Lecture Notes on
Call-by-Value and Call-by-Name

15-816: Substructural Logics
Frank Pfenning

Lecture 22
November 16, 2017

In this lecture we consider two different polarization strategies for structural intuitionistic natural deduction. If we decide to translate propositions *positively* we obtain, under the computational interpretation of proofs as computations, a call-by-value language. If we translate propositions *negatively* we obtain call-by-name. These embeddings are compositional, supporting Levy's claim [Lev01] that call-by-push-value (CBPV) is a unifying approach to functional programming. With CPBV we can easily choose, at a fine-grained level, which computations are eager and which are lazy.

1 Call-by-Value as Positive Polarization

Let's assume we have a source language

Types	A, B, C	$::=$	$A \supset B \mid A \wedge B \mid \top \mid \vee\{l : A_l\}_{l \in L}$	
Expressions	E	$::=$	$x \mid \lambda x.\, E \mid E_1\, E_2$	$A \supset B$
		$\mid$	$\langle E_1, E_2 \rangle \mid \pi_1\, E \mid \pi_2 E$	$A \wedge B$
		$\mid$	$\langle\,\rangle$	$\top$
		$\mid$	$l(E) \mid \mathsf{case}\ E\ (l(x) \Rightarrow E_l)_{l \in L}$	$A \vee B$

We write $(A)^+$ for the positive polarization of A, which is defined inductively as follows:

$$
\begin{array}{lcl}
(A \supset B)^+ & = & \downarrow((A)^+ \to \uparrow(B)^+) \\
(A \wedge B)^+ & = & (A)^+ \times (B)^+ \\
(\top)^+ & = & \mathbf{1} \\
(\vee\{l : A_l\}_{l \in L})^+ & = & +\{l : (A_l)^+\}_{l \in L}
\end{array}
$$

This would seem to be a minimal positive polarization. Based on this, we can now write translations of expressions. The theorem we are aiming for is

$$\textit{If } \Gamma \vdash E : A \textit{ then } (\Gamma)^+ \vdash (E)^+ : \uparrow (A)^+$$

Note that the translation of an expression should be a computation, so we have to coerce $(A)^+$ to be a negative type in this judgment. This principle and the translation of types leaves very little leeway. Let's work through this carefully for functions. Assume

$$\Gamma \vdash \lambda x.\, E : A \supset B$$

Then

$$\Gamma^+ \vdash (\lambda x.\, E)^+ : \uparrow (A \supset B)^+$$

which works out to

$$\Gamma^+ \vdash (\lambda x.\, E)^+ : \uparrow\downarrow ((A)^+ \to \uparrow (B)^+)$$

We also know

$$\Gamma, x{:}(A)^+ \vdash (E)^+ : \uparrow (B)^+$$

From that, we can fill in

$$(\lambda x.\, E)^+ = \mathsf{return}\ \mathsf{thunk}\, (\lambda x.\, (E)^+)$$

What about application $(E_1\ E_2)^+$? We know

$$\begin{array}{lcl} (\Gamma)^+ \vdash (E_1)^+ & : & \uparrow\downarrow ((A)^+ \to \uparrow (B)^+) \\ (\Gamma)^+ \vdash (E_2)^+ & : & \uparrow (A)^+ \\ (\Gamma)^+ \vdash (E_1\ E_2)^+ & : & \uparrow (B)^+ \end{array}$$

From this we can see that the types almost force:

$$(E_1\ E_2)^+ = \mathsf{let\ val}\ f = (E_1)^+\ \mathsf{in\ let\ val}\ x = (E_2)^+\ \mathsf{in}\ (\mathsf{force}\ f)\, x$$

Also well-typed would be the result of swapping the two lets, or performing one more result binding at the end:

$$\begin{array}{lcl} (E_1\ E_2)^+ & = & \mathsf{let\ val}\ f = (E_1)^+\ \mathsf{in} \\ & & \mathsf{let\ val}\ x = (E_2)^+\ \mathsf{in} \\ & & \mathsf{let\ val}\ y = (\mathsf{force}\ f)\, x\ \mathsf{in} \\ & & \quad \mathsf{return}\ y \end{array}$$

In summary, for functions:

$$\begin{array}{lcl}(x)^+ & = & \mathsf{return}\, x \\ (\lambda\, x.\, E)^+ & = & \mathsf{return\ thunk}\, (\lambda x.(E)^+) \\ (E_1\, E_2)^+ & = & \mathsf{let\ val}\ f = (E_1)^+\ \mathsf{in\ let\ val}\ x = (E_2)^+\ \mathsf{in}\ (\mathsf{force}\ f)\, x\end{array}$$

Translations of pairs is simpler, since we can arrange to use positive (eager) pairs in the target.

$$\begin{array}{lcl}(\langle E_1, E_2\rangle)^+ & = & \mathsf{let\ val}\ x_1 = (E_1)^+\ \mathsf{in\ let\ val}\ x_2 = (E_2)^+\ \mathsf{in\ return}\, (x_1, x_2) \\ (\pi_1\, E)^+ & = & \mathsf{let\ val}\ x = (E)^+\ \mathsf{in\ match}\ x\ \mathsf{as}\ (y, z) \Rightarrow \mathsf{return}\ y \\ (\pi_2\, E)^+ & = & \mathsf{let\ val}\ x = (E)^+\ \mathsf{in\ match}\ x\ \mathsf{as}\ (y, z) \Rightarrow \mathsf{return}\ z\end{array}$$

Similarly for $\top$ and $\vee\{l : A_l\}_{l\in L}$

$$\begin{array}{lcl}(\langle\,\rangle)^+ & = & \mathsf{return}\ (\,) \\ (l(E))^+ & = & \mathsf{let\ val}\ x = (E)^+\ \mathsf{in\ return}\ l(x) \\ (\mathsf{case}\ E\ (l(x) \Rightarrow E_l)_{l\in L})^+ & = & \mathsf{let\ val}\ x = (E)^+\ \mathsf{in\ match}\ x\ \mathsf{as}\ (l(x) \Rightarrow (E_l)^+)_{l\in L}\end{array}$$

At this point we have completed our embedding. We could, for example, give a call-by-value operational semantics on unpolarized expressions and then show that this particular translation is operationally adequate. We are more inclined to think of this translation as the definition of call-by-value and move on.

2 Call-by-Name as Negative Polarization

We now consider the *negative polarization* of an unpolarized type. For the conjunction, we clearly should choose the negative conjunction, corresponding to lazy pairs consistent with call-by-name.

$$\begin{array}{lcl}(A \supset B)^- & = & \downarrow (A)^- \to (B)^- \\ (A \wedge B)^- & = & \&\{\pi_1 : (A)^-, \pi_2 : (B)^-\} \\ (\top)^- & = & \&\{\,\} \\ (\vee\{l : A_l\}_{l\in L}) & = & \uparrow +\{l : \downarrow (A_l)^-\}_{l\in L}\end{array}$$

Now the judgment $\Gamma \vdash E : A$ will be translated to $(\Gamma)^+ \vdash (E)^- : (A)^-$, where for the hypotheses with have

$$(x_1{:}A_1, \ldots, x_n{:}A_n)^+ = x_1{:}\downarrow(A_1)^-, \ldots, x_n{:}\downarrow (A_n)^-$$

The extra down shift for the context is forced since call-by-push-value allows only positively typed variables, while for the translation of an expression to a computation, no additional shift is necessary.

As we design the translation for expressions, let the types be or guide as usual.

$$\begin{array}{lcl} (\lambda x.\, E)^- & : & \downarrow (A)^- \to (B)^- \\ (x)^- & : & \downarrow (A)^- \\ (E)^- & : & (B)^- \end{array}$$

Clearly, we have

$$\begin{array}{lcl} (\lambda x.\, E)^- & = & \lambda x.\, (E)^- \\ (x)^- & = & \mathsf{force}\, x \end{array}$$

In the same style of reasoning:

$$\begin{array}{lcl} (E_1\, E_2)^- & : & (B)^- \\ (E_1)^- & : & \downarrow (A)^- \to (B)^- \\ (E_2)^- & : & (A)^- \end{array}$$

Again, it seems our hand is forced:

$$(E_1\, E_2)^- \quad = \quad (E_1)^-\, (\mathsf{thunk}\, (E_2)^-)$$

In summary, for functions:

$$\begin{array}{lcl} (x)^- & = & \mathsf{force}\, x \\ (\lambda x.\, E)^- & = & \lambda x.\, (E)^- \\ (E_1\, E_2)^- & = & (E_1)^-\, (\mathsf{thunk}\, (E_2)^-) \end{array}$$

This clearly represent call-by-name. We pass a computation, packaged as a thunk, and force that thunk where the variable is used. In all *call-by-need* language such as Haskell, the value of this forced expression is memoized so that future evaluations of $\mathsf{force}\, x$ do not evaluate the thunk again but retrieve its value.

For conjunction, we abbreviate the process. Recall that $(A \wedge B)^- = \&\{\pi_1 : (A)^-, \pi_2 : (B)^-\}$

$$\begin{array}{lcl} (\langle E_1, E_2 \rangle)^- & = & \{\pi_1 \Rightarrow (E_1)^-, \pi_2 \Rightarrow (E_2)^-\} \\ (\pi_1\, E) & = & (E)^-.\pi_1 \\ (\pi_2\, E) & = & (E)^-.\pi_2 \end{array}$$

Truth $\top$ is the nullary case of conjunction and consequently becomes $\&\{\,\}$. We the translate

$$(\langle\rangle)^- \quad = \quad \{\,\}$$

It seems implication and conjunction translates more directly for call-by-name than for call-by-value. However, disjunction has two shifts and is therefore more complicated.

$$\begin{array}{lcl} (l(E))^- & : & \uparrow + \{l : \downarrow (A_l)^-\}_{l \in L} \quad \text{for } l \in L \\ E^- & : & (A_l)^- \end{array}$$

so

$$(l(E))^- \quad = \quad \mathsf{return}\, l(\mathsf{thunk}\, (E)^-)$$

The elimination form is our most complex case

$$\begin{array}{lcl} (\mathsf{case}\ E\ (l(x) \Rightarrow E_l)_{l \in L})^- & : & (C)^- \\ (E)^- & : & \uparrow + \{l : \downarrow (A_l)^-\}_{l \in L} \\ (E_l)^- & : & (C)^- \\ (x)^- & : & \downarrow (A_l)^- \end{array}$$

but if want to respect all these types, the following suggests itself

$$(\mathsf{case}\ E\ (l(x) \Rightarrow E_l)_{l \in L})^- \quad = \quad \mathsf{let\ val}\ y = (E)^- \mathsf{match}\ y\ \mathsf{as}\ (l(x) \Rightarrow (E_l)^-)_{l \in L}$$

Summarizing whole call-by-name translation (which is to say, the negative translation)

$$\begin{array}{lcl} (x)^- & = & \mathsf{force}\, x \\ (\lambda x.\, E)^- & = & \lambda x.\, (E)^- \\ (E_1\, E_2)^- & = & (E_1)^-\, (\mathsf{thunk}\, (E_2)^-) \\ \\ (\langle E_1, E_2 \rangle)^- & = & \{\pi_1 \Rightarrow (E_1)^-, \pi_2 \Rightarrow (E_2)^-\} \\ (\pi_1\, E) & = & (E)^-.\pi_1 \\ (\pi_2\, E) & = & (E)^-.\pi_2 \\ \\ (\langle\,\rangle)^- & = & \{\,\} \\ \\ (l(E))^- & = & \mathsf{return}\, l(\mathsf{thunk}\, (E)^-) \\ (\mathsf{case}\ E\ (l(x) \Rightarrow E_l)_{l \in L})^- & = & \mathsf{let\ val}\ y = (E)^- \mathsf{match}\ y\ \mathsf{as}\ (l(x) \Rightarrow (E_l)^-)_{l \in L} \end{array}$$

3 Destinations

The operational semantics of call-by-push-value is very direct using ordered inference. In the next lecture we will introduce the Concurrent Logical Framework (CLF) which, unfortunately, is linear and does not support

ordered specifications. One idea, not very elegant, is to create explicit sequences of assumptions. But there is a different way, namely to use *destinations* to tie the propositions together. In general, the ordered context

$$A_1 \ \ldots \ A_n$$

is represented by

$$A_1(d_0, d_1) \ A_2(d_1, d_2) \ \ldots \ A_n(d_{n-1}, d_n)$$

where all of $d_0, d_1, d_2, \ldots, d_{n-1}, d_n$ are distinct parameters, and d_0 and d_n represent the left and right endpoints [SP11]. You can think of them just as if they were *channels* in our previous linear specifications, but used in a disciplined way since the context *is* actually ordered.

In our particular example, the configuration would look like

$$\begin{array}{l}\mathsf{eval}(M, d_{n+1}, d_n) \ \mathsf{cont}(K_n, d_n, d_{n-1}) \ \ldots \ \mathsf{cont}(K_1, d_1, d_0) \\ \mathsf{retn}(T, d_{n+1}, d_n) \ \mathsf{cont}(K_n, d_n, d_{n-1}) \ \ldots \ \mathsf{cont}(K_1, d_1, d_0)\end{array}$$

We can rearrange and optimize slightly, noting, for example, that we never need d_{n+1} and we use

$$\begin{array}{l}\mathsf{eval}(M, d) \\ \mathsf{retn}(T, d) \\ \mathsf{cont}(d, K, d')\end{array}$$

As a sample, we given the rules for functions, first in their ordered form and then in destination passing style. Note that the rules for applications must introduce a fresh destination.

Ordered	Destination-Passing
$\dfrac{\mathsf{eval}(M\ V)}{\mathsf{eval}(M) \quad \mathsf{cont}(\_\ V)} \to C_1$	$\dfrac{\mathsf{eval}(M\ V, d)}{\mathsf{eval}(M, d') \quad \mathsf{cont}(d', \_\ V, d)} \to C_1^{d'}$
$\dfrac{\mathsf{eval}(\lambda x.\ M)}{\mathsf{retn}(\lambda x.\ M)} \to C_2$	$\dfrac{\mathsf{eval}(\lambda x.\ M, d)}{\mathsf{retn}(\lambda x.\ M, d)} \to C_2$
$\dfrac{\mathsf{retn}(\lambda x.\ M) \quad \mathsf{cont}(\_\ V)}{\mathsf{eval}([V/x]M)} \to C_3$	$\dfrac{\mathsf{retn}(\lambda x.\ M, d') \quad \mathsf{cont}(d', \_\ V, d)}{\mathsf{eval}([V/x]M, d)} \to C_3$

As a preview of CLF [WCPW02, CPWW02, WCPW04, SNS08, SN11], we show the relevant part of the file cbpv.clf which implements the above idea.

There are a few things we have not discussed, such as the *indexing* of values and computations by their positive or negative types. We first highlight the three rules on the right.

```
eval/app : eval (app M V) D
            -o {Exists d'. eval M d' * cont d' (app1 V) D}.
eval/lam : eval (lam (\!x. M !x)) D -o {retn (lam (\!x. M !x)) D}.
eval/app1 : retn (lam (\!x. M !x)) D' * cont D' (app1 V) D
             -o {eval (M !V) D}.
```

Because we index values and computations, the code below is not only an operational specification but also a type checker for call-by-push-value. We have limited ourselves to the *binary* forms of $\&$ and $+$ since label sets are not so easily represented.

```
% Call-by-push-value in CLF
% Pure function fragment with shifts

neg : type.
pos : type.

arrow : pos -> neg -> neg.   % A -> B
up : pos -> neg.             % up A
down : neg -> pos.           % down A

% values and computations, indexed by their type
val : pos -> type.
comp : neg -> type.

% negative types
% A -> B
lam : (val A -> comp B) -> comp (arrow A B).
app : comp (arrow A B) -> val A -> comp B.

% up A
return : val A -> comp (up A).
letval : comp (up A) -> (val A -> comp C) -> comp C.

% down A
thunk : comp A -> val (down A).
force : val (down A) -> comp A.
```

```
% runtime artefacts
dest : neg -> type.
frame : neg -> neg -> type.
app1 : val A -> frame (arrow A B) B.
letval1: (val A -> comp C) -> frame (up A) C.

% ssos predicates
eval : comp A -> dest A -> type.
retn : comp A -> dest A -> type.
cont : dest A -> frame A B -> dest B -> type.

% A -> B
eval/lam : eval (lam (\!x. M !x)) D -o {retn (lam (\!x. M !x)) D}.
eval/app : eval (app M V) D
            -o {Exists d'. eval M d' * cont d' (app1 V) D}.
eval/app1 : retn (lam (\!x. M !x)) D' * cont D' (app1 V) D
             -o {eval (M !V) D}.

% up A
eval/return : eval (return V) D -o {retn (return V) D}.
eval/letval : eval (letval M (\!x. N !x)) D
               -o {Exists d'. eval M d'
                   * cont d' (letval1 (\!x. N !x)) D}.
eval/letval1 : retn (return V) D' * cont D' (letval1 (\!x. N !x)) D
                -o {eval (N !V) D}.

% down A
eval/force : eval (force (thunk M)) D
             -o {eval M D}.

#query * 1 * 1
Pi d0. eval (lam (\!x. return x)) d0 -o {retn M d0}.

#query * 1 * 1
Pi d0. eval (app (lam (\!x. return x)) (thunk (lam (\!y. return y)))) d0
 -o {retn M d0}.
```

Exercises

Exercise 1 Both call-by-value and call-by-name lead to code that is considerably more complex than it needs to be, including, for example, patterns such as $\mathsf{let\ val}\ x' = \mathsf{return}\ x\ \mathsf{in}\ M$. These spurious introduction/elimination forms are called *administrative redices*. Begin by showing an example of an expression whose call-by-value translation contains an administrative redex.

If possible, rewrite the call-by-value translation using two different forms, one resulting directly in a value the other in a computation.

$$\Gamma^+ \vdash (E)^p : (A)^+$$
$$\Gamma^+ \vdash (E)^n : \uparrow(A)^+$$

calling upon the appropriate translation form. Try to write the refined translation so that no administrative redices arise. If this does not work, do you see another approach to avoiding administrative redices?

Exercise 2 Carry out Exercise 1 for call-by-name.

Exercise 3 Investigate a *linear call-by-push-value* combined with Levy's by an adjunction with two shifts. Explore the expressive power of the result. Does linearity describe and interesting and useful properties of functional computation?

Exercise 4 Using a substitution-free operational semantics as in Exercise L21.1, specify a *call-by-need* operational semantics. Can you do this on call-by-push-value in general, or should it be integrated somehow (or described directly) on call-by-name?

References

[CPWW02] 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.

[Lev01] Paul Blain Levy. *Call-by-Push-Value*. PhD thesis, University of London, 2001.

[SN11] Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

[SNS08] Anders Schack-Nielsen and Carsten Schürmann. Celf - a logical framework for deductive and concurrent systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR'08)*, pages 320–326, Sydney, Australia, August 2008. Springer LNCS 5195.

[SP11] Robert J. Simmons and Frank Pfenning. Logical approximation for program analysis. *Higher-Order and Symbolic Computation*, 24(1–2):41–80, 2011.

[WCPW02] 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.

[WCPW04] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. Specifying properties of concurrent computations in CLF. In C.Schürmann, editor, *Proceedings of the 4th International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, Cork, Ireland, July 2004. Electronic Notes in Theoretical Computer Science (ENTCS), vol 199, pp. 133–145, 2008.