Lecture 11: Effects and Coeffects in CBPV

Joseph Rotella

October 29, 2025

1 Outline

- Graded effects
- Examples
- Effect-preserving CBPV translation of CBV and CBN
- Coeffects

We've discussed before the notion of an *effect*: broadly speaking, it is something that a program does that changes the environment in some observable way. In this lecture, we will also consider the dual notion of a *coeffect*: that is, a *demand* that a program places on its environment, and how we can statically track such program characteristics in the setting of CBPV.

These notes are based on a guest lecture given by Shubh Agrawal, and portions are directly drawn from their lecture notes. The lecture's content derives from the 2024 paper "Effects and Coeffects in Call-by-Push-Value" by Torczon et al. [1].

2 A Type System with Graded Effects

We begin by enriching our type system with a mechanism by which to statically track effects via our typing judgment. (Algebraic effects and monads are alternative tools for accomplishing this.) In particular, we present a system of *graded effects*: intuitively, this will let us "assign values to effects" by which we may judgmentally track them. While we could, for instance, track *that* a computation prints to the console (e.g., via a writer monad), we may additionally be interested in bounding *how many* characters we print (this will be our running example throughout this presentation). Graded effects enable us to do this.

We begin by modifying our inference rules with annotations that track how effects are sequenced through a program. To admit such annotations, we must modify the shape of our typing judgment for negative types (i.e., computations) so that it now associates to each computation a value φ indicating (in our example) how many characters are printed by that computation. We must also

annotate our type U of suspended computations with the number of characters that will be printed by that computation once forced. Our rules are as follows, with effect annotations highlighted in red (note that this language also includes the type $A\&B$ of lazy pairs, which we have discussed before but did not add to our prior formulation of CBPV; see section 2.1):

$\boxed{\Gamma \vdash v : A^+}$

$$\frac{}{\Gamma_1, x : A, \Gamma_2 \vdash x : A}\ \mathsf{hyp} \qquad \frac{\Gamma \vdash v_1 : A \quad \Gamma \vdash v_2 : B}{\Gamma \vdash (v_1, v_2) : A \times B}\ \times I \qquad \frac{}{\Gamma \vdash () : 1}\ 1I$$

$$\frac{\Gamma \vdash v : A}{\Gamma \vdash \mathsf{inl}\, v : A + B}\ +I_1 \qquad \frac{\Gamma \vdash v : B}{\Gamma \vdash \mathsf{inr}\, v : A + B}\ +I_2 \qquad \frac{\Gamma \vdash e :^{\varphi} A^-}{\Gamma \vdash \mathsf{susp}\, e : U_{\varphi} A^-}\ UI$$

$\boxed{\Gamma \vdash e :^{\varphi} A^-}$

$$\frac{\Gamma, x : A \vdash e :^{\varphi} B}{\Gamma \vdash \lambda x.e :^{\varphi} A \to B}\ \to I \qquad \frac{\Gamma \vdash e :^{\varphi} A \to B \quad \Gamma \vdash v : A}{\Gamma \vdash e\, v :^{\varphi} B}\ \to E$$

$$\frac{\Gamma \vdash v : A \times B \quad \Gamma, x : A, y : B \vdash e :^{\varphi} C}{\Gamma \vdash \mathsf{split}(v, x.y.e) :^{\varphi} C}\ \times E$$

$$\frac{\Gamma \vdash e_1 :^{\varphi} A^- \quad \Gamma \vdash e_2 :^{\varphi} B^-}{\Gamma \vdash \langle e_1, e_2 \rangle :^{\varphi} A^- \& B^-}\ \& I \qquad \frac{\Gamma \vdash e :^{\varphi} A^- \& B^-}{\Gamma \vdash \mathsf{prjl}\, e :^{\varphi} A^-}\ \& E_1 \qquad \frac{\Gamma \vdash e :^{\varphi} A^- \& B^-}{\Gamma \vdash \mathsf{prjr}\, e :^{\varphi} B^-}\ \& E_2$$

$$\frac{\Gamma \vdash v : A + B \quad \Gamma, x : A \vdash e_1 :^{\varphi} C \quad \Gamma, y : B \vdash e_2 :^{\varphi} C}{\Gamma \vdash \mathsf{case}(v, x.e_1, y.e_2) :^{\varphi} C}\ +E \qquad \frac{\Gamma \vdash v : 0}{\Gamma \vdash \mathsf{abort}(v) :^{\varepsilon} C}\ 0E$$

$$\frac{\Gamma \vdash v : U_{\varphi} A^-}{\Gamma \vdash \mathsf{force}\, v :^{\varphi} A^-}\ UE \qquad \frac{\Gamma \vdash v : A^+}{\Gamma \vdash \mathsf{return}\, v :^{\varepsilon} F\, A^+}\ FI \qquad \frac{\Gamma \vdash e :^{\varphi_1} F\, A^+ \quad \Gamma, x : A^+ \vdash e :^{\varphi_2} C}{\Gamma \vdash e\, \mathsf{to}\, x.e' :^{\varphi_1 \cdot \varphi_2} C}\ FE$$

$$\frac{\Gamma \vdash e :^{\varphi'} A^- \quad \varphi' \leq \varphi}{\Gamma \vdash e :^{\varphi} A^-}\ \mathsf{sub\text{-}eff} \qquad \frac{}{\Gamma \vdash \mathsf{print}\, s :^{|s|} F\, 1}\ \mathsf{print}$$

Note that we formulate our effect annotations as ranging over an arbitrary preordered monoid; for our running example here, we'll use the natural numbers, so $\varepsilon = 0$ and $(\cdot) = (+)$.

Several of these rules merit closer inspection. The print, $0E$, and FI rules are the only ones that specify concrete values for the annotation φ: aborting and returning a value are both non-printing computations (and so are labeled with the nullary annotation ε), while print s prints at most (and, indeed, exactly) $|s|$ characters. From these rules alone, it may seem odd that we merely insist that φ be an upper bound (rather than an exact count) or that we include the rule sub-eff. An explanation of this decision is found in the $+E$ rule: while the two branches of a case expression may have differing effects, we insist that the typing judgment be annotated by a single value φ, which must describe the effects of *both* e_1 and e_2. (A similar situation occurs with the rules for lazy pairs.) This then demands the existence of the subeffecting rule sub-eff to allow us to overapproximate the printing behavior of a computation.

Effects are also "combined" in the FE rule, though here we use the multiplication operation of our monoid (here, natural-number addition) to describe the cumulative effect of the computations e and e'. Finally, the UI and UE rules mediate the interchange of an effect annotation between the thunk type U and the computation typing juudgment. The rest of the computation typing rules "thread" existing effect annotations through introduction and elimination forms.

2.1 Interlude: Lazy Pairs

Suppose we want to translate

$$\begin{array}{l}\mathsf{let}\ p = (\mathsf{print}\ \text{“hello”};\ 5, \mathsf{print}\ \text{“world”}; 6)\ \mathsf{in}\\ \mathsf{snd}\ p\end{array}$$

We can translate this pair as eager or lazy.

To translate this pair eagerly, we need to put *values* in the positive pair, which means we are forced to evaluate the pieces first:

$$\begin{array}{l}(\mathsf{print}\ \text{“abc”}\ \mathsf{to}\ \_.\ \mathsf{return}\ 5)\ \mathsf{to}\ p_1.\\ (\mathsf{print}\ \text{“defgh”}\ \mathsf{to}\ \_.\ \mathsf{return}\ 6)\ \mathsf{to}\ p_2.\\ \mathsf{return}\ (p_1, p_2)\ \mathsf{to}\ p.\\ \mathsf{split}\ (p, x.y.y)\end{array}$$

For a lazy translation, we have no way of binding the pair itself, so we have to wrap it in a thunk. It will turn out that we can only bind *after* projecting, which means only one side of it will be evaluated (which is precisely the semantics we would expect of a lazy pair when only one of its components is subsequently used):

$$\begin{array}{l}\mathsf{return}\ \mathsf{susp}\ \langle(\mathsf{print}\ \text{“abc”}\ \mathsf{to}\ \_.\ \mathsf{return}\ 5),\\ \qquad\qquad (\mathsf{print}\ \text{“defgh”}\ \mathsf{to}\ \_.\ \mathsf{return}\ 6)\rangle\ \mathsf{to}\ p.\\ \mathsf{prjr}\ \mathsf{force}\ p\end{array}$$

3 Annotating an Example Program

To concretely demonstrate graded effects, we return to (a slight modification of) our previous example of a program with conflicting call-by-name and call-by-value semantics:

$$\begin{array}{l}\mathsf{let}\ y = \mathsf{print}\ \text{“hello”};\ 3\ \mathsf{in}\\ \mathsf{let}\ f = \lambda x : \mathbb{Z}.\ \mathsf{print}\ \text{“worlds”};\ x\ \mathsf{in}\\ (f\ y, f\ (y+1))\end{array}$$

Recall that, while this program always evaluates to $(3, 4)$, it prints differing output in the call-by-name and call-by-value settings:

- **CBV**: helloworldsworlds

- **CBN**: worldshelloworldshello

In particular, it prints differing numbers of characters to the console: 17 under CBV, and 22 under CBN. We can see this by applying our system of graded effects to our call-by-value and call-by-name translations of this program. Below is the CBV translation, annotated where effects occur:

$$
\begin{array}{l}
(\mathsf{print}\ \text{“hello”}\ \mathsf{to}\ \_.\ \mathsf{return}\ 3)^{5}\ \mathsf{to}\ y. \\
\mathsf{return}\ \mathsf{susp}\ (\lambda x : \mathbb{Z}.\ \mathsf{print}\ \text{“worlds”}\ \mathsf{to}\ \_.\ \mathsf{return}\ x)\ \mathsf{to}\ f. \\
(\mathsf{force}\ f)^{6}\ y\ \mathsf{to}\ r_1. \\
(\mathsf{force}\ f)^{6}\ (y+1)\ \mathsf{to}\ r_2. \\
\mathsf{return}\ (r_1, r_2)
\end{array}
$$

Observe that the print on the first line immediately prints 5 characters, while the thunk $f : U_6(\mathbb{Z} \to F\mathbb{Z})$ suspends a 6-character-printing computation that is then forced twice.

Now consider the CBN translation:

$$
\begin{array}{l}
\mathsf{return}\ (\mathsf{susp}\ (\mathsf{print}\ \text{“hello”}\ \mathsf{to}\ \_.\ \mathsf{return}\ 3))\ \mathsf{to}\ y. \\
\mathsf{return}\ (\mathsf{susp}\ (\lambda x : U_6(F\mathbb{Z}).\ \mathsf{print}\ \text{“worlds”}\ \mathsf{to}\ \_.\ \mathsf{force}\ x))\ \mathsf{to}\ f. \\
((\mathsf{force}\ f)\ y)^{11}\ \mathsf{to}\ r_1. \\
((\mathsf{force}\ f)\ (y+1))^{11}\ \mathsf{to}\ r_2. \\
\mathsf{return}\ (r_1, r_2)
\end{array}
$$

In this translation, y is now also a thunk, namely of type $U_5(F\mathbb{Z})$. Thus, its effect is not incurred until it is (twice) forced by the suspended body of f. The above program also illustrates an interesting characteristic of this calculus: we do not have “effect-polymorphic” lambdas and so must decide *a priori* the “printing budget” we will afford an argument to (the forcing of) f: this is encoded by the type $U_6(F\mathbb{Z})$ of its parameter.

4 CBV and CBN Translations

Translating from CBV (with graded effects) into our CBPV calculus is no more difficult than in the absence of graded effects: we can simply reuse our translation from last time. We will then have that whenever $\Gamma \vdash m :^{\varphi} A$, it is the case that $\ulcorner\Gamma\urcorner \vdash \ulcorner m\urcorner :^{\varphi} F\ulcorner A\urcorner$. Adding effect annotations to the lambda calculus is straightforward because it is eager, so we just write down whatever effects occur in a given term. In general, there will be terms in which multiple effects occur and are sequenced—in the term translation, those will all translate into our F-elimination.

Translating from CBN is more complicated because effects in CBN are themselves more complicated. It is hard to annotate terms with effects because it is hard to locally determine whether the effects will actually occur (since this will depend on how—or if—the term is subsequently used). The typical approach

here is to introduce a *monad* M, which will allow us to internalize the effect annotations we previously introduced in our CBPV typing judgment:

$$\frac{\Gamma \vdash m : A}{\Gamma \vdash \mathsf{return}\ m : M_{\varepsilon} A}\ MI$$

$$\frac{\Gamma \vdash m_1 : M_{\varphi_1} A \quad \Gamma, x : A \vdash m_2 : M_{\varphi_2} B}{\Gamma \vdash \mathsf{bind}\ x \leftarrow m_1.\ m_2 : M_{\varphi_1 \cdot \varphi_2} B}\ ME$$

Notice that the elimination rule requires that the second expression *also* be of monadic type (cf. FE, which does not). In this way, monadic type systems tend to lead to monads infecting all your code.

As before, the CBN translation takes a type A to a CBPV computation type A^- and terms of type A to computations of type A^-. In particular, it sends these terms to *pure* computations. So, if $\Gamma \vdash m : A$, then $\ulcorner\Gamma\urcorner \vdash \ulcorner m\urcorner :^{\varepsilon} \ulcorner A\urcorner$. The translation proceeds as follows:

$$\begin{aligned}
\ulcorner M_{\varphi} A\urcorner &= F(U_{\varphi}(F(U_{\varepsilon}\ulcorner A\urcorner)))\ldots \\
\ulcorner \mathsf{return}\ m\urcorner &= \mathsf{return}\ (\mathsf{susp}\ (\mathsf{return}\ (\mathsf{susp}\ \ulcorner m\urcorner))) \\
\ulcorner \mathsf{bind}\ x\ \mathsf{from}\ m_1\ .\ m_2\urcorner &= \mathsf{return}\ (\mathsf{susp}\ (\\
&\qquad (\ulcorner m_1\urcorner\ \mathsf{to}\ y.\ \mathsf{force}\ y)\ \mathsf{to}\ x. \\
&\qquad \ulcorner m_2\urcorner\ \mathsf{to}\ z. \\
&\qquad \mathsf{force}\ z))
\end{aligned}$$

$U \circ F$ is the monad—the other parts bracket it. This makes sense: we think of a monad as a suspended computation that returns the inside type. But the inside type needs to be turned into a value. Then the whole thing needs to be a computation, so we are ultimately returning that.

5 Generalizing Effect Structure

We previously remarked that our effect annotations can range over any preordered monoid. Why does this exhibit the structural properties we need to capture the desired properties of effects? We need to be able to capture:

- The empty effect, for returning
- A sequencing operation
- An ordering operation for approximation/subeffecting

These desiderata translate directly into the laws for a preordered monoid:

- Identity: $\forall \varphi,\ \varepsilon \cdot \varphi = \varphi = \varphi \cdot \varepsilon$
- Associativity: $\forall \varphi_1\, \varphi_2\, \varphi_3,\ \varphi_1 \cdot (\varphi_2 \cdot \varphi_3) = (\varphi_1 \cdot \varphi_2) \cdot \varphi_3$

- Respect for ordering: $\forall \varphi_1\, \varphi_1'\, \varphi_2,\ \varphi_1 \leq \varphi_1' \implies \varphi_1 \cdot \varphi_2 \leq \varphi_1' \cdot \varphi_2$

We can also specialize this construction in various ways. For instance, replacing the preorder with equality yields a *precise* ordering: you know exactly what effect will happen. The trade-off, as we have seen, is that many branching programs will no longer type-check. Another interesting example is taking our monoid to be a powerset with set union, using set inclusion as the preorder. This lets us reason about a *set* of possible effects (throwing, diverging, writing, ...) that can be unioned together.

6 Coeffects

Coeffects are about *resource usage*. The idea is that our program somehow makes demands on its environment. (Note that, in colloquial usage, we may sometimes refer to such behaviors as "effects!") For instance, we may be interested in tracking the number of times we use a variable: this will be our running example for this section.

Our rules are as follows:

$$\boxed{\gamma\Gamma \vdash v : A^+}$$

$$\frac{}{\overline{0}\Gamma_1, x :^1 A^+, \overline{0}\Gamma_2 \vdash x : A^+}\ \mathsf{hyp} \qquad \frac{\gamma_1\Gamma \vdash v_1 : A \quad \gamma_2\Gamma \vdash v_2 : B}{(\gamma_1 + \gamma_2)\Gamma \vdash (v_1, v_2) : A \times B}\ \times I \qquad \frac{}{\overline{0}\Gamma \vdash () : 1}\ 1I$$

$$\frac{\gamma\Gamma \vdash v : A^+}{\gamma\Gamma \vdash \mathsf{inl}\, v : A^+ + B^+}\ +I_1 \qquad \frac{\gamma\Gamma \vdash v : B^+}{\gamma\Gamma \vdash \mathsf{inr}\, v : A^+ + B^+}\ +I_2 \qquad \frac{\gamma\Gamma \vdash e : A^-}{\gamma\Gamma \vdash \mathsf{susp}\, e : U\, A^-}\ UI$$

$$\frac{\gamma\Gamma \vdash v : A^+ \quad \gamma' \leq_{\mathrm{co}} \gamma}{\gamma'\Gamma \vdash v : A^+}\ \mathsf{sub}$$

$\boxed{\gamma\Gamma \vdash e : A^-}$

$$\frac{\gamma\Gamma, x :^q A \vdash e : B \quad q' \leq_{\text{co}} q}{\gamma\Gamma \vdash \lambda^q x.e : A^{q'} \to B} \to I \qquad \frac{\gamma_1\Gamma \vdash e : A^q \to B \quad \gamma_2\Gamma \vdash v : A}{(\gamma_1 + q \cdot \gamma_2)\Gamma \vdash e\, v : B} \to E$$

$$\frac{\gamma_1\Gamma \vdash v : A \times B \quad \gamma_2\Gamma, x :^q A, y :^q B \vdash e : C}{(q \cdot \gamma_1 + \gamma_2)\Gamma \vdash \mathsf{split}_q(v, x.y.e) : C} \times E$$

$$\frac{\gamma\Gamma \vdash e_1 : A^- \quad \gamma\Gamma \vdash e_2 : B^-}{\gamma\Gamma \vdash \langle e_1, e_2\rangle : A^- \& B^-} \& I \qquad \frac{\gamma\Gamma \vdash e : A^- \& B^-}{\gamma\Gamma \vdash \mathsf{prjl}\, e : A^-} \& E_1 \qquad \frac{\gamma\Gamma \vdash e : A^- \& B^-}{\gamma\Gamma \vdash \mathsf{prjr}\, e : B^-} \& E_2$$

$$\frac{\gamma\Gamma \vdash v : A + B \quad \gamma'\Gamma, x :^q A \vdash e_1 : C \quad \gamma'\Gamma, y :^q B \vdash e_2 : C \quad q \leq_{\text{co}} 1}{(q \cdot \gamma + \gamma')\Gamma \vdash \mathsf{case}_q(v, x.e_1, y.e_2) : C} + E$$

$$\frac{\gamma\Gamma \vdash v : 0}{\gamma\Gamma \vdash \mathsf{abort}(v) : C}\ 0E \qquad \frac{\gamma\Gamma \vdash v : U\, A^-}{\gamma\Gamma \vdash \mathsf{force}\, v : A^-}\ UE$$

$$\frac{\gamma\Gamma \vdash v : A^+}{q \cdot \gamma\Gamma \vdash \mathsf{return}_q v : F_q\, A^+}\ FI \qquad \frac{\gamma\Gamma \vdash e : F_{q_1}\, A^+ \quad \gamma'\Gamma, x :^{q_1 \cdot q_2} A^+ \vdash e' : C}{(q_2 \cdot \gamma + \gamma')\Gamma \vdash e\ \mathsf{to}_{q_2}\ x.e' : C}\ FE$$

Note that we need a bit more structure on coeffect annotations than for effect annotations: to fully capture resource management—splitting, moving around, and multiplying resources—we require the richer structure of a *semiring*.

We illustrate this system's usage with an example program, written as call-by-value and call-by-name variants:

CBV	**CBN**
$\mathsf{return}_2\ z + z\ \mathsf{to}\ y.$	$\mathsf{return}_2\ \mathsf{susp}\ (z + z)\ \mathsf{to}\ y.$
$\mathsf{return}_2\ \mathsf{susp}\ (\lambda x.$	$\mathsf{return}_2\ \mathsf{susp}\ (\lambda x.\ \mathsf{force}\ x\ \mathsf{to}_3\ x'.$
$\quad \mathsf{return}\ x + x + x)\ \mathsf{to}\ f.$	$\quad \mathsf{return}\ x' + x' + x')\ \mathsf{to}\ f.$
$(\mathsf{force}\ f)\ y\ \mathsf{to}\ r_1.$	$(\mathsf{force}\ f)\ y\ \mathsf{to}\ r_1.$
$(\mathsf{force}\ f)\ (y + 1)\ \mathsf{to}\ r_2.$	$(\mathsf{force}\ f)\ (y + 1)\ \mathsf{to}\ r_2.$
$\mathsf{return}\ (r_1, r_2)$	$\mathsf{return}\ (r_1, r_2)$

The presentation we have given in this section is an example of *structured coeffects*, in which coeffects are associated with variables. It is also possible to formulate *flat coeffects*, which give a single annotation to an entire context.

References

[1] Torczon, C., Suárez Acevedo, E., Agrawal, S., Velez-Ginorio, J., and Weirich, S. Effects and coeffects in call-by-push-value. *Proc. ACM Program. Lang. 8*, OOPSLA2 (Oct. 2024).