Lecture Notes on Polarized Types

15-417/817: HOT Compilation
Frank Pfenning

Lecture 22
April 15, 2025

1 Introduction

Throughout the semester we have been talking about the polarity of types. Fundamentally, we can observe values of positive types but we can only interact with values of negative types. While this distinction is reflected in the dynamics of ND and Sax, it is not made explicit in the types themselves. Polarization does just that.

Polarization [Girard, 1991] in its most straightforward incarnation explicitly divides the language of types into positive and negative ones, requiring explicit modal operators to shift between them.

There are many consequences of this decision, both proof-theoretic and computational. On the proof-theoretic side, we obtain a system of focusing [Andreoli, 1992, Liang and Miller, 2009, Simmons, 2014] which in turn is the basis of logical frameworks such as CLF ([Watkins et al., 2002, Cervesato et al., 2002, Schack-Nielsen, 2011], used in Lecture 24 to formalize linear Sax). We have also already used this as the basis for pattern matching in Lecture 4. Polarization gives rise to logic programming engines [López et al., 2005] and efficient theorem provers [McLaughlin and Pfenning, 2009] because it allows us to perform "big-step inferences" while remaining logically complete [Andreoli, 2001].

On the computational side (or, to be more precise, on the side of computation-as-proof-reduction) we obtain call-by-push-value (CBPV) [Levy, 2001, 2006]. Call-by-push-value was conceived as a unifying framework for call-by-value and call-by-name, rich enough to also encompass effects. Levy's *values* are all of positive type, his *computations* are all of negative type, where he distinguishes *terminal computations* as what we have called non-observable values. The slogan is *values are, and computations do*.

We do not give a full account of call-by-push-value, which is extensively covered in the references. Instead, we reimagine it in the context of this course.

2 Basic Polarized Types

We stratify the language of types into *value types* (which are positive) and *computation types* (which are negative).

Value types	A, B	$::=$	$A \times B \mid \mathbf{1} \mid +\{\ell : A_\ell\}_{\ell \in L} \mid \uparrow \underline{A}$
Computation types	$\underline{A}, \underline{B}$	$::=$	$A \to \underline{B} \mid \&\{\ell : \underline{A}_\ell\}_{\ell \in L} \mid \downarrow A$
Contexts	Γ	$::=$	$x : A \mid \Gamma_1, \Gamma_2 \mid (\cdot)$

Because the design of ND was significantly influenced by call-by-push-value, the meaning of all types remains what is has been for us all along. For example, $\uparrow\underline{A}$ is inhabited by suspended computations (which count as values), while $\downarrow A$ represents the trivial computation consisting of a value. We also restrict the context to assign only positive types to variables, which therefore stand for values.

Here is a small table with the mapping of notations.

ND	CBPV
$\uparrow\underline{A}$	$U\ \underline{A}$
$\downarrow A$	$F\ A$
$+\{\ell : A_\ell\}_{\ell\in L}$	$\sum_{\ell\in L} A_\ell$
$\&\{\ell : A_\ell\}_{\ell\in L}$	$\prod_{\ell\in L} \underline{A}_\ell$

If we think of value types as being at one mode (say v) and computation types being at another mode (say c), the the shifts are implicitly decorated as $\uparrow^{\mathsf{v}}_{\mathsf{c}}\underline{A}$ and $\downarrow^{\mathsf{v}}_{\mathsf{c}}A$. The only surprising thing here may be that the function type is of mixed mode:

$$A \to \underline{B} \sim A_{\mathsf{v}} \to \underline{B}_{\mathsf{c}}$$

Because polarization is based on the proof-theoretic properties of the types viewed as propositions, there is not much choice here. We know that implication $A \supset B$ is right invertible and negative. In order to continue inversion, since means A needs to be positive so it is left invertible. And we want to continue inversion because any change with respect to inversion should only be due to embedded shifts. If we encode this as

$$A \to \underline{B} \triangleq (\downarrow A) \to \underline{B} \qquad ??$$

its meaning would change: rather than a function taking as argument a *value* of type A it is a function taking as argument a *computation* of type $\downarrow A$, that is, a computation returning a value of type A. When and how should such a computation be executed? Especially in the presence of effects, there is no clear answer to this, so such a definition would break the call-by-push-value discipline.

So in call-by-push-value, we have a binary connective with the two operands being at different mode, a so-called *skew* connective. We can do a quick check if it actually makes sense to have such a connective from a proof-theoretic perspective, that is, in the sequent calculus. We write $A_k \to B_m$ for a skew proposition (that is, type) at mode m.

$$\frac{\Gamma, A_k \vdash B_m}{\Gamma \vdash A_k \to B_m}\ {\to}R \qquad \frac{\Gamma \geq k \quad \Gamma \vdash A_k \quad \Delta, B_m \vdash C_r}{\Gamma\,;\,\Delta\,;\,A_k \to B_m \vdash C_r}\ {\to}L$$

First, we observe that for the right rule to be invertible (as it should be) we need $k \geq m$ in the dependency order. Second, we need the condition $\Gamma \geq k$ in the $\to L$ rule in order to maintain independence. Nothing needs to be checked for the second premise since $\Delta \geq r$ and $m \geq r$ are known by presupposition of independence for the conclusion.

Now recall the general rule of cut.

$$\frac{\Gamma \geq m \geq r \quad \Gamma \vdash A_m \quad \Delta, A_m \vdash C_r}{\Gamma\,;\,\Delta \vdash C_r}\ \mathsf{cut}$$

We can form the particular cut where $\to R$ meets $\to L$.

$$\dfrac{\Gamma \geq m \geq r \quad \dfrac{\overset{\mathcal{D}'}{\Gamma, A_k \vdash B_m}}{\Gamma \vdash A_k \to B_m}\to R \quad \dfrac{\Delta_1 \geq k \quad \overset{\mathcal{E}_1}{\Delta_1 \vdash A_k} \quad \overset{\mathcal{E}_2}{\Delta_2, B_m \vdash C_r}}{\Delta_1\,;\,\Delta_2\,;\,A_k \to B_m \vdash C_r}\to L}{\Gamma\,;\,\Delta_1\,;\,\Delta_2 \vdash C_r}\,\mathsf{cut}$$

$$\longrightarrow$$

$$\dfrac{(\Gamma\,;\,\Delta_1) \geq m \geq r \quad \dfrac{\Delta_1 \geq k \geq m \quad \overset{\mathcal{E}_1}{\Delta_1 \vdash A_k} \quad \overset{\mathcal{D}'}{\Gamma, A_k \vdash B_m}}{\Gamma\,;\,\Delta_1 \vdash B_m}\,\mathsf{cut} \quad \overset{\mathcal{E}_2}{\Delta_2, B_m \vdash C_r}}{\Gamma\,;\,\Delta_1\,;\,\Delta_2 \vdash C_r}\,\mathsf{cut}$$

The only possible question here is $k \geq m$ in the condition of the first cut, but that is guaranteed by the presupposition that $A_k \to B_m$ is well-formed.

So, yes, the skew implication $A_k \to B_m$ connective is a first-class connective if $k \geq m$, and one may only object on the grounds of minimalism since there is a logically equivalent formula $(\downarrow_m^k A_k) \to B_m$ that is not skew. But, as discussed above, in the setting of call-by-push-value those two do not mean the same thing.

3 Polarized Expressions

The language of expressions splits into (large) values v and computations e, with the judgments

$$\Gamma \vdash v : A$$
$$\Gamma \vdash e : \underline{A}$$

where Γ consists entirely of value typings $x : A$. The mode discipline would allow variables $x : \underline{A}$ to range over computation in the second judgment, but this is ruled out basically because of the possible presence of effects in computations. We will talk about this a bit more in the next lecture on the Enriched Effect Calculus (EEC), and below when we discuss the dynamics.

We have the following syntax, taken from ND rather than from CBPV.

Values	v	::=	(v_1, v_2)	$(A \times B)$
		|	$()$	$(\mathbf{1})$
		|	$k(v)$	$(+\{\ell : A_\ell\})$
		|	$\mathbf{susp}\ e$	$(\uparrow\underline{A})$
Computations	e	::=	$\lambda x.\, e \mid e\, v$	$(A \to \underline{B})$
		|	$\{\ell \Rightarrow e_\ell\}_{\ell \in L} \mid e.k$	$(\&\{\ell : A_\ell\})$
		|	$v.\mathbf{force}$	$(\uparrow\underline{A})$
		|	$\langle v \rangle \mid \mathbf{match}\ e\ (\langle x \rangle \Rightarrow e')$	$(\downarrow A)$
		|	$\mathbf{match}\ v\ ((x, y) \Rightarrow e')$	$(A \times B)$
		|	$\mathbf{match}\ v\ (() \Rightarrow e')$	$(\mathbf{1})$
		|	$\mathbf{match}\ v\ \{\ell(x_\ell) \Rightarrow e_\ell\}_{\ell \in L}$	$(+\{\ell : A_\ell\})$

We might expect that **match** eliminations over values of positive types also exist in the language of values. This is indeed semantically and proof-theoretically justifiable—Levy [2006] call this a

language with *complex values*. Unfortunately it negates the slogan "*values are*" because now we have to compute them. It also complicates his stack machine that then has to construct simple values from complex values without the possibility of effects. This is certainly possible (see, for example, Lecture Notes on the K Machine), just a little unfortunate in this context.

4 A Stack Machine

To justify the name "call-by-push-value" we have to see where values are actually pushed. The abstract machine state has the form

$$e \rhd K$$

where e is a computation and K is a *stack*. K is also called the *continuation* since it represents all the computation that has to take place after e completes. We show some of the transition rules for the machine; others are easily copied from the K machine or imported from Levy's description.

$$\begin{array}{llllllll} (\mathsf{push}) & e\,v & \rhd & K & \longrightarrow & e & \rhd & \_\,v\,;\,K \\ (\mathsf{pop}_x) & \lambda x.\,e(x) & \rhd & \_\,v\,;\,K & \longrightarrow & e(v) & \rhd & K \end{array}$$

The first rule push pushes a value onto the stack K and therefore the name call-by-push-value. The second rule pops such a value v from the stack and substitutes it for x in e. If we denote the empty stack (also called the initial continuation) by ϵ then

$$\lambda x.\,e(x) \rhd \epsilon$$

is a final configuration. Lazy records work analogously, so let's concentrate on the shifts.

Regarding the upshift, **susp** e is a value, so the only relevant rule is

$$(\mathsf{force}) \quad (\mathbf{susp}\ e).\mathbf{force} \quad \rhd \quad K \quad \longrightarrow \quad e \quad \rhd \quad K$$

Regarding the downshift, we have

$$\begin{array}{lll} (\mathsf{bind}) & \mathbf{match}\ e\ (\langle x\rangle \Rightarrow e') \rhd K & \longrightarrow \quad e \rhd \mathbf{match}\ \_\ (\langle x\rangle \Rightarrow e'(x))\ ;\ K \\ (\mathsf{return}) & \langle v\rangle \rhd \mathbf{match}\ \_\ (\langle x\rangle \Rightarrow e'(x))\ ;\ K & \longrightarrow \quad e'(v) \rhd K \end{array}$$

In CBPV, $\langle v\rangle$ is written **return** v. We have named the rules bind and return because $\downarrow A$ plays the role of a strong monad by sequencing computations, passing on the return value of the first expression to the second. This yields another final machine state:

$$\langle v\rangle \rhd \epsilon$$

We imagine that even though this is a computation, the value v in this case would be observable representing the overall outcome of the computation.

Typing of the stack machine is actually somewhat interesting and also relevant to the next lecture. We need to express that the computation e in the state $e \rhd K$ is a computation of type $\underline{A}$ which is expected by the continuation K. We write $\Gamma \mid \underline{A} \vdash K : \underline{C}$ for this judgment.

$$\frac{\Gamma \vdash e : \underline{A} \quad \Gamma \mid \underline{A} \vdash K : \underline{C}}{\Gamma \vdash e \rhd K : \underline{C}}$$

This is the only time we consider a computation among the antecedents, separated from the others by a vertical bar. We only show two representative rules.

$$\frac{}{\Gamma \mid \underline{C} \vdash \epsilon : \underline{C}} \qquad \frac{\Gamma \vdash v : A \quad \Gamma \mid \underline{B} \vdash K : \underline{C}}{\Gamma \mid A \to \underline{B} \vdash (\_\, v\, ;\, K) : \underline{C}}$$

It's easy to verify that the transition rules for the abstract machine preserve typing. When the machine actually executes, Γ in these rules will be empty because we only evaluated closed computations. We still show the rules in this generality because this will be helpful in the next lecture. Levy also considers effects in the computations such as printing and updating a state. This is also something we'll consider in the next lecture.

5 Translating Call-By-Value

One of Levy's motivations is to design call-by-push-value as a subsuming language into which we can embed call-by-value and call-by-name. The key is the type translation A^v, where A is a call-by-value type, that is, a type in the ND language. We can think of it as a translation from ND to a fragment of ND with some special properties. Positive types are translated to corresponding positive types, while we insert shifts in the translation of negative types. This can be improved to avoid some double shifts.

$$\begin{array}{lcl}
(A \to B)^v & = & \uparrow(A^v \to \downarrow B^v) \\
(\&\{\ell : A_\ell\}_{\ell \in L})^v & = & \uparrow\&\{\ell : \downarrow A_\ell^v\}_{\ell \in L} \\
(A \times B)^v & = & A^v \times B^v \\
(\mathbf{1})^v & = & \mathbf{1} \\
(+\{\ell : A_\ell\}_{\ell \in L})^v & = & +\{\ell : A_\ell^v\}_{\ell \in L}
\end{array}$$

We translate call-by-value expressions e to computations e^c, where

$$x_1 : A_1, \ldots, x_n : A_n \vdash e : A \qquad \rightsquigarrow \qquad x_1 : A_1^v, \ldots, x_n : A_n^v \vdash e^c : \downarrow A^v$$

We show only the cases for functions.

$$\begin{array}{lcl}
(x)^c & = & \langle x \rangle \\
(\lambda x.\, e)^c & = & \langle \mathbf{susp}\ \lambda x.\, e^c \rangle \\
(e_1\, e_2)^c & = & \mathbf{match}\ e_1^c\ (\langle f \rangle \Rightarrow \mathbf{match}\ e_2^c\ (\langle x \rangle \Rightarrow f.\mathbf{force}\ x))
\end{array}$$

References

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

Jean-Marc Andreoli. Focussing and proof construction. *Annals of Pure and Applied Logic*, 107(1–3): 131–163, 2001.

Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Jean-Yves Girard. A new constructive logic: Classical logic. *Mathematical Structures in Computer Science*, 1(3):255–296, 1991.

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.

Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, November 2009.

Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In A.Felty, editor, *Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46, Lisbon, Portugal, July 2005. ACM Press.

Sean McLaughlin and Frank Pfenning. Efficient intuitionistic theorem proving with the polarized inverse method. In R.A.Schmidt, editor, *Proceedings of the 22nd International Conference on Automated Deduction (CADE-22))*, pages 230–244, Montreal, Canada, August 2009. Springer LNCS 5663.

Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

Robert J. Simmons. Structural focalization. *ACM Transactions on Computational Logic*, 15(3):21:1–21:33, 2014.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.