \vdash_{full} M :^{\phi} B$ refer to the CBPV type system with effect annotations from Figure 1 and coeffect annotations (resource tracking version) from Figure 6.[5]

This type system is a straightforward combination of the systems presented earlier. For example, the typing rule FULL-LETIN combines rule EFF-LETIN with rule LIN-LETIN and includes both the grade vector $q_2' \cdot \gamma_1 + \gamma_2$ and the effect $\phi_1 \cdot \phi_2$ for the computation.

$$\textsc{full-letin}\quad \frac{q_2' \;=\; q_2 \parallel 1 \qquad \gamma_1 \cdot \Gamma \vdash_{full} M_1 :^{\phi_1} \mathbf{F}_{q_1}\, A \qquad \gamma_2 \cdot \Gamma, x :^{q_1 \cdot q_2'} A \vdash_{full} M_2 :^{\phi_2} B}{(q_2' \cdot \gamma_1) + \gamma_2 \cdot \Gamma \vdash_{full} x \leftarrow^{q_2} M_1 \textbf{ in } M_2 :^{\phi_1 \cdot \phi_2} B}$$

Similarly, we augment our instrumented operational semantics to track both effects and coeffects.

Definition 5.2 (Combined Resource Semantics). The judgements $\gamma \cdot \rho \vdash_{full} V \Downarrow W$ and $\gamma \cdot \rho \vdash_{full} M \Downarrow T \,\#\, \phi$ refer to the CBPV operational semantics with effect annotations from Figures 2 and 3 and coeffect annotations from Figure 5, with updates for resource tracking from Figure 6.

For example, the **letin** evaluation rule computes the instrumented grade vector and effect and requires that the computation M be evaluated at least once, as in rule EVAL-LIN-COMP-LETIN-RET.

$$\textsc{eval-full-comp-letin}\quad \frac{q_2' \;=\; q_2 \parallel 1 \qquad \gamma_1 \cdot \rho \vdash_{full} M \Downarrow \textbf{return}_{q_1}\, W \,\#\, \phi_1 \qquad \gamma_2 \cdot \rho,\; x \mapsto^{q_1 \cdot q_2'} W \vdash_{full} N \Downarrow T \,\#\, \phi_2}{q_2' \cdot \gamma_1 + \gamma_2 \cdot \rho \vdash_{full} x \leftarrow^{q_2} M \textbf{ in } N \Downarrow T \,\#\, \phi_1 \cdot \phi_2}$$

We can use this operational semantics to show both effect and coeffect soundness of the combined type system. However, before we do so, we make one more extension to the language.

Skipping Unused Discardable Computations. In Section 4, we developed several "zero" rules for discarding unused *values*. But, unused *computations* could not be discarded, because they may have effects. However, in this system, we can identify unused, pure computations, and add a new syntactic form, written $x \leftarrow^{0}_{\varepsilon} M$ **in** N, indicating that they can be discarded. The typing rule (below left) requires that M be effect free and its result unused in N.

$$\textsc{full-letin-zero}\quad \frac{\gamma_1 \cdot \Gamma \vdash_{full} M :^{\epsilon} \mathbf{F}_q\, A \qquad \gamma_2 \cdot \Gamma, x :^{0} A \vdash_{full} N :^{\phi} B}{\gamma_2 \cdot \Gamma \vdash_{full} x \leftarrow^{0}_{\varepsilon} M \textbf{ in } N :^{\phi} B}$$

$$\textsc{eval-full-comp-letin-zero}\quad \frac{\gamma \cdot \rho,\; x \mapsto^{0} \not\downarrow \vdash_{full} N \Downarrow T \,\#\, \phi}{\gamma \cdot \rho \vdash_{full} x \leftarrow^{0}_{\varepsilon} M \textbf{ in } N \Downarrow T \,\#\, \phi}$$

[5] For space, we do not include the entire combined system here. The full rules of this system are available in the extended version of this paper [Torczon et al. 2024a] and in the Coq development.

Furthermore, the operational semantics of this new expression form (above right) does not evaluate M. Instead it uses the junk value $\lightning$ for the result of this computation.

To see this rule in action, consider the CBV translation of an expression $y_2\ (y_1\ x)$, where y_2 is a constant function and y_1 is pure. In this case, the type system can observe that x does not contribute to the final result when the application of y_1 to x is marked as discardable.

$$x :^0 A, y_1 :^0 \mathbf{U}_\varepsilon\ (A^1 \to \mathbf{F}\,A), y_2 :^1 \mathbf{U}_\phi\ (A^0 \to B) \vdash_{full} z \leftarrow^0_\varepsilon y_1!\,x\ \mathbf{in}\ y_2!\,z :^\phi B$$

Soundness Proof for Discardable Computations. We next show that discarding unused values and unused pure computations does not change the evaluation behavior of computations. To do so, we need the following properties that state that ε is the minimum element of the effect preorder. [6]

Definition 5.3 (Min identity). (1) For all ϕ, $\varepsilon \leq_{eff} \phi$ (2) For all ϕ, $\phi \leq_{eff} \varepsilon$ *implies* $\phi = \varepsilon$.

To prove that discarding is sound, we establish a relation between our combined resource semantics and one that does not discard terms.

Definition 5.4 (Combined nondiscarding semantics). The judgement $\gamma \cdot \rho \vdash_{gen} M \Downarrow T \# \phi$ refers to the operational semantics that is the combination of CBPV with effect annotations from Figure 3 and coeffect annotations from Figure 5, with the modified rule FULL-LETIN that always evaluates computations. This semantics does not include rules that discard values or computations and uses rule EVAL-FULL-COMP-LETIN to evaluate the new **letin** expression.

Our simulation lemma states that for closed boolean-valued computations[7], evaluating with either the nondiscarding semantics (Definition 5.4) or with the resource semantics (Definition 5.2) produces the same result and the same effect.

LEMMA 5.5 (RESOURCE SIMULATION). *If* $\emptyset \cdot \varnothing \vdash_{full} M :^\phi \mathbf{F}_1\ (\mathbf{unit} + \mathbf{unit})$ *then either*

(1) $\emptyset \cdot \emptyset \vdash_{gen} M \Downarrow \mathbf{return}_1(\mathbf{inl}\,()) \# \phi_1$ *and* $\emptyset \cdot \emptyset \vdash_{full} M \Downarrow \mathbf{return}_1(\mathbf{inl}\,()) \# \phi_1$ *or*
(2) $\emptyset \cdot \emptyset \vdash_{gen} M \Downarrow \mathbf{return}_1(\mathbf{inr}\,()) \# \phi_1$ *and* $\emptyset \cdot \emptyset \vdash_{full} M \Downarrow \mathbf{return}_1(\mathbf{inr}\,()) \# \phi_1$.

This simulation lemma is a corollary of a much more general result—the fundamental lemma for a binary logical relation between computations that are evaluated with the two different semantics. This relation, shown below, is mutually defined with relations between closed values and closed terminals (not shown, but available in the extended version [Torczon et al. 2024a] and in the Coq development).

$$\begin{aligned}\mathcal{M}[\![B]\!]^\phi \quad = \quad \{\ (\gamma\cdot\rho_1, M_1, \gamma\cdot\rho_2, M_2) \mid \gamma\cdot\rho_1 \vdash_{gen} M_1 \Downarrow T_1 \# \phi_1 \ \textit{and}\ \gamma\cdot\rho_2 \vdash_{full} M_2 \Downarrow T_2 \# \phi_1 \\ \textit{and}\ (T_1, T_1) \in \mathcal{T}[\![B]\!]^{\phi_2}\ \textit{and}\ (T_1, T_2) \in \mathcal{T}[\![B]\!]^{\phi_2}\ \textit{and}\ \phi_1 \cdot \phi_2 \leq_{eff} \phi\ \}\end{aligned}$$

Using this relation, we define a binary version of the semantic typing relation. Two environments ρ_1 and ρ_2 are related when the closed values in the first environment are related to themselves, and, if the usage is nonzero, the closed value in the second environment is related to the first. The first condition ensures that we know something about closed values in the first relation even when the corresponding value in the second relation has been discarded in the resource semantics.

[6] These properties hold for **tick** effects, but we have not used them before now. [7] The system in our extended version [Torczon et al. 2024a] and Coq development includes sums, necessary to implement the boolean type.

Definition 5.6 (Semantic double typing).

$$\begin{aligned}
\gamma \cdot \Gamma \vDash \rho_1 \sim \rho_2 \quad &= x :^q A \in \gamma \cdot \Gamma \text{ implies } x \mapsto W_1 \in \rho_1 \text{ and } (W_1, W_1) \in \mathcal{W}[\![A]\!] \text{ and} \\
&\quad (q = 0 \text{ or } (x \mapsto W_2 \in \rho_2 \text{ and } (W_1, W_2) \in \mathcal{W}[\![A]\!])) \\
\gamma \cdot \Gamma \vDash_{full} V_1 \sim V_2 : A \quad &= \text{forall } \rho_1, \rho_2, \gamma \cdot \Gamma \vDash \rho_1 \sim \rho_2 \\
&\quad \text{implies } \gamma \cdot \rho_1 \vdash_{full} V_1 \Downarrow W_1 \text{ and } \gamma \cdot \rho_2 \vdash_{full} V_1 \Downarrow W_1 \\
&\quad \text{and } (W_1, W_1) \in \mathcal{W}[\![A]\!] \text{ and } (W_1, W_2) \in \mathcal{W}[\![A]\!] \\
\gamma \cdot \Gamma \vDash_{full} M_1 \approx M_2 :^\phi B \quad &= \text{forall } \rho_1, \rho_2, \gamma \cdot \Gamma \vDash \rho_1 \sim \rho_2 \text{ implies } (\gamma \cdot \rho_1, M_1, \gamma \cdot \rho_2, M_2) \in \mathcal{M}[\![B]\!]^\phi
\end{aligned}$$

The fundamental theorem shows that this binary relation is reflexive.

Theorem 5.7 (Fundamental lemma: simulation).

(1) *For all* γ, Γ, *if* $\gamma \cdot \Gamma \vdash_{full} V : A$, *then* $\gamma \cdot \Gamma \vDash_{full} V \sim V : A$, *and*
(2) *for all* γ, Γ, *if* $\gamma \cdot \Gamma \vdash_{full} M :^\phi B$, *then* $\gamma \cdot \Gamma \vDash_{full} M \approx M :^\phi B$.

This fundamental lemma combines and generalizes prior results of this paper. In particular, it shows the effect-and-coeffect soundness of the combined type system with respect to both the nondiscarding and resource accounting semantics–the effects and coeffects of the evaluation are bounded by the type system. For clarity, we also separately show effect-and-coeffect soundness of the combined type system in the Coq development.

CBN and CBV Translations. Finally, we have defined CBN and CBV with combined effects and coeffects and have proved the soundness of translations to the combined CPBV type system.

Theorem 5.8 (CBN and CBV translation correctness).

(1) *For all* γ, Γ, e, τ, *if* $\gamma \cdot \Gamma \vdash_{cbncoeff} e : \tau$, *then* $\gamma \cdot [\![\Gamma]\!]_N \vdash_{full} [\![e]\!]_N :^\varepsilon [\![\tau]\!]_N$, *and*
(2) *For all* γ, Γ, e, τ, *if* $\gamma \cdot \Gamma \vdash_{cbvcoeff} e : \tau$, *then* $\gamma \cdot [\![\Gamma]\!]_V \vdash_{full} [\![e]\!]_V :^\varepsilon \mathbf{F}_1\ [\![\tau]\!]_V$,

Like 2.5, these proofs follow by simple induction, so we omit them here; however, they can be found in the Coq development.

6 Related Work

Call-by-push-value (CBPV) was originally developed by Levy [2003b]. Forster et al. [2019] mechanized proofs of its metatheoretic properties and translation soundness and inspired our mechanized proofs. Current applications of CBPV include modeling compiler intermediate languages [New 2019; Rizkallah et al. 2018], understanding the role that polarity plays in bidirectional typing [Dunfield and Krishnaswami 2021] and subtyping [Lakhani et al. 2022], and incorporating effects into dependent type theories [Pédrot and Tabareau 2019; Pédrot et al. 2019].

CBPV and Effects. Call-by-value languages with effect tracking go back to FX [Lucassen and Gifford 1988]. Wadler and Thiemann [2003] showed the connection between graded monads and effects by translating the effect system of Talpin and Jouvelot [1994] to a language that isolates effects using graded monads. Our monadic effect language is inspired by this paper, generalized following Katsumata [2014]. In this paper, our translation is the reverse of Wadler and Thiemann, mapping a language with graded monads to an effect-style extension of CBPV. Like us, Rajani et al. [2021] use a logical relation to show the soundness of their monadic cost analysis.

Although CBPV has often been used to model the semantics of effects, its type system has only rarely been extended with effect tracking. The type system that we present in Section 2 is most similar to MAM (multi-adjunctive metalanguage) from Forster et al. [2017], which builds on Kammar and Plotkin [2012] and Kammar et al. [2013]. Forster et al. use MAM to compare the relative expressiveness of effect handlers, monadic reflection and delimited control. The differences between our system and MAM are in the abstract structure of effects: MAM does not use a preordered

monoid to track effects. Instead, in each extension effects are interpreted differently. For effect handlers, effects are a set of operations specified by some effect signature; for monadic reflection, effects are monad stacks; for delimited control, effects are a stack of computation types.

Wuttke [2021] defines a cost-annotated version of CBPV by annotating the thunk type in CBPV with a bound $[a < I]$ that limits the number of times that thunks can be forced. This work includes both call-by-value and call-by-name translations from cost-annotated PCF terms to cost-annotated CBPV. For expressiveness, the system includes subtyping and indexed types.

Some extensions of CBPV annotate effects on $\mathbf{F}\,A$ instead of $\mathbf{U}\,B$. These systems isolate effects so that they need not be tracked by the typing judgement. Extended Call-by-Push-Value (ECBPV) [McDermott and Mycroft 2019] adds call-by-need evaluation to CBPV and layers an effect system to augment equational reasoning. This system uses an operation $\langle\phi\rangle B$ to extend the effect annotation to other computation types, combining effects in returner types and pushing effects to the result type of functions and inside with-products. Rioux and Zdancewic [2020] tracks divergence. In this system, the sequencing operation requires that the annotation on the returner type be less than or equal to any annotation on the result of the continuation.

Coeffects. Type systems that track coeffects were introduced by Brunel et al. [2014]; Ghica and Smith [2014]; Petricek et al. [2014] and developed by Abel and Bernardy [2020]; Orchard and Eades III [2022]; Orchard et al. [2019]. Early applications were for bounded linearity; but these systems have also been used for tracking information flow in differential privacy [Reed and Pierce 2010], dynamic binding [Nanevski 2003] and have also been applied for resource usage in Haskell [Bernardy et al. 2017] and irrelevance in dependently-typed languages [Abel et al. 2023; Atkey 2018; Choudhury et al. 2021]. Petricek et al. [2014] give a number of additional examples, including dataflow (the number of past values needed in a stream processing language) and data liveness (whether references to a variable are still needed).

As in our work, all prior semantics that "count" uses of variables are imprecise and allow execution to waste resources. Abel et al. [2023] and Choudhury et al. [2021] use a heap-based operational semantics to show coeffect soundness for a language with a small-step, call-by-name semantics, but do not consider the interactions with effects. Bianchini et al. [2023] proves resource soundness for a fine-grained call-by-value language using a big-step semantics. Their language includes a nontermination effect through recursive functions and recursive types. Their soundness proof is based on a heap-based semantics, which must simultaneously evaluate q copies of an expression. In contrast, because our environment-based semantics can separate the resource usage of a subexpression from the rest of the computation, our semantics uses multiplication instead of multi-usage. For consistency with effects, several rules of their type system require that the number of copies of the produced value to be nonzero, similar to our use of $q \parallel 1$.

Dal Lago and Gavazzo [2022] also explore the addition of effects and coeffects to a fine-grained call-by-value language. They also force the **letin** term to count the coeffects of the computation at least once, through the use of $q \wedge 1$. (This rule is derived from Gavazzo [2018].) Unlike our work, Dal Lago and Gavazzo give a denotational semantics based on a monadic evaluation function and do not track resource usage. Their main result is a definition of a program relation in the presence of effects and coeffects. Their approach is to refine a standard logical relation with *relators* and *corelators* that capture the interaction of effects and coeffects with the language semantics. This approach is more general than ours, which is tied to a specific effect and coeffect.

CBPV and Linearity. Our extension of CBPV with coeffect typing is novel and inspired by the duality with effects. The most related systems are those involving linearity in the context of low-level or compiler intermediate languages. Schöpp [2015] develops a low-level language, similar to CBPV, that includes linear operations in its type system. The *enriched effect calculus* [Egger

et al. 2009, 2012] extends a type theory for computational effects, with primitives from linear logic. Ahmed et al. [2007] augment a variant of typed assembly language with linear types. Jang et al. [2024] develop a natural deduction formulation of adjoint logic (which is similar to CBPV) and use its structure to combine linear, affine, strict and intuitionistic logics in a uniform setting.

Interactions Between Effects and Coeffects. Several systems describe interactions between effects and coeffects. Nanevski [2003] uses comonads to guard the usage of local state (dynamic binding) and monads to guard the usage of global state. In each case the type system tracks the set of locations can be safely read and updated. In future work, we would like to extend this work with state effects and local effect handlers so that we can track this interaction using annotations on thunk and returner types, instead of encapsulated within monad and comonadic structures.

Gaboardi et al. [2016] present a combined calculus featuring effects and coeffects. Unlike this work, their lambda calculus isolates effects and coeffects using graded monadic and comonadic modal types. A key feature of their system are "graded distributive laws", that permit interactions between the monad and comonad. The exact interactions are mediated by operations determined by the particular effects and coeffects being modeled. For example, we could distribute a term of type $\Box_3\ (\mathbf{T}_{2_{eff}}\ \tau)$ into a term of type $\mathbf{T}_{6_{eff}}\ (\Box_3\ \tau)$. That is, it could turn 3 copies of a monad which ticks twice and returns a term of type τ into a monad which ticks 6 times then returns 3 copies of the term.

In future work, we hope to add distributivity to this system. Unlike the distributive property described above, in this context the distributive laws need not change the structure of the computation. Instead, we would like it to redistribute grades on types in the form $\mathbf{F}_{q_1}\ \mathbf{U}_{\phi}\ \mathbf{F}_{q_2}\ A$ or $\mathbf{U}_{\phi_1}\ \mathbf{F}_{q}\ \mathbf{U}_{\phi_2}\ B$. However, we have yet to determine what sorts of rearrangement are sound in this context.

7 Conclusion and Future Work

In this paper we have annotated the ambient monad and comonad of CBPV to statically track effects and coeffects. We have presented these extensions separately to provide a gentle introduction, before developing a combined calculus that tracks both simultaneously. We have identified semantic subtleties in resource tracking and have developed an alternative semantics that better describes our understanding of this coeffect. We have proven soundness for all versions of our type system, identifying the required assumptions of the effect and coeffect algebras. To make sure that our designs are expressive, we have shown the standard translations from call-by-value and call-by-name lambda calculi into call-by-push-value preserve tick and resource tracking with our system. By exploring both effects and coeffects together, we were also able to observe similarities between these dual notions, and, more importantly, identify their differences.

However, this work is only the starting point for investigation in this space. The natural next step is to go beyond a single effect (tick) and single coeffect (resource usage) to develop a more general structure for extensions of CBPV, perhaps based on algebraic effects [Plotkin and Pretnar 2008] or effect signatures [Katsumata 2014]. This structure would allow us to verify that our rules stay general in the presence of other effects, such as nontermination and state, or other coeffects, such as information-flow tracking and differential privacy.

We can also extend this work by adding language features that interact with effect and coeffect tracking, such as polymorphism, indexed or dependent types, and quantification over effects and coeffects. Subtyping would captures the idea that the type $\mathbf{U}_{\phi_1}\ B$ is a subtype of $\mathbf{U}_{\phi_2}\ B$ when $\phi_1 \leq_{eff} \phi_2$, and that the type $\mathbf{F}_{q_1}\ A$ is a subtype of $\mathbf{F}_{q_2}\ A$ when $q_2 \leq_{co} q_1$. Finally, we would like to explore the practical concerns of this system in more depth, focusing on how users or compilers might make effective use of the statically tracked information.

Data-Availability Statement

The Coq proofs are available at https://github.com/plclub/cbpv-effects-coeffects and archived at 10.5281/zenodo.12654517.

Acknowledgments

Thanks to Dominic Orchard, Richard Eisenberg and Kevin Diggs for comments and suggestions. Yiyun Liu assisted with the initial setup of our Coq proofs, building on a prior Autosubst development of CBPV in Coq [Forster et al. 2019]. This work was supported by the National Science Foundation under grants CCF-2006535, CNS-2244494, and CCF-2327738.

References

Andreas Abel and Jean-Philippe Bernardy. 2020. A unified view of modalities in type systems. *Proc. ACM Program. Lang.* 4, ICFP, Article 90 (aug 2020), 28 pages. https://doi.org/10.1145/3408972

Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized. *Proc. ACM Program. Lang.* 7, ICFP, Article 220 (aug 2023), 35 pages. https://doi.org/10.1145/3607862

Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L^3: A Linear Language with Locations. *Fundam. Informaticae* 77, 4 (2007), 397–449. http://content.iospress.com/articles/fundamenta-informaticae/fi77-4-06

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 56–65. https://doi.org/10.1145/3209108.3209189

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (dec 2017), 29 pages. https://doi.org/10.1145/3158093

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2023. Resource-Aware Soundness for Big-Step Semantics. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 267 (oct 2023), 29 pages. https://doi.org/10.1145/3622843

Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 76 (apr 2022), 30 pages. https://doi.org/10.1145/3527320

Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–370.

Pritam Choudhury, Harley D. Eades III, Richard A. Eisenberg, and Stephanie Weirich. 2021. A Graded Dependent Type System with a Usage-Aware Semantics. *Proc. ACM Program. Lang.* 5, POPL (Jan. 2021). https://doi.org/10.1145/3434331 Artifact available.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo bee doo bee doo. *Journal of Functional Programming* 30 (2020), e9. https://doi.org/10.1017/S0956796820000039

Ugo Dal Lago and Francesco Gavazzo. 2022. A relational theory of effects and coeffects. *Proc. ACM Program. Lang.* 6, POPL, Article 31 (jan 2022), 28 pages. https://doi.org/10.1145/3498692

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (may 2021), 38 pages. https://doi.org/10.1145/3450952

Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. 2009. Enriching an Effect Calculus with Linear Types. In *Computer Science Logic*, Erich Grädel and Reinhard Kahle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 240–254. https://doi.org/10.1007/978-3-642-04027-6

Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. 2012. The enriched effect calculus: syntax and semantics. *Journal of Logic and Computation* 24, 3 (06 2012), 615–654. https://doi.org/10.1093/logcom/exs025

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *Proc. ACM Program. Lang.* 1, ICFP, Article 13 (aug 2017), 29 pages. https://doi.org/10.1145/3110257

Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. 2019. Call-by-Push-Value in Coq: Operational, Equational, and Denotational Theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and*

Proofs (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 118–131. https://doi.org/10.1145/3293880.3294097

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 476–489. https://doi.org/10.1145/2951913.2951939

Dmitri Garbuzov, William Mansky, Christine Rizkallah, and Steve Zdancewic. 2018. Structural Operational Semantics for Control Flow Graph Machines. arXiv:1805.05400 [cs.PL] https://arxiv.org/abs/1805.05400

Francesco Gavazzo. 2018. Quantitative Behavioural Reasoning for Higher-order Effectful Programs: Applicative Distances. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 452–461. https://doi.org/10.1145/3209108.3209149

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 331–350.

Junyoung Jang, Sophia Roshal, Frank Pfenning, and Brigitte Pientka. 2024. Adjoint Natural Deduction. In *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 299)*, Jakob Rehof (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:23. https://doi.org/10.4230/LIPIcs.FSCD.2024.15

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 145–158. https://doi.org/10.1145/2500365.2500590

Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic Foundations for Effect-Dependent Optimisations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 349–360. https://doi.org/10.1145/2103656.2103698

Shin-ya Katsumata. 2014. Parametric effect monads and semantics of effect systems. *SIGPLAN Not.* 49, 1 (jan 2014), 633–645. https://doi.org/10.1145/2578855.2535846

Zeeshan Lakhani, Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. 2022. Polarized Subtyping. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 431–461.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), 100–126. https://doi.org/10.4204/eptcs.153.8

Paul Blain Levy. 2003a. Adjunction Models For Call-By-Push-Value With Stacks. *Electronic Notes in Theoretical Computer Science* 69 (2003), 248–271. https://doi.org/10.1016/S1571-0661(04)80568-1 CTCS'02, Category Theory and Computer Science.

Paul Blain Levy. 2003b. *Call-by-push-value:A Functional/Imperative Synthesis*. Springer Dordrecht. https://doi.org/10.1007/978-94-007-0954-6

Paul Blain Levy. 2006. Call-by-Push-Value: Decomposing Call-by-Value and Call-by-Name. *Higher Order Symbol. Comput.* 19, 4 (dec 2006), 377–414. https://doi.org/10.1007/s10990-006-0480-6

Paul Blain Levy. 2022. Call-by-Push-Value. *ACM SIGLOG News* 9, 2 (may 2022), 7–29. https://doi.org/10.1145/3537668.3537670

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

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

Dylan McDermott and Alan Mycroft. 2019. Extended Call-by-Push-Value: Reasoning About Effectful Programs and Evaluation Order. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 235–262. https://doi.org/10.1007/978-3-030-17184-1_9

Eugenio Moggi. 1989. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. 14–23. https://doi.org/10.1109/LICS.1989.39155

Aleksandar Nanevski. 2003. From Dynamic Binding to State via Modal Possibility. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming* (Uppsala, Sweden) *(PPDP '03)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/888251.888271

Max S. New. 2019. From Call-by-push-value to Stack-based TAL? Presentation at LOLA 2019. https://maxsnew.com/docs/cbpv-stal-lola-2019.pdf

Dominic Orchard and Harley Eades III. 2022. The Granule Project. https://granule-project.github.io/

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

Dominic Orchard and Tomas Petricek. 2014. Embedding effect systems in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2633357.2633368

Pierre-Marie Pédrot and Nicolas Tabareau. 2019. The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects. *Proc. ACM Program. Lang.* 4, POPL, Article 58 (dec 2019), 28 pages. https://doi.org/10.1145/3371126

Pierre-Marie Pédrot, Nicolas Tabareau, Hans Jacob Fehrmann, and Éric Tanter. 2019. A Reasonably Exceptional Type Theory. *Proc. ACM Program. Lang.* 3, ICFP, Article 108 (jul 2019), 29 pages. https://doi.org/10.1145/3341712

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-Dependent Computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 123–135. https://doi.org/10.1145/2628136.2628160

Gordon Plotkin and Matija Pretnar. 2008. A Logic for Algebraic Effects. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science*. 118–129. https://doi.org/10.1109/LICS.2008.45

Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2021. A Unifying Type-Theory for Higher-Order (Amortized) Cost Analysis. *Proc. ACM Program. Lang.* 5, POPL, Article 27 (jan 2021), 28 pages. https://doi.org/10.1145/3434308

Jason Reed and Benjamin C. Pierce. 2010. Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/1863543.1863568

Nick Rioux and Steve Zdancewic. 2020. Computation Focusing. *Proc. ACM Program. Lang.* 4, ICFP, Article 95 (aug 2020), 27 pages. https://doi.org/10.1145/3408977

Christine Rizkallah, Dmitri Garbuzov, and Steve Zdancewic. 2018. A Formal Equational Theory for Call-By-Push-Value. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer International Publishing, Cham, 523–541.

Ulrich Schöpp. 2015. *Computation-by-Interaction for Structuring Low-Level Computation.* Ph. D. Dissertation. Habilitation thesis, Ludwig-Maximilians-Universität München.

AL Smirnov. 2008. Graded monads and rings of polynomials. *Journal of Mathematical Sciences* 151, 3 (2008), 3032–3051.

Jean-Pierre Talpin and Pierre Jouvelot. 1994. The Type and Effect Discipline. *Inf. Comput.* 111, 2 (1994), 245–296. https://doi.org/10.1006/inco.1994.1046

Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. 2024a. Effects and Coeffects in Call-By-Push-Value (Extended Version). arXiv:2311.11795 [cs.PL]

Cassia Torczon, Emmanuel Suarez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. 2024b. *Artifact associated with "Effects and Co-effects in Call-By-Push-Value"*. https://doi.org/10.5281/zenodo.12654518

Verse development team. 2023. *Verse Language Reference.* Epic Games. https://dev.epicgames.com/documentation/en-us/uefn/verse-language-reference.

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

Maxi Wuttke. 2021. *Sound and Relatively Complete Coeffect and effect refinement type systems for call-by-push-value PCF.* Master's thesis. Universität des Sarrlandes.

Received 2024-04-06; accepted 2024-08-18