A Bidirectional Krivine Evaluator

Mikaël Mayer and Ravi Chugh
University of Chicago, Chicago, IL, USA
{mikaelm,rchugh}@uchicago.edu

Abstract

Bidirectional evaluation [Mayer et al., 2018] allows the output value of a λ-term to be modified and then back-propagated through the term, repairing leaf terms as necessary. To accompany their call-by-value formulation, we present two call-by-name systems. First, we recount the call-by-value approach proposed in [Mayer et al., 2018]; the key idea concerns back-propagating values through function application. Next, we formulate an analogous call-by-name system and establish corresponding correctness properties. Lastly, we define a backward Krivine-style evaluator that, compared to the functionally equivalent "direct" by-name backward evaluator, is notable for its relative simplicity.

Introduction

Bidirectional evaluation is a technique that allows arbitrary expressions in a standard λ-calculus to be "run in reverse" [Mayer et al., 2018]. Using this approach, (1) an expression e is evaluated to a value v, (2) the user makes "small" changes to the value yielding v' (structurally equivalent to v), and (3) the new value v' is "pushed back" through the expression, generating repairs as necessary to ensure that the new expression e' (structurally equivalent to e) evaluates to v'.

Shown below is the syntax of a pure λ-calculus extended with constants c. Our presentation employs natural (big-step, environment-style) semantics [Kahn, 1987], where function values are closures. Call-by-value function closures $\langle E; \lambda x.e \rangle$ refer to call-by-value environments E—which bind call-by-value values—and call-by-name function closures $\langle D; \lambda x.e \rangle$ refer to call-by-name environments D—which bind expression closures $\langle D; e \rangle$ yet to be evaluated. A stack S is a list of call-by-name expression closures.

Expressions	e	$::=$	$c \mid \lambda x.e \mid x \mid e_1\, e_2$
Call-By-Value Values	v	$::=$	$c \mid \langle E; \lambda x.e \rangle$
Call-By-Value Environments	E	$::=$	$- \mid E, x \mapsto v$
Call-By-Name Values	u	$::=$	$c \mid \langle D; \lambda x.e \rangle$
Call-By-Name Environments	D	$::=$	$- \mid D, x \mapsto \langle D_x; e \rangle$
Krivine Argument Stacks	S	$::=$	$[] \mid \langle D; e \rangle :: S$

Definition 1 (Structural Equivalence)**.** *Structural equivalence of expressions ($e_1 \sim e_2$), values ($v_1 \sim v_2$ and $u_1 \sim u_2$), environments ($E_1 \sim E_2$ and $D_1 \sim D_2$), and expression closures ($\langle E_1; e_1 \rangle \sim \langle E_2; e_2 \rangle$ and $\langle D_1; e_1 \rangle \sim \langle D_2; e_2 \rangle$) is equality modulo constants c_1 and c_2, which may differ, at the leaves of terms.*

In: J. Cheney, H-S. Ko (eds.): Proceedings of the Eighth International Workshop on Bidirectional Transformations (Bx 2019), Philadelphia, PA, USA, June 4, 2019, published at http://ceur-ws.org

1 Bidirectional Call-By-Value Evaluation

Figure 1 shows the bidirectional call-by-value evaluation rules; [Mayer et al., 2018] extends the core language with numbers, strings, tuples, lists, etc. In addition to a conventional "forward" evaluator, there is a "backward" evaluator (also referred to as "evaluation update" or simply "update"), whose behavior is customizable in [Mayer et al., 2018]. The environment-style semantics simplifies the presentation of backward evaluation; a substitution-based presentation would require tracking provenance.

BV Evaluation $\langle E; e\rangle \Rightarrow_{BV} v$

$$\text{BV-E-Const}\ \frac{}{\langle E; c\rangle \Rightarrow c} \qquad \text{BV-E-Fun}\ \frac{}{\langle E; \lambda x.e\rangle \Rightarrow \langle E; \lambda x.e\rangle}$$

$$\text{BV-E-Var}\ \frac{}{\langle E; x\rangle \Rightarrow E(x)} \qquad \text{BV-E-App}\ \frac{\begin{array}{c}\langle E; e_1\rangle \Rightarrow \langle E_f; \lambda x.e_f\rangle \\ \langle E; e_2\rangle \Rightarrow v_2 \\ \langle E_f,\, x \mapsto v_2; e_f\rangle \Rightarrow v\end{array}}{\langle E; e_1\ e_2\rangle \Rightarrow v}$$

BV Evaluation Update $\langle E; e\rangle \Leftarrow_{BV} v' \rightsquigarrow \langle E'; e'\rangle$

$$\text{BV-U-Const}\ \frac{}{\langle E; c\rangle \Leftarrow c' \rightsquigarrow \langle E; c'\rangle} \qquad \text{BV-U-Fun}\ \frac{}{\langle E; \lambda x.e\rangle \Leftarrow \langle E'; \lambda x.e'\rangle \rightsquigarrow \langle E'; \lambda x.e'\rangle}$$

$$\text{BV-U-Var}\ \frac{}{\langle E; x\rangle \Leftarrow v' \rightsquigarrow \langle E[x \mapsto v']; x\rangle}$$

$$\text{BV-U-App}\ \frac{\begin{array}{c}\langle E; e_1\rangle \Rightarrow \langle E_f; \lambda x.e_f\rangle \\ \langle E; e_2\rangle \Rightarrow v_2 \\ \langle E_f,\, x \mapsto v_2; e_f\rangle \Leftarrow v' \rightsquigarrow \langle E'_f,\, x \mapsto v'_2; e'_f\rangle \\ \langle E; e_1\rangle \Leftarrow \langle E'_f; \lambda x.e'_f\rangle \rightsquigarrow \langle E_1; e'_1\rangle \\ \langle E; e_2\rangle \Leftarrow v'_2 \rightsquigarrow \langle E_2; e'_2\rangle\end{array}}{\langle E; e_1\ e_2\rangle \Leftarrow v' \rightsquigarrow \langle E_1\ {}^{e_1}\!\oplus^{e_2} E_2; e'_1\ e'_2\rangle}$$

Figure 1: Bidirectional Call-By-Value Evaluation [Mayer et al., 2018].

Given an expression closure $\langle E; e\rangle$ (a "program") that evaluates to v, together with an updated value v', evaluation update traverses the evaluation derivation and rewrites the program to $\langle E'; e'\rangle$ such that it evaluates to v'. The first three update rules are simple. Given a new constant c', the BV-U-Const rule retains the original environment and replaces the original constant. Given a new function closure $\langle E'; \lambda x.e'\rangle$, the BV-U-Fun rule replaces both the environment and expression. Given a new value v', the BV-U-Var rule replaces the environment binding corresponding to the variable x used; $E[x \mapsto v']$ denotes structure-preserving replacement.

The rule BV-U-App for function application is what enables values to be pushed back through *all* expression forms. The first two premises evaluate the function and argument expressions using forward evaluation, and the third premise pushes the new value v' back through the function body under the appropriate environment. Two key aspects of the remainder of the rule warrant consideration. The first key is that update generates three new terms to grapple with: E'_f, v'_2, and e'_f. The first and third are "pasted together" to form the new closure $\langle E'_f; \lambda x.e'_f\rangle$ that some new function expression e'_1 must evaluate to, and the second is the value that some new argument expression e'_2 must evaluate to; these obligations are handled recursively by update (the fourth and fifth premises). The second key is that two new environments E_1 and E_2 are generated; these are reconciled by the following merge operator, which requires that all uses of a variable be updated consistently in the output.[1]

Definition 2 (BV Environment Merge $E_1\ {}^{e_1}\!\oplus^{e_2} E_2$)**.**

$$-\,{}^{e_1}\!\oplus^{e_2}\,- = -$$

$$(E_1,\, x \mapsto v_1)\,{}^{e_1}\!\oplus^{e_2}(E_2,\, x \mapsto v_2) = (E',\, x \mapsto v) \text{ where } E' = E_1\ {}^{e_1}\!\oplus^{e_2} E_2 \text{ and } v = \begin{cases} v_1 & \text{if } v_1 = v_2 \\ v_1 & \text{if } x \notin \mathit{freeVars}(e_2) \\ v_2 & \text{if } x \notin \mathit{freeVars}(e_1) \end{cases}$$

Theorem 1 (Structure Preservation of BV Update)**.**
If $\langle E; e\rangle \Rightarrow_{BV} v$ *and* $v \sim v'$ *and* $\langle E; e\rangle \Leftarrow_{BV} v' \rightsquigarrow \langle E'; e'\rangle$*, then* $\langle E; e\rangle \sim \langle E'; e'\rangle$*.*

Theorem 2 (Soundness of BV Update)**.**
If $\langle E; e\rangle \Leftarrow_{BV} v' \rightsquigarrow \langle E'; e'\rangle$*, then* $\langle E'; e'\rangle \Rightarrow_{BV} v'$*.*

[1] In practice, it is often useful to allow the user to specify a single example of a change, to be propagated to other variable uses automatically. [Mayer et al., 2018] proposes an alternative merge operator which trades soundness for practicality.

2 Bidirectional Call-By-Name Evaluation

Next, we define a bidirectional call-by-name system, which largely follows the call-by-value version.

BN Evaluation $\langle D; e\rangle \Rightarrow_{BN} u$

$$\textsc{BN-E-Const}\ \frac{}{\langle D; c\rangle \Rightarrow c}$$

$$\textsc{BN-E-Fun}\ \frac{}{\langle D; \lambda x.e\rangle \Rightarrow \langle D; \lambda x.e\rangle}$$

$$\textsc{BN-E-Var}\ \frac{D(x) = \langle D_x; e\rangle \qquad \langle D_x; e\rangle \Rightarrow u}{\langle D; x\rangle \Rightarrow u}$$

$$\textsc{BN-E-App}\ \frac{\langle D; e_1\rangle \Rightarrow \langle D_f; \lambda x.e_f\rangle \qquad \langle D_f,\, x \mapsto \langle D; e_2\rangle; e_f\rangle \Rightarrow u}{\langle D; e_1\ e_2\rangle \Rightarrow u}$$

BN Evaluation Update $\langle D; e\rangle \Leftarrow_{BN} u' \rightsquigarrow \langle D'; e'\rangle$

$$\textsc{BN-U-Const}\ \frac{}{\langle D; c\rangle \Leftarrow c' \rightsquigarrow \langle D; c'\rangle}$$

$$\textsc{BN-U-Fun}\ \frac{}{\langle D; \lambda x.e\rangle \Leftarrow \langle D'; \lambda x.e'\rangle \rightsquigarrow \langle D'; \lambda x.e'\rangle}$$

$$\textsc{BN-U-Var}\ \frac{D(x) = \langle D_x; e\rangle \qquad \langle D_x; e\rangle \Leftarrow u' \rightsquigarrow \langle D'_x; e'\rangle}{\langle D; x\rangle \Leftarrow u' \rightsquigarrow \langle D[x \mapsto \langle D'_x; e'\rangle]; x\rangle}$$

$$\textsc{BN-U-App}\ \frac{\begin{array}{c}\langle D; e_1\rangle \Rightarrow \langle D_f; \lambda x.e_f\rangle \\ \langle D_f,\, x \mapsto \langle D; e_2\rangle; e_f\rangle \Leftarrow u' \rightsquigarrow \langle D'_f,\, x \mapsto \langle D_2; e'_2\rangle; e'_f\rangle \\ \langle D; e_1\rangle \Leftarrow \langle D'_f; \lambda x.e'_f\rangle \rightsquigarrow \langle D_1; e'_1\rangle\end{array}}{\langle D; e_1\ e_2\rangle \Leftarrow u' \rightsquigarrow \langle D_1\, {}^{e_1}\!\oplus^{e_2} D_2; e'_1\ e'_2\rangle}$$

Figure 2: Bidirectional Call-By-Name Evaluation.

The BN-U-Const and BV-U-Fun rules are analogous to the call-by-value versions. The BN-U-Var rule for variables must now evaluate the expression closure to a value, and recursively update that evaluation derivation. Being call-by-name, rather than call-by-need, we do so every time the variable is used, without any memoization. The BN-U-App for application is a bit simpler than BV-U-App, because the argument expression is not forced to evaluate; thus, there is no updated argument expression to push back. Environment merge for call-by-name environments (defined in a companion technical report [Mayer and Chugh, 2019]) is similar to before.

Theorem 3 (Structure Preservation of BN Update)**.**
If $\langle D; e\rangle \Rightarrow_{BN} u$ *and* $u \sim u'$ *and* $\langle D; e\rangle \Leftarrow_{BN} u' \rightsquigarrow \langle D'; e'\rangle$*, then* $\langle D; e\rangle \sim \langle D'; e'\rangle$*.*

Theorem 4 (Soundness of BN Update)**.**
If $\langle D; e\rangle \Leftarrow_{BN} u' \rightsquigarrow \langle D'; e'\rangle$*, then* $\langle D'; e'\rangle \Rightarrow_{BN} u'$*.*

In addition to being sound with respect to forward call-by-name evaluation, it is sound with respect to forward call-by-value evaluation. To formalize this proposition below, we refer to the lifting $\lceil E\rceil$ and $\lceil v\rceil$ of by-value environments and values, respectively, to by-name versions, and to the evaluation $[\![\langle D; e\rangle]\!]$ of a delayed expression closure to a by-value value—these definitions can be found in [Mayer and Chugh, 2019].

Theorem 5 (Completeness of BN Evaluation)**.**
If $\langle E; e\rangle \Rightarrow_{BV} v$*, then* $\langle \lceil E\rceil; e\rangle \Rightarrow_{BN} \lceil v\rceil$*.*

Theorem 6 (Soundness of BN Update for BV Evaluation)**.**
If $\langle E; e\rangle \Rightarrow_{BV} v$ *and* $\langle \lceil E\rceil; e\rangle \Leftarrow_{BN} \lceil v'\rceil \rightsquigarrow \langle D'; e'\rangle$*, then* $[\![\langle D'; e'\rangle]\!] = v'$*.*

3 Bidirectional Krivine Evaluation

Lastly, we define a bidirectional "Krivine evaluator" (Figure 3) in the style of the classic (forward) Krivine machine [Krivine, 1985], an abstract machine that implements call-by-name semantics for the lambda-calculus. While lower-level than the "direct" call-by-name formulation, the forward and backward Krivine evaluators are even more closely aligned than the prior versions.

Following the approach of the Krivine machine, the forward evaluator maintains a stack S of arguments (i.e. expression closures). When evaluating an application $e_1\ e_2$, rather than evaluating the e_1 to a function closure, the argument expression e_2—along with the current environment D—is pushed onto the stack S of function arguments (the K-E-App rule); only when a function expression "meets" a (non-empty) stack of arguments is the function body evaluated (the K-E-Fun-App rule). The K-E-Const, K-E-Fun, and K-E-Var rules are similar to the call-by-name system, now taking stacks into account.

The backward evaluator closely mirrors the forward direction. Recall the two keys for updating applications (BV-U-App and BN-U-App): pasting together new function closures to be pushed back to the function

Forward K-Evaluation $(\langle D; e\rangle; S) \Rrightarrow u$

K-E-Const
$$\frac{}{(\langle D; c\rangle; []) \Rrightarrow c}$$

K-E-Fun
$$\frac{}{(\langle D; \lambda x.e\rangle; []) \Rrightarrow \langle D; \lambda x.e\rangle}$$

K-E-Var
$$\frac{D(x) = \langle D_x; e\rangle \quad (\langle D_x; e\rangle; S) \Rrightarrow u}{(\langle D; x\rangle; S) \Rrightarrow u}$$

K-E-Fun-App
$$\frac{(\langle D_f,\, x \mapsto \langle D_2; e_2\rangle; e_f\rangle; S) \Rrightarrow u}{(\langle D_f; \lambda x.e_f\rangle; \langle D_2; e_2\rangle :: S) \Rrightarrow u}$$

K-E-App
$$\frac{(\langle D; e_1\rangle; \langle D; e_2\rangle :: S) \Rrightarrow u}{(\langle D; e_1\ e_2\rangle; S) \Rrightarrow u}$$

Backward K-Evaluation $(\langle D; e\rangle; S) \Lleftarrow u' \rightsquigarrow (\langle D'; e'\rangle; S')$

K-U-Const
$$\frac{}{(\langle D; c\rangle; []) \Lleftarrow c' \rightsquigarrow (\langle D; c'\rangle; [])}$$

K-U-Fun
$$\frac{}{(\langle D; \lambda x.e\rangle; []) \Lleftarrow \langle D'; \lambda x.e'\rangle \rightsquigarrow (\langle D'; \lambda x.e'\rangle; [])}$$

K-U-Var
$$\frac{D(x) = \langle D_x; e\rangle \quad (\langle D_x; e\rangle; S) \Lleftarrow u' \rightsquigarrow (\langle D'_x; e'\rangle; S')}{(\langle D; x\rangle; S) \Lleftarrow u' \rightsquigarrow (\langle D[x \mapsto \langle D'_x; e'\rangle]; x\rangle; S')}$$

K-U-Fun-App
$$\frac{(\langle D_f,\, x \mapsto \langle D_2; e_2\rangle; e_f\rangle; S) \Lleftarrow u' \rightsquigarrow (\langle D'_f,\, x \mapsto \langle D'_2; e'_2\rangle; e'_f\rangle; S')}{(\langle D_f; \lambda x.e_f\rangle; \langle D_2; e_2\rangle :: S) \Lleftarrow u' \rightsquigarrow (\langle D'_f; \lambda x.e'_f\rangle; \langle D'_2; e'_2\rangle :: S')}$$

K-U-App
$$\frac{(\langle D; e_1\rangle; \langle D; e_2\rangle :: S) \Lleftarrow u' \rightsquigarrow (\langle D_1; e'_1\rangle; \langle D_2; e'_2\rangle :: S')}{(\langle D; e_1\ e_2\rangle; S) \Lleftarrow u' \rightsquigarrow (\langle D_1\, {}^{e_1}\!\oplus^{e_2} D_2; e'_1\ e'_2\rangle; S')}$$

Figure 3: Bidirectional Krivine Evaluation.

expression, and merging updated environments. Because the forward evaluation rule K-E-App does not syntactically manipulate a function closure, the update rule K-U-App does not construct a new closure to be pushed back. Indeed, only the environment merging aspect from the previous treatments is needed in K-U-App. The K-U-Fun-App rule for the new Krivine evaluation form—following the structure of the K-E-Fun-App rule—creates a new function closure and argument which will be reconciled by environment merge.[2]

Theorem 7 (Structure Preservation of Krivine Update)**.**
If $(\langle D; e\rangle; S) \Rrightarrow u$ *and* $u \sim u'$ *and* $(\langle D; e\rangle; S) \Lleftarrow u' \rightsquigarrow (\langle D'; e'\rangle; S')$*, then* $\langle D; e\rangle \sim \langle D'; e'\rangle$*.*

Theorem 8 (Soundness of Krivine Update)**.**
If $(\langle D; e\rangle; S) \Rrightarrow u$ *and* $(\langle D; e\rangle; S) \Lleftarrow u' \rightsquigarrow (\langle D'; e'\rangle; S')$*, then* $(\langle D'; e'\rangle; S') \Rrightarrow u'$*.*

The following connects the Krivine system to our (natural-semantics style) call-by-name system (and, hence, our call-by-value system)—analogous to the connection between the Krivine machine and traditional substitution-based call-by-name systems (e.g. [Biernacka and Danvy, 2007]).

Theorem 9 (Equivalence of Krivine Evaluation and BN Evaluation)**.**
$(\langle -; e\rangle; []) \Rrightarrow u$ *iff* $\langle -; e\rangle \Rightarrow_{BN} u$*.*

Corollary 1 (Soundess of Krivine Update for BN Evaluation)**.**
If $\langle -; e\rangle \Rightarrow_{BN} u$ *and* $(\langle -; e\rangle; []) \Lleftarrow u' \rightsquigarrow (\langle -; e'\rangle; [])$*, then* $\langle -; e'\rangle \Rightarrow_{BN} u'$*.*

Corollary 2 (Soundess of Krivine Update for BV Evaluation)**.**
If $\langle -; e\rangle \Rightarrow_{BV} v$ *and* $(\langle -; e\rangle; []) \Lleftarrow \lceil v'\rceil \rightsquigarrow (\langle -; e'\rangle; [])$*, then* $\langle -; e'\rangle \Rightarrow_{BV} v'$*.*

We conclude with two observations about the backward Krivine evaluator. First, it never creates new values (function closures, in particular) to be pushed back (like BV-U-App and BN-U-App do). Therefore, if the user interface is configured to disallow function values from being updated (that is, if the original program produces a first-order value c), then the K-U-Fun rule for bare function expressions can be omitted from the system.

Second, unlike the call-by-value and call-by-name versions, the backward Krivine evaluator does not refer to forward evaluation at all! The backward rules are straightforward analogs to the forward rules, using environment merge to reconcile duplicated environments. We believe this simplicity may help when scaling the design and implementation of bidirectional evaluation to larger, more full-featured languages.

[2]Our backward evaluator is not a proper "machine": the K-U-Var and K-U-App rules manipulate environments produced by recursive calls. Although not our goal here, we expect that existing approaches for turning our natural semantics formulation into an abstract state-transition machine (including the use of, e.g., markers or continuations) ought to work for turning our natural semantics into one of the "next 700 Krivine machines" [Douence and Fradet, 2007].

Acknowledgments

This work was supported by Swiss National Science Foundation Early Postdoc.Mobility Fellowship No. 175041 and U.S. National Science Foundation Grant No. 1651794.

References

[Biernacka and Danvy, 2007] Biernacka, M. and Danvy, O. (2007). A Concrete Framework for Environment Machines. *ACM Transactions on Computational Logic (TOCL)*.

[Douence and Fradet, 2007] Douence, R. and Fradet, P. (2007). The Next 700 Krivine Machines. *Higher-Order and Symbolic Computation*.

[Kahn, 1987] Kahn, G. (1987). Natural Semantics. In *Symposium on Theoretical Aspects of Computer Sciences (STACS)*.

[Krivine, 1985] Krivine, J.-L. (1985). Un interprète du λ-calculus.

[Mayer and Chugh, 2019] Mayer, M. and Chugh, R. (2019). A Bidirectional Krivine Evaluator (Technical Report). Forthcoming via `https://arxiv.org/`.

[Mayer et al., 2018] Mayer, M., Kunčak, V., and Chugh, R. (2018). Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*. Extended version available as *CoRR abs/1809.04209v2*.