A Bidirectional Krivine Machine

(Short Paper)

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, while being functionally equivalent to the "direct" by-name backward evaluator, exhibits notable characteristics.

0 Introduction

Recently, we proposed a technique called *bidirectional evaluation* [Mayer et al., 2018] that allows arbitrary expressions in a λ-calculus to be "run in reverse," including those which produce function values. In this system, (1) an expression e is evaluated to a value v, (2) the user makes "small" changes to the value yielding v', and (3) the new value v' (structurally equivalent to 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'.

Show 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: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, 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 $\langle E; e\rangle \Rightarrow_{BV} v$ **BV Evaluation Update** $\langle E; e\rangle \Leftarrow_{BV} v' \rightsquigarrow \langle E'; e'\rangle$

BV-E-Const
$$\langle E; c\rangle \Rightarrow c$$

BV-U-Const
$$\langle E; c\rangle \Leftarrow c' \rightsquigarrow \langle E; c'\rangle$$

BV-E-Fun
$$\langle E; \lambda x.e\rangle \Rightarrow \langle E; \lambda x.e\rangle$$

BV-U-Fun
$$\langle E; \lambda x.e\rangle \Leftarrow \langle E'; \lambda x.e'\rangle \rightsquigarrow \langle E'; \lambda x.e'\rangle$$

BV-E-Var
$$\langle E; x\rangle \Rightarrow E(x)$$

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

BV-E-App
$$\frac{\langle E; e_1\rangle \Rightarrow \langle E_f; \lambda x.e_f\rangle \quad \langle E; e_2\rangle \Rightarrow v_2 \quad \langle E_f,\, x \mapsto v_2; e_f\rangle \Rightarrow v}{\langle E; e_1\ e_2\rangle \Rightarrow v}$$

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

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 corresponding environment binding; $E[x \mapsto v']$ 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 an analogous bidirectional call-by-name system. The definitions largely follow the call-by-value version, so we elide the details here.

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

Briefly, because expression evaluation is delayed, the BN-U-Var rule does more than BV-U-Var (the premise requires a recursive update), and the BN-U-App rule does less than BV-U-App (the function expression is evaluated but not the argument). See § A.2 for more details.

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, we first define appropriate relationships between the call-by-value and call-by-name systems.

Definition 3 (BV Value and BV Environment Lifting)**.**

$$\lceil\langle E_f; \lambda y.e\rangle\rceil = \langle\lceil E_f\rceil; \lambda y.e\rangle \qquad \lceil c\rceil = c$$

$$\lceil -\rceil = - \qquad \lceil E, x \mapsto c\rceil = \lceil E\rceil, x \mapsto \langle E; c\rangle \qquad \lceil E, x \mapsto \langle E_f; \lambda y.e\rangle\rceil = \lceil E\rceil, x \mapsto \lceil\langle E_f; \lambda y.e\rangle\rceil$$

Definition 4 (BN Value, BN Environment, and BN Closure Evaluation)**.**

$$[\![c]\!] = c \qquad [\![\langle D; \lambda x.e\rangle]\!] = \langle[\![D]\!]; \lambda x.e\rangle$$

$$[\![-]\!] = - \qquad [\![D, x \mapsto \langle D_x; e\rangle]\!] = [\![D]\!], x \mapsto [\![\langle D_x; e\rangle]\!]$$

$$\frac{\langle[\![D]\!]; e\rangle \Rightarrow_{BV} v}{[\![\langle D; e\rangle]\!] = v}$$

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 2) in the style of the classic (forward) Krivine machine [Krivine, 1985]. Following that approach for 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 a 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.

Compared to the call-by-value and call-by-name versions, the backward Krivine evaluator is notable for its simplicity. 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 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 are needed in K-U-App. The K-U-Fun-App rule for the new Krivine evaluation form poses no challenges: the update rule, following the structure of the K-E-Fun-App rule, creates a new function closure and argument which will be reconciled by environment merge in the K-U-App rule.[2]

[2]Our backward evaluator is not a proper "machine": the K-U-App rule manipulates the results of the recursive call (to merge environments D_1 and D_2). 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].

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

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

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

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

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

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

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

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 2: Bidirectional Krivine Evaluation.

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! Indeed, with the sole exception of the environment merge operation in K-U-App, the backward rules are "obvious" analogs to the forward rules. However pleasant this may be, backward Krivine evaluation (essentially a call-by-name system) re-evaluates an expression closure every time it is used (K-U-Var). Addressing this performance concern would require the incorporation of memoization as in call-by-need systems. Appendix B discusses how restricting the syntax of function calls can be used to partially mitigate the re-evaluation problem.

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*.

[Flanagan et al., 1993] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. (1993). The Essence of Compiling with Continuations. In *Conference on Programming Language Design and Implementation (PLDI)*.

[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 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*.

[Sabry and Felleisen, 1992] Sabry, A. and Felleisen, M. (1992). Reasoning About Programs in Continuation-Passing Style. In *Conference on LISP and Functional Programming (LFP)*.

A Additional Definitions and Proofs

A.1 Appendix to § 1: Bidirectional Call-By-Value Evaluation

Theorem (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* $\langle E; e\rangle \sim \langle E'; e'\rangle$*.*

Proof. By induction. □

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

Proof. See extended version of [Mayer et al., 2018]. □

A.2 Appendix to § 2: Bidirectional Call-By-Name Evaluation

The call-by-name system in Figure 3 is similar to the call-by-value system.

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

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

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

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

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

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

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

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

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

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

Figure 3: Bidirectional Call-By-Name Evaluation.

The replacement rules (BN-U-Const and BV-U-Fun) are analogous. 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 (and, thus, there is no updated argument expression to push back). Environment merge for call-by-name environments is analogous to merge for call-by-value environments.

Definition 5 (BN Environment Merge)**.**

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

$$(D_1,\, x \mapsto k_1)^{e_1}\!\oplus^{e_2}(D_2,\, x \mapsto k_2) = (D',\, x \mapsto k) \text{ where } D' = D_1\,{}^{e_1}\!\oplus^{e_2} D_2 \text{ and } k = \begin{cases} k_1 & \text{if } k_1 = k_2 \\ k_1 & \text{if } x \notin \mathit{freeVars}(e_2) \\ k_2 & \text{if } x \notin \mathit{freeVars}(e_1) \end{cases}$$

Theorem (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* $\langle D; e\rangle \sim \langle D'; e'\rangle$*.*

Proof. By induction. □

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

Proof. By induction, similar to the proof for Soundness of BV Update. □

Lemma 1 (Strong Completeness of BN Evaluation)**.**
If $\langle D; e\rangle \Rightarrow_{BN} u$ *and* $\langle [\![D]\!]; e\rangle \Rightarrow_{BV} v$*, then* $[\![u]\!] = v$*.*

Proof. By simultaneous induction on the evaluation derivations. Assume $\langle D; e\rangle \Rightarrow_{BN} u$ and $\langle [\![D]\!]; e\rangle \Rightarrow_{BV} v$.

Case $e = c$: By Definition 4.

Case $e = \lambda x.e'$: By Definition 4.

Case $e = x$:
By Assumption and BN-E-Var, $D(x) = \langle D_x; e_x\rangle$ and $\langle D_x; e_x\rangle \Rightarrow_{BN} u$.
By Assumption and BV-E-Var, $v = [\![D]\!](x)$. By Definition 4, $\langle [\![D_x]\!]; e_x\rangle \Rightarrow_{BV} v$.
By the Induction Hypothesis, $[\![u]\!] = v$.

Case $e = e_1\ e_2$:
By Assumption and BV-E-App, $\exists E_f, \lambda x.e_f, v_2$ s.t. $\langle [\![D]\!]; e_1\rangle \Rightarrow_{BV} \langle E_f; \lambda x.e_f\rangle$ and $\langle [\![D]\!]; e_2\rangle \Rightarrow_{BV} v_2$.
By Definition 4, $[\![\langle D; e_2\rangle]\!] = v_2$.
By Assumption and BN-E-App, $\exists D_f, \lambda y.g_f$ s.t. $\langle D; e_1\rangle \Rightarrow_{BN} \langle D_f; \lambda y.g_f\rangle$.
By the Induction Hypothesis, $[\![\langle D_f; \lambda y.g_f\rangle]\!] = \langle E_f; \lambda x.e_f\rangle$. Thus, $[\![D_f]\!] = E_f$ and $\lambda y.g_f = \lambda x.e_f$.
Furthermore, $\langle D_f,\ x \mapsto \langle D; e_2\rangle; e_f\rangle \Rightarrow_{BN} u$ on one side, and $\langle E_f,\ x \mapsto v_2; e_f\rangle \Rightarrow_{BV} v$ on the other.
By Definition 4, $([\![D_f,\ x \mapsto \langle D; e_2\rangle]\!]) = ([\![D_f]\!],\ x \mapsto [\![\langle D; e_2\rangle]\!]) = (E_f,\ x \mapsto v_2)$.
By the Induction Hypothesis, $[\![u]\!] = v$. □

Lemma 2 (Strong Soundness of BN Update for BV Evaluation)**.**
If $\langle [\![D]\!]; e\rangle \Rightarrow_{BV} v$ *and* $\langle D; e\rangle \Leftarrow_{BN} u' \rightsquigarrow \langle D'; e'\rangle$*, then* $\langle [\![D']\!]; e'\rangle \Rightarrow_{BV} [\![u']\!]$*.*

Proof. By simultaneous induction on the derivations. Assume $\langle [\![D]\!]; e\rangle \Rightarrow_{BV} v$ and $\langle D; e\rangle \Leftarrow_{BN} u' \rightsquigarrow \langle D'; e'\rangle$.

Case $e = c$: By Lemma 1.

Case $e = \lambda x.e'$: By Lemma 1.

Case $e = x$:
By Assumption and BV-E-Var, $\langle [\![D]\!]; x\rangle \Rightarrow_{BV} v$.
By Assumption, $\langle D; x\rangle \Leftarrow_{BN} u' \rightsquigarrow \langle D'; x\rangle$.
By BN-U-Var, $\exists D_x, e_x$ s.t. $D(x) = \langle D_x; e_x\rangle$, $\langle D; x\rangle \Leftarrow u' \rightsquigarrow \langle D[x \mapsto \langle D'_x; e'\rangle]; x\rangle$, and $D' = D[x \mapsto \langle D'_x; e'\rangle]$.
Moreover, $[\![D]\!](x) = v$, so by induction, $\langle [\![D']\!]; x\rangle \Rightarrow_{BV} [\![u']\!]$.

Case $e = e_1\ e_2$:
By Assumption, $\langle [\![D]\!]; e_1\ e_2\rangle \Rightarrow_{BV} v$ and $\langle D; e_1\ e_2\rangle \Leftarrow u' \rightsquigarrow \langle D_1 {}^{e_1}\!\oplus^{e_2} D_2; e'_1\ e'_2\rangle$ for some D_1, D_2, e_1 and e_2.
By rule BV-E-App, exists E_f and $\lambda x.e_f$ and v_2 such that $\langle [\![D]\!]; e_1\rangle \Rightarrow_{BV} \langle E_f; \lambda x.e_f\rangle$ and $\langle [\![D]\!]; e_2\rangle \Rightarrow_{BV} v_2$.
By the first premise BN-U-App, exists D' and $\lambda y.g_f$ such that $\langle D; e_1\rangle \Rightarrow_{BN} \langle D_f; \lambda y.g_f\rangle$.
Using Lemma 1, $[\![D_f]\!] = E_f$ and $\lambda y.g_f = \lambda x.e_f$.
By the second premise BN-U-App, $\langle D_f,\ x \mapsto \langle D; e_2\rangle; e_f\rangle \Leftarrow_{BN} u' \rightsquigarrow \langle D'_f,\ x \mapsto \langle D_2; e'_2\rangle; e'_f\rangle$.
By definition, $[\![D_f,\ x \mapsto \langle D; e_2\rangle]\!] = [\![D_f]\!],\ x \mapsto [\![\langle D; e_2\rangle]\!] = E_f,\ x \mapsto v_2$ and $\langle E_f,\ x \mapsto v_2; e_f\rangle \Rightarrow_{BV} v$.
So we can apply Lemma 2 recursively and obtain that $\langle [\![D'_f,\ x \mapsto \langle D_2; e'_2\rangle]\!]; e'_f\rangle \Rightarrow_{BV} [\![u']\!]$.
Therefore $\langle [\![D'_f]\!],\ x \mapsto [\![\langle D_2; e'_2\rangle]\!]; e'_f\rangle \Rightarrow_{BV} [\![u']\!]$.
By the third premise of BN-U-App and induction, we obtain that $\langle [\![D_1]\!]; e'_1\rangle \Rightarrow_{BV} [\![\langle D'_f; \lambda v.e'_f\rangle]\!]$.
Therefore, $\langle [\![D_1]\!]; e'_1\rangle \Rightarrow_{BV} \langle [\![D'_f]\!]; \lambda v.e'_f\rangle$.
By applying a BN version of Lemma A.3 from the appendix of [Mayer et al., 2018], and because of the definition of the two-way conservative merge, $\langle [\![D_1]\!]^{e_1}\!\oplus^{e_2} [\![D_2]\!]; e'_1\rangle \Rightarrow_{BV} \langle [\![D'_f]\!]; \lambda x.e'_f\rangle$
Similarly, $\langle [\![D_1]\!]^{e_1}\!\oplus^{e_2} [\![D_2]\!]; e'_2\rangle \Rightarrow_{BV} [\![\langle D_2; e'_2\rangle]\!]$.
By BV-E-App, we thus obtain that $\langle [\![D_1]\!]^{e_1}\!\oplus^{e_2} [\![D_2]\!]; e'_1\ e'_2\rangle \Rightarrow_{BV} [\![u']\!]$.
Because the BN conservative two-way merge (definition 5) has the same rules as the BV conservative two-way merge in Definition 3.1 from [Mayer et al., 2018], by induction we can prove that $\langle [\![D_1 {}^{e_1}\!\oplus^{e_2} D_2]\!]; e'_1\rangle = \langle [\![D_1]\!]^{e_1}\!\oplus^{e_2} [\![D_2]\!]; e'_1\rangle$, and conclude that $\langle [\![D_1 {}^{e_1}\!\oplus^{e_2} D_2]\!]; e'_1\ e'_2\rangle \Rightarrow_{BV} [\![u']\!]$.

□

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

Proof. By Strong Completeness of BN Evaluation. □

Theorem (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* $[\![\langle D'; e'\rangle]\!] = v'$*.*

Proof. By Strong Soundness of BN Update for BV Evaluation, since $[\![\lceil E\rceil]\!] = E$ and $[\![\lceil v\rceil]\!] = v$ by definition. □

A.3 Appendix to § 3: Bidirectional Krivine Evaluation

Theorem (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* $\langle D; e\rangle \sim \langle D'; e'\rangle$*.*

Proof. By induction. □

Theorem (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* $(\langle D'; e'\rangle; S') \Rrightarrow u'$*.*

Proof. By induction. □

Lemma 3 (Strong Equivalence of Krivine Evaluation and BN Evaluation)**.**
Assume $x_1, \ldots, x_n \notin dom(D)$*. Then:*
$(\langle D; e\rangle; \langle D_1; e_1\rangle :: \ldots :: \langle D_n; e_n\rangle :: []) \Rrightarrow u$ *iff* $\langle D,\, x_1 \mapsto \langle D_1; e_1\rangle \ldots,\, x_n \mapsto \langle D_n; e_n\rangle; e\, x_1 \ldots x_n\rangle \Rightarrow_{BN} u$*.*

Proof of $\Rrightarrow$ *direction.* By induction on the Krivine evaluation tree.

Case $e = c$: Then n must be zero and thus it holds.

Case $e = x$:
Then we can assume that $D(x) = \langle D_x; e_x\rangle$ for some D_x and e_x.
We can also assume that $(\langle D_x; e_x\rangle; \langle D_1; e_1\rangle :: \ldots :: \langle D_n; e_n\rangle :: []) \Rrightarrow u$.
By induction, $\langle D_x,\, x_1 \mapsto \langle D_1; e_1\rangle \ldots,\, x_n \mapsto \langle D_n; e_n\rangle; e_x\, x_1 \ldots x_n\rangle \Rightarrow_{BN} u$.
Thus, using BN-E-Var and BN-E-App by induction on n, we can verify that
$\langle D,\, x_1 \mapsto \langle D_1; e_1\rangle \ldots,\, x_n \mapsto \langle D_n; e_n\rangle; x\, x_1 \ldots x_n\rangle \Rightarrow_{BN} u$.

Case $e = \lambda x.e'$:
If n is zero, then it holds again.
If $n \geq 1$, then from
$(\langle D_f; \lambda x.e_f\rangle; \langle D_1; e_1\rangle :: \ldots :: \langle D_n; e_n\rangle :: []) \Rrightarrow u$
we can assume that
$(\langle D_f,\, x \mapsto \langle D_1; e_1\rangle; e_f\rangle; \langle D_2; e_2\rangle :: \ldots :: \langle D_n; e_n\rangle :: []) \Rrightarrow u$
By induction, we obtain that:
$\langle D_f,\, x \mapsto \langle D_1; e_1\rangle,\, x_2 \mapsto \langle D_2; e_2\rangle \ldots,\, x_n \mapsto \langle D_n; e_n\rangle; e_f\, x_2 \ldots x_n\rangle \Rightarrow_{BN} u$.
By induction on n, we can prove that
$\langle D,\, x_1 \mapsto \langle D_1; e_1\rangle \ldots,\, x_n \mapsto \langle D_n; e_n\rangle; ((\lambda x.e_f)\, x_1)\, x_2 \ldots x_n\rangle \Rightarrow_{BN} u$.

Case $e = e_1\, e_2$: Similar reasoning to above.

□

Proof of $\Lleftarrow$ *direction.* By induction on the BN evaluation tree. The reasoning is similar to above.

□

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

Proof. Follows from Strong Equivalence of Krivine Evaluation and BN Evaluation. □

Corollary (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* $\langle -; e'\rangle \Rightarrow_{BN} u'$*.*

Proof. By Soundness of Krivine Update and Equivalence of Krivine Evaluation and BN Evaluation. □

Corollary (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* $\langle -; e'\rangle \Rightarrow_{BV} v'$*.*

Proof. By Completeness of BN Evaluation. □

B Performance Considerations for Updating Function Applications

Whereas the backward call-by-value and call-by-name evaluators rely on their forward counterparts (i.e. in the BV-U-App and BN-U-App) rules, the backward Krivine evaluator does not (cf. K-U-App).

In an attempt to eliminate, or at least reduce, the reliance of the former two systems on forward evaluation, consider the restriction of function applications to the form $f\ x$, where both the function and argument are variables—a kind of A-normal form (ANF) [Sabry and Felleisen, 1992, Flanagan et al., 1993]. This restriction would require the language to include an explicit $\texttt{let}\ x\ e_1\ e_2$ construct. Below, we consider the update rules for these two expression forms.

B.1 BV-U-App in ANF

The following two rules are admissible in the "full" call-by-value system:

BV-U-App-ANF

$$\frac{E(f) = \langle E_f; \lambda y.e_f\rangle \qquad E(x) = v_x \\ \langle E_f,\, y \mapsto v_x; e_f\rangle \Leftarrow v' \rightsquigarrow \langle E'_f,\, y \mapsto v'_x; e'_f\rangle}{\langle E; f\ x\rangle \Leftarrow v' \rightsquigarrow \langle E[f \mapsto \langle E'_f; \lambda y.e'_f\rangle][x \mapsto v'_x]; f\ x\rangle}$$

BV-U-Let

$$\frac{\langle E; e_1\rangle \Rightarrow v_1 \\ \langle E,\, x \mapsto v_1; e_2\rangle \Leftarrow v'_2 \rightsquigarrow \langle E_2,\, x \mapsto v'_1; e'_2\rangle \\ \langle E; e_1\rangle \Leftarrow v'_1 \rightsquigarrow \langle E_1; e'_1\rangle}{\langle E; \texttt{let}\ x\ e_1\ e_2\rangle \Leftarrow v'_2 \rightsquigarrow \langle {E_1}^{e_1} \oplus_E^{e_2} E_2; \texttt{let}\ x\ e'_1\ e'_2\rangle}$$

Notice that BV-U-App-ANF would not refer to the forward evaluator, but the BV-U-Let rule does. Thus, simply restricting to ANF would not eliminate the dependency on forward evaluation.

Furthermore, the BV-U-Let rule highlights situations in which a single expression will be forward evaluated multiple times during backward evaluation: when `let`-expressions appear nested in equations (e.g. $\texttt{let}\ x_1\ (\texttt{let}\ x_2\ e_2\ (\texttt{let}\ x_3\ e_3 \cdots))\ e$). Memoization would need to be employed to reduce this re-computation.

B.2 BN-U-App in ANF

What about the same approach in the call-by-name system? This time, the ANF application rule is no simpler (in fact, it's more complicated), because variable expressions do not benefit from already having been evaluated. Evaluation would be needed by BN-U-App-ANF but not BN-U-Let. Again, memoization would need to be employed to reduce re-computation.

BN-U-App-ANF

$$\frac{D(f) = \langle D_1; e_1\rangle \qquad D(x) = \langle D_2; e_2\rangle \\ \langle D; e_1\rangle \Rightarrow \langle D_f; \lambda y.e_f\rangle \\ \langle D_f,\, y \mapsto \langle D_2; e_2\rangle; e_f\rangle \Leftarrow u' \rightsquigarrow \langle D'_f,\, y \mapsto \langle D'_2; e'_2\rangle; e'_f\rangle \\ \langle D_1; e_1\rangle \Leftarrow \langle D'_f; \lambda y.e'_f\rangle \rightsquigarrow \langle D'_1; e'_1\rangle}{\langle D; f\ x\rangle \Leftarrow u' \rightsquigarrow \langle D[f \mapsto \langle D'_1; e'_1\rangle][x \mapsto \langle D'_2; e'_2\rangle]; f\ x\rangle}$$

BN-U-Let

$$\frac{\langle D,\, x \mapsto \langle D; e_1\rangle; e_2\rangle \Leftarrow u'_2 \rightsquigarrow \langle D_2,\, x \mapsto \langle D_1; e'_1\rangle; e'_2\rangle}{\langle D; \texttt{let}\ x\ e_1\ e_2\rangle \Leftarrow u'_2 \rightsquigarrow \langle {D_1}^{e_1} \oplus^{e_2} D_2; \texttt{let}\ x\ e'_1\ e'_2\rangle}$$