Lecture Notes on Call-by-Push-Value

15-814: Types and Programming Languages
Hemant Gouni
Notes by Frank Pfenning

Lecture 23
Thu Dec 4, 2025

Throughout the course we have emphasized the distinction between types whose values can be directly observed and types whose values can be observed only indirectly, by interacting with them (for example, applying a function or projecting from a lazy record). Levy's work [2001, 2006] on call-by-push-value elevates this distinction to first-class status. Because in proof theory, directly observable types are called *positive* and others *negative* we say that the language is explicitly *polarized*.

In addition he formulates a dynamics for the language in the form of an abstract machine with a stack, which is why he called his paradigm *call-by-push-value* (as distinct from *call-by-value* or *call-by-name*). We will not show the stack machine underlying call-by-push-value, preferring to remain within our current framework.

One of the consequences of polarization is that it makes evaluation order explicit in the syntax, which in turn leads to an elegant way to incorporate computational effects such as input, output, or mutable storage. This simplifies reasoning about programs (although we do not cover this aspect in this lecture, either). For more on effects, a good source is a set of lecture notes [Harper, 2025]; for uses of call-by-push-value in a compiler see Jiang et al. [2025]. Both of these also contain further references.

Call-by-push-value *subsumes* call-by-value and call-by-name in the sense that there are simple compositional translations from call-by-value and call-by-name.

1 Polarizing the Syntax

In the Levy's parlance, we split the syntax into *values* (observables) and *computations* (non-observables). This is a slight variance of our previous terminology, as we will explain.

$$\begin{array}{lrcl} \text{Value types} & A & ::= & A_1 \times A_2 \mid 1 \mid \Sigma_{\ell \in L}(\ell : A_\ell) \mid \mathsf{U}\underline{B} \\ \text{Computation types} & \underline{B} & ::= & A \to \underline{B} \mid \&_{\ell \in L}(\ell : \underline{B}_\ell) \mid \mathsf{F}A \\ \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A \end{array}$$

We omit recursive, universal, and existential types merely for the sake of brevity. The new constructs $\mathsf{U}\underline{B}$ and $\mathsf{F}A$ serve to include values in computations and vice versa. That allows us to suspend a computation, or for a computation return a value. The rules for them will require special attention.

Because we want to make evaluation order fully explicit (including the case when we have effects), variables range only over values. Further, we have two judgments: one for typing values and one for typing computations.

2 Values

Values are pretty straightforward, since they mirror what we had so far for observable in our call-by-value language. One difference, though: variables count as values, because they stand for values when programs execute.

$$\text{Values} \quad V \quad ::= \quad x \mid \langle V_1, V_2 \rangle \mid \langle\,\rangle \mid k \cdot V \mid \mathbf{susp}\ M$$

In a sense, the only interesting part is here is the suspension of a computation M counts as a value. The other part is that the destructors for these values are computations. Important is that the subject of each case construct must be a value. This only make sense because values include variables, otherwise all destructors could be statically reduced.

Pairs. The destructor references the typing judgment $\Gamma \vdash M : \underline{B}$ for computations M. The destructors for values have a single reduction rule because, well, values are values and this don't reduce.

$$\frac{\Gamma \vdash V_1 : A_1 \quad \Gamma \vdash V_2 : A_2}{\Gamma \vdash \langle V_1, V_2 \rangle : A_1 \times A_2}\ \mathsf{tpv/pair} \qquad \frac{\Gamma \vdash V : A_1 \times A_2 \quad \Gamma, x_1 : A_1, x_2 : A_2 \vdash M : \underline{B}}{\Gamma \vdash \mathbf{case}\ V\ (\langle x_1, x_2 \rangle \Rightarrow M) : \underline{B}}\ \mathsf{tpc/case/pair}$$

$$\frac{}{\mathbf{case}\ \langle V_1, V_2 \rangle\ (\langle x_1, x_2 \rangle \Rightarrow M) \mapsto [V_1/x_1, V_2/x_2]M}\ \mathsf{step/pair}$$

Unit.

$$\frac{}{\Gamma \vdash \langle\,\rangle : 1}\ \mathsf{tpv/unit} \qquad \frac{\Gamma \vdash V : 1 \quad \Gamma \vdash M : \underline{B}}{\Gamma \vdash \mathbf{case}\ V\ (\langle\,\rangle \Rightarrow M) : \underline{B}}\ \mathsf{tpc/case/unit}$$

$$\frac{}{\mathbf{case}\ \langle\,\rangle\ (\langle\,\rangle \Rightarrow M) \mapsto M}\ \mathsf{step/unit}$$

Sums.

$$\frac{(k \in L) \quad \Gamma \vdash V : A_k}{\Gamma \vdash k \cdot V : \Sigma_{\ell \in L}(\ell : A_\ell)}\ \mathsf{tpv/inj} \qquad \frac{\Gamma \vdash V : \Sigma_{\ell \in L}(\ell : A_\ell) \quad (\Gamma, x_\ell : A_\ell \vdash M_\ell : \underline{B}) \quad (\forall \ell \in L)}{\Gamma \vdash \mathbf{case}\ V\ (\ell \cdot x_\ell \Rightarrow M_\ell)_{\ell \in L} : \underline{B}}\ \mathsf{tpv/case/inj}$$

$$\frac{}{\mathbf{case}\ (k \cdot V)\ (\ell \cdot x_\ell \Rightarrow M_\ell)_{\ell \in L} \mapsto [V/x_k]M_k}\ \mathsf{step/inj}$$

Suspensions. Suspensions, the constructors for type $\mathsf{U}\underline{B}$, break the pattern we have so far because the destructor is not a **case** expression. Instead we simply force a suspension to obtain a computation. This is because $\mathsf{U}\underline{B}$ is the only *unobservable*—or negative—value type, including computations into values.

$$\frac{\Gamma \vdash M : \underline{B}}{\Gamma \vdash \mathbf{susp}\ M : \mathsf{U}\underline{B}}\ \mathsf{tpv/susp} \qquad \frac{\Gamma \vdash V : \mathsf{U}\underline{B}}{\Gamma \vdash V.\mathbf{force} : \underline{B}}\ \mathsf{tpc/force}$$

$$\frac{}{(\mathbf{susp}\ M).\mathbf{force} \mapsto M}\ \mathsf{step/susp}$$

3 Computations

The language of computations (aside from destructors for values) is richer than that of values. An interesting twist is that we have *terminal computations*, that is, computations that cannot be further reduced. However, they are still computations in the sense that we cannot observe them directly. We show not only the typing, but also the stepping rules for each computation type.

Functions. The type $A \to \underline{B}$ indicates that the argument to a function should be a value, but the application should be computation.

$$\frac{\Gamma, x : A \vdash M : \underline{B}}{\Gamma \vdash \lambda x.\, M : A \to \underline{B}}\ \mathsf{tpc/lam} \qquad \frac{\Gamma \vdash M : A \to \underline{B} \quad \Gamma \vdash V : A}{\Gamma \vdash M\, V : \underline{B}}\ \mathsf{tpc/app}$$

$$\frac{}{(\lambda x.\, M)\, V \mapsto [V/x]M}\ \mathsf{step/lam} \qquad \frac{M \mapsto M'}{M\, V \mapsto M'\, V}\ \mathsf{step/app}$$

Note the symmetry between the rules: x stands for a value, so $x : A$ is added to the context in the rule tpc/lam. Correspondingly, the argument of the function is a value. Even though a λ-abstraction does not step on its own, it steps when applied to an argument (which must be a value).

Lazy Records. There are no surprises here, following the pattern for functions.

$$\frac{(\Gamma \vdash M_\ell : \underline{B}_\ell) \quad (\forall \ell \in L)}{\Gamma \vdash \langle\!|\ell \Rightarrow M_\ell|\!\rangle_{\ell \in L} : \Sigma_{\ell \in L}(\ell : \underline{B}_\ell)}\ \mathsf{tpc/record} \qquad \frac{\Gamma \vdash M : \&_{\ell \in L}(\ell : \underline{B}_\ell) \quad (k \in L)}{\Gamma \vdash M.k : \underline{B}_k}\ \mathsf{tpc/proj}$$

$$\frac{(k \in L)}{\langle\!|\ell \Rightarrow M_\ell|\!\rangle_{\ell \in L}.k \mapsto M_k}\ \mathsf{step/record} \qquad \frac{M \mapsto M'}{M.k \mapsto M'.k}\ \mathsf{step/proj}$$

Return and Bind. The transition from values to computations is the most interesting part; $\mathsf{F}A$ is the only *observable*—or positive—computation type, including values into computations.

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathbf{ret}\ V : \mathsf{F}A}\ \mathsf{tpc/ret} \qquad \frac{\Gamma \vdash M_1 : \mathsf{F}A \quad \Gamma, x : A \vdash M_2 : \underline{B}}{\Gamma \vdash \mathbf{bind}\ M_1\ (\mathbf{ret}\, x \Rightarrow M_2) : \underline{B}}\ \mathsf{tpc/bind}$$

$$\frac{}{\mathbf{bind}\ (\mathbf{ret}\ V_1)\ (\mathbf{ret}\, x \Rightarrow M_2) \mapsto [V_1/x]M_2}\ \mathsf{step/ret}$$

$$\frac{M_1 \mapsto M_1'}{\mathbf{bind}\ M_1\ (\mathbf{ret}\, x \Rightarrow M_2) \mapsto \mathbf{bind}\ M_1'\ (\mathbf{ret}\, x \Rightarrow M_2)}\ \mathsf{step/bind}$$

Bind is the only computation that references two others, but since the second is under a binding for a variable x, it cannot be reduced unless the subject of the bind is a return. This preserves the simplicity of the reduction relation. We don't write the destructor as a **case** expression because the subject is a computation, not a value. But its action is nevertheless similar to a **case** expression.

Value types	A	::=	$A_1 \times A_2 \mid 1 \mid \Sigma_{\ell \in L}(\ell : A_\ell) \mid \mathsf{U}\underline{B}$	
Computation types	$\underline{B}$	::=	$A \to \underline{B} \mid \&_{\ell \in L}(\ell : \underline{B}_\ell) \mid \mathsf{F}A$	
Contexts	Γ	::=	$\cdot \mid \Gamma, x : A$	
Values	V	::=	x	
		$\mid$	$\langle V_1, V_2 \rangle$	$(A_1 \times A_2)$
		$\mid$	$\langle\,\rangle$	(1)
		$\mid$	$k \cdot V$	$(\Sigma_{\ell \in L}(\ell : A_\ell))$
		$\mid$	$\mathbf{susp}\ M$	$(\mathsf{U}\underline{B})$
Computations	M	::=	$\mathbf{case}\ V\ (\langle x_1, x_2 \rangle \Rightarrow M)$	$(A_1 \times A_2)$
		$\mid$	$\mathbf{case}\ V\ (\langle\,\rangle \Rightarrow M)$	(1)
		$\mid$	$\mathbf{case}\ V\ (\ell \cdot x_\ell \Rightarrow M_\ell)_{\ell \in L}$	$(\Sigma_{\ell \in L}(\ell : A_\ell)$
		$\mid$	$M.\mathbf{force}$	$(\mathsf{U}\underline{B})$
		$\mid$	$\lambda x.\, M \mid M\, V$	$(A \to \underline{B})$
		$\mid$	$\{\!\{\ell \Rightarrow M_\ell\}\!\}_{\ell \in L} \mid M.k$	$(\&_{\ell \in L}(\ell : \underline{B}_\ell))$
		$\mid$	$\mathbf{ret}\ V \mid \mathbf{bind}\ M_1\ (\mathbf{ret}\ x \Rightarrow M_2)$	$(\mathsf{F}A)$

For this language (even if extended with recursive, universal, and existential types) we obtain the expected canonical forms, progress, preservation, and sequentiality theorems. Because they follow established patterns, they are not particularly interesting.

4 Effects

For the three form of computation types we have *terminal computations* $\lambda x.\, M$, $\{\!\{\ell \Rightarrow M_\ell\}\!\}_{\ell \in L}$, and $\mathbf{ret}\, V$. We postulate that we cannot observe computations, so here this includes terminal computations. This creates a bit of an issue, because overall we want to run a computation and see the result. So, at the top level, we think of running a computation $M : \mathsf{F}A$ and observe the result $\mathbf{ret}\, V$. Conversely, even though $\mathbf{susp}\ M : \mathsf{U}\underline{B}$ is a value, we do not think of M as being observable. In a sense, $\mathsf{F}A$ and $\mathsf{U}\underline{B}$ open or close a window to observability.

One reason this matters is because how we treat effects. As a sample effect we add a value type string and a computation with an effect $\mathbf{print}\ V$ where V is of type string. In the dynamics, we need to add an output O to the stepping judgment we represents the sequence strings that were printed. Printing returns the unit element and appends the string to the outputs, as an effect.

$$\frac{\Gamma \vdash V : \mathsf{string}}{\Gamma \vdash \mathbf{print}\ V : \mathsf{F1}}\ \mathsf{tp/print}$$

$$\frac{}{\mathbf{print}\ V \mid O \mapsto \mathbf{ret}\,\langle\,\rangle \mid O, V}\ \mathsf{step/print}$$

The output O is now threaded through the computation rules, remaining unchanged except for the printing rule. We show only the rules for **ret** and **bind**.

$$\frac{}{\mathbf{bind}\ (\mathbf{ret}\ V_1)\ (\mathbf{ret}\ x \Rightarrow M_2) \mid O \mapsto [V_1/x]M_2 \mid O}\ \mathsf{step/ret}$$

$$\frac{M_1 \mid O \mapsto M_1' \mid O'}{\mathbf{bind}\ M_1\ (\mathbf{ret}\ x \Rightarrow M_2) \mid O \mapsto \mathbf{bind}\ M_1'\ (\mathbf{ret}\ x \Rightarrow M_2) \mid O'}\ \mathsf{step/bind}$$

Because computations are not observable, we can now obtain some surprising equivalences. For example, the two computations

$$\lambda x.\, \mathbf{bind}\ (\mathbf{print}\ \text{“first”})\ (\mathbf{ret}\, u \Rightarrow \mathbf{ret}\, u) : 1 \to \mathsf{F}\, 1$$
$$\mathbf{bind}\ (\mathbf{print}\ \text{“first”})\ (\mathbf{ret}\, u \Rightarrow \lambda x.\, \mathbf{ret}\, u) : 1 \to \mathsf{F}\, 1$$

may be considered equivalent because the output cannot be observed until the computation has “reached” $\mathsf{F}1$, that is, the function has been applied to the unit element (and that only at the top level).

We can capture this notion of equality in two logical relations, one from values and one for computations. Because terminal computations aren't values, the overall structure is a bit different from our previous call-by-value logical relation, but there are still plenty of similarities.

$$\begin{array}{lcl}
V = V' \in [A_1 \times A_2] & \iff & V = \langle V_1, V_2\rangle \wedge V' = \langle V_1', V_2'\rangle \\
 & & \quad \wedge\, V_1 = V_1' \in [A_1] \wedge V_2 = V_2' \in [A_2] \\
V = V' \in [1] & \iff & V = \langle\,\rangle \wedge V' = \langle\,\rangle \\
V = V' \in [\Sigma_{\ell \in L}(\ell : A_\ell)] & \iff & V = k \cdot V_1 \wedge V' = k \cdot V_1' \wedge V_1 = V_1' \in [A_k] \\
V = V' \in [\mathsf{U}\underline{B}] & \iff & V.\mathbf{force} = V'.\mathbf{force} \in [\![\underline{B}]\!] \\
 & & \\
M = M' \in [\![A \to \underline{B}]\!] & \iff & M\, V = M'\, V' \in [\![\underline{B}]\!] \quad \text{for all } V = V' \in [A] \\
M = M' \in [\![\&_{\ell \in L}(\ell : \underline{B}_\ell)]\!] & \iff & M.\ell = M'.\ell \in [\![\underline{B}_\ell]\!] \quad \text{for all } \ell \in L \\
M = M' \in [\![\mathsf{F}A]\!] & \iff & M \mid \cdot \mapsto^* \mathbf{ret}\, V \mid O \wedge M' \mid \cdot \mapsto^* \mathbf{ret}\, V' \mid O' \\
 & & \quad \wedge\, O = O' \wedge V = V' \in [A]
\end{array}$$

Under this definition the two programs shown above will be logically equal computations. That's because we don't consider reduction and judge the effects (here: output) until we reach a computation of type $\mathsf{F}A$. That's okay here, because other computations are not observable.

With this definition we can prove the fundamental theorem, namely that well-typed values and computations are logically related to themselves.

References

Robert Harper. Effects in call-by-push-value, Spring 2025. URL https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/effects.pdf. Lecture Notes.

Yuchen Jiang, Runze Xue, and Max S. New. Notions of stack-manipulating computation and relative monads. *Proceedings of the ACM on Programming Languages*, 9 (OOPSLA1):563–589, 2025.

Paul Blain Levy. *Call-by-Push-Value*. PhD thesis, University of London, 2001.

Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006.