Reinvestigation of Symmetric Lambda Calculus

Yayoi Ueda   Kenichi Asai
Ochanomizu University

Abstract

This paper presents a symmetric lambda calculus (SLC) in which both the duality between call-by-value and call-by-name and the duality between expressions and continuations hold at the same time. The idea of SLC was originally introduced by Filinski in 1989, but has not been investigated seriously since then. This paper first reformulates SLC using small-step reduction semantics in a completely symmetric way. We then show call-by-value and call-by-name evaluation strategies for SLC and prove that both enjoy various formal properties, such as subject reduction. Finally, we prove that the small-step semantics shown here coincides with Filinski's original definition of SLC using the functional correspondence approach: the two semantics are related via defunctionalization.

Keywords: symmetric lambda calculus, control operators, functional correspondence, defunctionalization

1 Introduction

Background and related work In 1990, Griffin [7] showed the correspondence between the law of double-negation elimination in classical logic and the control operator $\mathcal{C}$ introduced by Felleisen and Hieb [5]. Since then, many researchers explored the foundational theory for continuations and control operators. Parigot [8] introduced $\lambda\mu$-calculus and showed that it corresponds to the classical natural deduction. Curien and Herbelin [3] introduced $\overline{\lambda}\mu\tilde{\mu}$-calculus that exhibits symmetry between expressions and continuations as well as between call-by-value (CBV) and call-by-name (CBN) based on the classical sequent calculus. Selinger [12] gave a categorical semantics to $\lambda\mu$-calculus and showed categorical duality between CBV and CBN. These work led to Wadler's Dual Calculus [14] that shows CBV/CBN symmetry in a particularly clear way in a simple syntax and operational semantics. Tzevelekos [13] studied the Dual Calculus in detail and showed various syntactic properties.

Underlying these work is an intuition that expressions and continuations are dual. Surprisingly, this observation was made as early as in 1989 by Filinski [6]. He presented a symmetric lambda calculus (SLC) in which expressions and continuations are treated in a complete symmetry. Just like we abstract over expressions to receive an argument expression, we can abstract over continuations in SLC to capture the current continuation. This system has many implications. The CBV/CBN duality naturally arises from the priority of execution: to execute expressions first leads to CBV and continuations first CBN. The duality between types of expressions and continuations even suggests a relationship to classical logic via de Morgan's laws.

However, although most of the previous work mentioned Filinski's work, none of them actually investigated the relationship to SLC seriously. This is partly because Filinski's SLC lacks small-step reduction semantics. It makes it harder to compare Filinski's SLC to other calculi such as Dual Calculus, because they are often defined as small-step reduction semantics. This is unfortunate because SLC not only contains the standard lambda-calculus naturally but also presents control operators in a particularly simple and intuitive way.

The name symmetric lambda calculus was independently used by Barbanera and Berardi [2]. However, their work is quite different from Filinski's SLC. Their calculus includes a notion of symmetric application where either component of an application can be a function or an argument. They proved the strong normalization property of this calculus.

This work This paper investigates small-step reduction semantics for Filinski's SLC. It handles both non-deterministic, CBV, and CBN evaluation strategies. The non-deterministic semantics exhibits com-

expression	e	$::=$	$n \mid x \mid () \mid (e,\, e) \mid e \uparrow f \mid \lceil f \rceil$
function	f	$::=$	$g \mid p \Rightarrow e \mid c \Leftarrow q \mid \overline{e} \mid \underline{c}$
continuation	c	$::=$	$\bullet \mid y \mid \{\} \mid \{c,\, c\} \mid f \downarrow c \mid \lfloor f \rfloor$
pattern for e	p	$::=$	$x \mid () \mid \lceil g \rceil \mid (p,\, p)$
pattern for c	q	$::=$	$y \mid \{\} \mid \lfloor g \rfloor \mid \{q,\, q\}$

Figure 1: Syntax of SLC

plete symmetry, whereas CBV and CBN strategies are dual to each other. A type system for SLC is shown which satisfies various properties such as subject reduction. The paper then shows that the small-step semantics presented here and the original denotational semantics of SLC given by Filinski are in functional correspondence with each other. This work is also a non-trivial application of Danvy's functional correspondence approach [1, 4].

Overview In the next Section, we introduce SLC, its syntax, types, typing rules, non-deterministic reduction rules, and some properties. In Section 3, we show the CBV strategy and its properties such as uniqueness of evaluation. In Section 4, we show the CBN strategy and its properties. We defunctionalize Filinski's original CBV denotational semantics, and show the correspondence between the two SLC's in Section 5. Finally, we also defunctionalize Filinski's CBN semantics and show their correspondence in Section 6.

2 Symmetric Lambda Calculus

This section introduces the symmetric lambda calculus (SLC). A standard way to represent reduction semantics is to use an evaluation context: $E[M] \rightsquigarrow E[M']$ if $M \rightsquigarrow M'$. In SLC, we represent it with a configuration $\langle\ e \mid c\ \rangle$, where e is an expression (corresponding to M) and c is a continuation (corresponding to $E[\,]$). In addition, SLC has three-place configuration $\langle\ e \mid f \mid c\ \rangle$ which represents that an expression e is passed to a function f in a context c. In SLC, an evaluation context and searching for a redex are both explicit in a configuration. SLC calculates an expression and a continuation in a completely dual manner.

2.1 Syntax

Figure 1 shows the syntax of SLC.[1] An expression e is either a natural number n, a variable x, a unit $()$, a pair of expressions $(e,\, e)$, an application $e \uparrow f$, or a function treated as an expression $\lceil f \rceil$. An application $e \uparrow f$ passes an expression e to a function f. Unlike the standard lambda-calculus, a function f is placed after the argument e. In SLC, an expression and a function are distinct syntactic objects. To treat a function as an expression (to pass a higher-order function to another function, for example), a function is tagged with $\lceil\ \rceil$.

A continuation is defined as a dual notion of an expression. A continuation c is either an initial continuation $\bullet$, a continuation variable y, a counit $\{\}$, a pair of continuations $\{c,\, c\}$, a continuation application $f \downarrow c$, which represents a continuation that applies f before passing a value to c, or a function treated as a continuation $\lfloor f \rfloor$. An initial continuation $\bullet$ represents a continuation that receives a natural number n. When the initial continuation receives a natural number n, the computation finishes with the result n. The counit $\{\}$ represents a continuation that never receives any value.

A function is either a function variable g, an expression abstraction $p \Rightarrow e$ which binds the current expression to p and replaces the expression with e, a continuation abstraction $c \Leftarrow q$ which binds the current continuation to q and replaces the continuation with c, $\overline{e}$ an expression that evaluates to a function $\lceil f \rceil$, or $\underline{c}$ a continuation that evaluates to a function $\lfloor f \rfloor$.

[1]This syntax slightly deviates from Filinski's original syntax. Filinski used $f \uparrow e$ and $q \Rightarrow c$ instead of our $e \uparrow f$ and $c \Leftarrow q$, respectively. We changed the syntax because the new syntax exhibits beautiful symmetry when written in a small-step semantics.

$$\begin{array}{rcll} T & ::= & +A & \text{(types of expressions } e\text{)} \\ & | & A \supset B & \text{(types of functions } f\text{)} \\ & | & \neg B & \text{(types of continuations } c\text{)} \\ A, B & ::= & \multicolumn{2}{l}{\bot \mid \top \mid \mathsf{int} \mid A \wedge B \mid A \vee B \mid A \rightarrow B \mid A - B} \end{array}$$

Figure 2: Types of SLC

An expression abstraction $p \Rightarrow e$ has a pattern p as its formal argument. If the pattern is a variable x, $x \Rightarrow e$ corresponds to $\lambda x.e$ in the standard lambda calculus. A pattern p can also be a unit, a pair, or a function pattern $\lceil g \rceil$. Dually, q represents a pattern for a continuation abstraction $c \Leftarrow q$.

We assume that the sets of variable names for expressions, continuations, and functions do not overlap. We use a meta variable var to represent any of the three kinds of variables.

As an example, Filinski represented call/cc in SLC as follows [6]:

$$\mathsf{call/cc} = (\lceil g \rceil \Rightarrow \lceil y \Leftarrow \_ \rceil \uparrow g) \downarrow y \Leftarrow y$$

where _ represents a dummy variable not used anywhere else. It grabs the current continuation in y and replaces the current continuation with $(\lceil g \rceil \Rightarrow \lceil y \Leftarrow \_ \rceil \uparrow g) \downarrow y$. The installed function $(\lceil g \rceil \Rightarrow \lceil y \Leftarrow \_ \rceil \uparrow g)$ receives a function g and passes the representation of the current continuation $\lceil y \Leftarrow \_ \rceil$ to g. The passed continuation, when applied, will replace the then continuation with y, achieving jump to the continuation captured in y. (See Section 2.4 for an example reduction sequence for call/cc.) Similarly, Felleisen's $\mathcal{C}$ operator can be defined as follows:

$$\mathcal{C} = (\lceil g \rceil \Rightarrow \lceil y \Leftarrow \_ \rceil \uparrow g) \downarrow \bullet \Leftarrow y$$

2.2 Types

Figure 2 shows types of SLC. Since SLC consists of three syntactic objects, the types of SLC consist of three types: an expression type $+A$, a continuation type $\neg B$, and a function type $A \supset B$.

The type $+A$ represents a type of an expression. The prefix $+$ indicates that it is a type of an expression. The type $\neg B$ represents a type of a continuation that receives a value of type $+B$. In this paper we use $\neg$ for a type of continuations. The type $A \supset B$ represents a type of functions that receive an expression of type $+A$ and return an expression of type $+B$. At the same time, it represents a type of functions that receive a continuation of type $\neg B$ and return a continuation of type $\neg A$. A function has an implicative and contrapositive type together.

A neutral type A, B forms the inner structure of types. A neutral type is either true $\top$, false $\bot$, integer int, conjunction $A \wedge B$, disjunction $A \vee B$, implication $A \rightarrow B$, or minus $A - B$.[2] The type $+\top$ is a type of () and $\neg\bot$ is a type of {}.

2.3 Typing Rules

The typing rules of SLC are shown in Figure 3. The left column shows typing rules for expressions, while the right for continuations. Let Γ be a type environment. We write $\Gamma \vdash e : +A$ to mean that an expression e has a type $+A$ under Γ, and similarly for a continuation and for a function.

The type environment Γ is defined as a set of type bindings var : T, where all the variables occurring in a type environment have to be distinct. Given a type environment Γ, we define Γ_p as a type environment where bindings for variables in the pattern p are removed (Figure 4).

The typing rules for expressions are mostly standard. To support a nested pattern for abstractions, the judgement $\Gamma \vdash_p p : +A$ for expression patterns (and $\Gamma \vdash_q q : \neg B$ for continuation patterns) is introduced, which is defined in the second part of Figure 3. The typing rules for continuations are obtained by taking the dual of the ones for expressions. In the rule $\overline{\texttt{TOr}}$, a pair of continuations is given a disjunctive continuation type. Logically, it is a de Morgan dual of $\neg A \wedge \neg B$. The rule $\overline{\texttt{TApp}}$ says that $f \downarrow c$ as a whole can be regarded as a continuation of type $\neg A$: it receives a value of type $+A$, turns it

[2]The intuitive logical meaning of minus operator $A - B$ is $\neg(\neg B \rightarrow \neg A)$. Filinski expresses this type as $[A \leftarrow B]$ [6].

$$\frac{}{\Gamma_g \cup \{g : A \supset B\} \vdash g : A \supset B}\;\texttt{TFVar}$$

$$\frac{}{\Gamma_x \cup \{x : +A\} \vdash x : +A}\;\texttt{TVar} \qquad \frac{}{\Gamma_y \cup \{y : \neg B\} \vdash y : \neg B}\;\overline{\texttt{TVar}}$$

$$\frac{}{\Gamma \vdash n : +\mathsf{int}}\;\texttt{TInt} \quad \frac{}{\Gamma \vdash () : +\top}\;\texttt{TUnit} \qquad \frac{}{\Gamma \vdash \{\} : \neg\bot}\;\overline{\texttt{TUnit}} \quad \frac{}{\Gamma \vdash \bullet : \neg\mathsf{int}}\;\overline{\texttt{TInt}}$$

$$\frac{\Gamma \vdash e_1 : +A \quad \Gamma \vdash e_2 : +B}{\Gamma \vdash (e_1,\, e_2) : +(A \wedge B)}\;\texttt{TAnd} \qquad \frac{\Gamma \vdash c_1 : \neg A \quad \Gamma \vdash c_2 : \neg B}{\Gamma \vdash \{c_1,\, c_2\} : \neg(A \vee B)}\;\overline{\texttt{TOr}}$$

$$\frac{\Gamma \vdash_p p : +A \quad \Gamma \vdash e : +B}{\Gamma_p \vdash p \Rightarrow e : A \supset B}\;\texttt{TFun} \qquad \frac{\Gamma \vdash c : \neg A \quad \Gamma \vdash_q q : \neg B}{\Gamma_q \vdash c \Leftarrow q : A \supset B}\;\overline{\texttt{TFun}}$$

$$\frac{\Gamma \vdash e : +A \quad \Gamma \vdash f : A \supset B}{\Gamma \vdash e \uparrow f : +B}\;\texttt{TApp} \qquad \frac{\Gamma \vdash f : A \supset B \quad \Gamma \vdash c : \neg B}{\Gamma \vdash f \downarrow c : \neg A}\;\overline{\texttt{TApp}}$$

$$\frac{\Gamma \vdash f : A \supset B}{\Gamma \vdash \lceil f \rceil : +(A \to B)}\;\texttt{TFClo} \qquad \frac{\Gamma \vdash f : A \supset B}{\Gamma \vdash \lfloor f \rfloor : \neg(A - B)}\;\overline{\texttt{TFClo}}$$

$$\frac{\Gamma \vdash e : +(A \to B)}{\Gamma \vdash \overline{e} : A \supset B}\;\texttt{TCFun} \qquad \frac{\Gamma \vdash c : \neg(A - B)}{\Gamma \vdash \underline{c} : A \supset B}\;\overline{\texttt{TCFun}}$$

$$\frac{}{\Gamma \vdash_p () : +\top}\;\texttt{PUnit} \qquad \frac{}{\Gamma \vdash_q \{\} : \neg\bot}\;\overline{\texttt{QUnit}}$$

$$\frac{}{\Gamma_x \cup \{x : +A\} \vdash_p x : +A}\;\texttt{PVar} \qquad \frac{}{\Gamma_y \cup \{y : \neg B\} \vdash_q y : \neg B}\;\overline{\texttt{QVar}}$$

$$\frac{}{\Gamma_g \cup \{g : A \supset B\} \vdash_p \lceil g \rceil : +(A \to B)}\texttt{PFClo} \qquad \frac{}{\Gamma_g \cup \{g : A \supset B\} \vdash_q \lfloor g \rfloor : \neg(A - B)}\overline{\texttt{QFClo}}$$

$$\frac{\Gamma_{p_2} \vdash_p p_1 : +A \quad \Gamma_{p_1} \vdash_p p_2 : +B}{\Gamma \vdash_p (p_1,\, p_2) : +(A \wedge B)}\;\texttt{PAnd} \qquad \frac{\Gamma_{q_2} \vdash_q q_1 : \neg A \quad \Gamma_{q_1} \vdash_q q_2 : \neg B}{\Gamma \vdash_q \{q_1,\, q_2\} : \neg(A \vee B)}\;\overline{\texttt{QOr}}$$

$$\frac{\vdash e : +A \quad \vdash c : \neg A}{\vdash \langle\, e \mid c \,\rangle}\;\texttt{TProg1} \qquad \frac{\vdash e : +A \quad \vdash f : A \supset B \quad \vdash c : \neg B}{\vdash \langle\, e \mid f \mid c \,\rangle}\;\texttt{TProg2}$$

Figure 3: Typing Rules for SLC

$$\Gamma_{\mathsf{var}} = \{(\mathsf{var}' : T) \in \Gamma \mid \mathsf{var}' \neq \mathsf{var}\}$$

$$\begin{array}{rcl@{\qquad}rcl}
\Gamma_{()} & = & \Gamma & \Gamma_{\{\}} & = & \Gamma \\
\Gamma_{\lceil g \rceil} & = & \Gamma_g & \Gamma_{\lfloor g \rfloor} & = & \Gamma_g \\
\Gamma_{(p_1,\, p_2)} & = & (\Gamma_{p_1})_{p_2} & \Gamma_{\{q_1,\, q_2\}} & = & (\Gamma_{q_1})_{q_2}
\end{array}$$

Figure 4: Definition of Environment Removed Patterns

$$\begin{array}{lrcll}
(\mathit{begin}) & e : +\mathsf{int} & \rightsquigarrow & \langle\ e \mid \bullet\ \rangle & \\
(\mathit{left}) & \langle\ (e_1,\, e_2) \mid c\ \rangle & \rightsquigarrow & \langle\ e_1 \mid (x \Rightarrow (x,\, e_2)) \downarrow c\ \rangle & \\
(\mathit{right}) & \langle\ (e_1,\, e_2) \mid c\ \rangle & \rightsquigarrow & \langle\ e_2 \mid (x \Rightarrow (e_1,\, x)) \downarrow c\ \rangle & \\
(\mathit{pop}) & \langle\ e \uparrow f \mid c\ \rangle & \rightsquigarrow & \langle\ e \mid f \mid c\ \rangle & \\
(\mathit{push}) & \langle\ e \mid \underline{f} \mid c\ \rangle & \rightsquigarrow & \langle\ e \mid f \downarrow c\ \rangle & \\
(\mathit{exc}) & \langle\ e \mid \overline{e'} \mid c\ \rangle & \rightsquigarrow & \langle\ e' \mid (\lceil g \rceil \Rightarrow e \uparrow g) \downarrow c\ \rangle & \\
(\beta L) & \langle\ e \mid p \Rightarrow e' \mid c\ \rangle & \rightsquigarrow & \langle\ e'\, [\mathcal{L}[\![p]\!] \mapsto e] \mid c\ \rangle & \\
(\beta R) & \langle\ e \mid p \Rightarrow e' \mid c\ \rangle & \rightsquigarrow & \langle\ e'\, [\mathcal{R}[\![p]\!] \mapsto e] \mid c\ \rangle & \text{if } e \text{ matches } p \\
(\overline{\beta R}) & \langle\ e \mid c' \Leftarrow q \mid c\ \rangle & \rightsquigarrow & \langle\ e \mid c'\, [\overline{\mathcal{R}}[\![q]\!] \mapsto c]\ \rangle & \text{if } c \text{ matches } q \\
(\overline{\beta L}) & \langle\ e \mid c' \Leftarrow q \mid c\ \rangle & \rightsquigarrow & \langle\ e \mid c'\, [\overline{\mathcal{L}}[\![q]\!] \mapsto c]\ \rangle & \\
(\overline{\mathit{exc}}) & \langle\ e \mid \underline{c'} \mid c\ \rangle & \rightsquigarrow & \langle\ e \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor) \mid c'\ \rangle & \\
(\overline{\mathit{push}}) & \langle\ e \mid f \mid c\ \rangle & \rightsquigarrow & \langle\ e \uparrow f \mid c\ \rangle & \\
(\overline{\mathit{pop}}) & \langle\ e \mid f \downarrow c\ \rangle & \rightsquigarrow & \langle\ e \mid f \mid c\ \rangle & \\
(\overline{\mathit{right}}) & \langle\ e \mid \{c_1,\, c_2\}\ \rangle & \rightsquigarrow & \langle\ e \uparrow (\{c_1,\, y\} \Leftarrow y) \mid c_2\ \rangle & \\
(\overline{\mathit{left}}) & \langle\ e \mid \{c_1,\, c_2\}\ \rangle & \rightsquigarrow & \langle\ e \uparrow (\{y,\, c_2\} \Leftarrow y) \mid c_1\ \rangle & \\
(\overline{\mathit{end}}) & \langle\ n \mid \bullet\ \rangle & \rightsquigarrow & n &
\end{array}$$

Figure 5: Non-Deterministic Small-Step Semantics of SLC

into a value of type $+B$ using f, and passes the result to c. In other words, the function f transforms a continuation of type $\neg B$ to a continuation of type $\neg A$, effectively acting as a function from $\neg B$ to $\neg A$.

The rules `TProg1` and `TProg2` at the bottom of Figure 3 are the typing rules for configurations. A configuration is well-typed if all of its components are well-typed under an empty environment.

In this type system, the function $\mathsf{call/cc}$ has a function type $((A \to B) \to A) \supset A$ (what we call Peirce's Law).

2.4 Reduction Rules (Non-Deterministic)

In the pure lambda-calculus, reduction rules are written as a binary relation between terms because it consists of only terms. SLC operates not only on expressions but also on continuations. Therefore, reduction rules of SLC are written as a binary relation between configurations, $\langle\ e \mid c\ \rangle$ or $\langle\ e \mid f \mid c\ \rangle$. Figure 5 shows the (non-deterministic) reduction rules of SLC.

In the standard lambda-calculus, reduction proceeds by decomposing the input into a redex and a context, reducing the redex, and plugging the result into the context. In SLC, such decomposition and plugging are described within the reduction rules. If the expression e in a configuration $\langle\ e \mid c\ \rangle$ is an application, its function part is popped from the application via (pop) and the focus of reduction moves to the function part. In case we do not want to perform β-reduction of the popped function right now, the rule (push) pushes the function to the continuation part and the argument is further reduced. In case the function is not a value, on the other hand, the rule (exc) is used to exchange e and e' to move the focus to e'. After e' is reduced to $\lceil f \rceil$, the function $\lceil g \rceil \Rightarrow e \uparrow g$ brings it back into an application. Similarly, (left) and (right) reduce the left and right elements of a pair, respectively.

SLC has two kinds of β-reduction, eager one (βR) and lazy one (βL), depending on how to handle pattern matching. Eager β-reduction requires the argument to match the pattern at reduction time, deconstructs it, and binds variables in the pattern to each component using $\mathcal{R}$ defined in Figure 6. Lazy one does not deconstruct the argument but binds variables in the pattern to code that deconstructs the argument when the variables are used later (see the definition of $\mathcal{L}$ in Figure 6). The former is required to perform the real deconstruction of patterns, while the latter enables to defer pattern matching as long as possible (like the irrefutable pattern in Haskell).

The reduction rules for continuations are defined completely symmetrically to those for expressions. In particular, we can freely capture the current continuation using continuation abstractions, just as we can freely receive the current expression using expression abstractions.

As an example of reduction, we can capture the current continuation in x by passing a function $\lceil x \Rightarrow e \rceil$ to $\mathsf{call/cc}$. See Figure 7.

$$\begin{array}{rcl}
[\mathcal{R}[\![x]\!] \mapsto e] & = & [e/x] \\
[\mathcal{R}[\![()]\!] \mapsto e] & = & [] \\
[\mathcal{R}[\![\lceil g \rceil]\!] \mapsto \lceil f \rceil] & = & [f/g] \\
[\mathcal{R}[\![(p_1,\, p_2)]\!] \mapsto (e_1,\, e_2)] & = & [\mathcal{R}[\![p_1]\!] \mapsto e_1][\mathcal{R}[\![p_2]\!] \mapsto e_2] \\
[\overline{\mathcal{R}}[\![\{q_1,\, q_2\}]\!] \mapsto \{c_1,\, c_2\}] & = & [\overline{\mathcal{R}}[\![q_1]\!] \mapsto c_1][\overline{\mathcal{R}}[\![q_2]\!] \mapsto c_2] \\
[\overline{\mathcal{R}}[\![\lfloor g \rfloor]\!] \mapsto \lfloor f \rfloor] & = & [f/g] \\
[\overline{\mathcal{R}}[\![\{\}]\!] \mapsto c] & = & [] \\
[\overline{\mathcal{R}}[\![y]\!] \mapsto c] & = & [c/y]
\end{array}$$

$$\begin{array}{rcl}
[\mathcal{L}[\![x]\!] \mapsto e] & = & [e/x] \\
[\mathcal{L}[\![()]\!] \mapsto e] & = & [] \\
[\mathcal{L}[\![\lceil g \rceil]\!] \mapsto e] & = & [\overline{e}/g] \\
[\mathcal{L}[\![(p_1,\, p_2)]\!] \mapsto e] & = & [\mathcal{L}[\![p_1]\!] \mapsto e \uparrow ((x_1,\, x_2) \Rightarrow x_1)][\mathcal{L}[\![p_2]\!] \mapsto e \uparrow ((x_1,\, x_2) \Rightarrow x_1)] \\
[\overline{\mathcal{L}}[\![\{q_1,\, q_2\}]\!] \mapsto c] & = & [\overline{\mathcal{L}}[\![q_1]\!] \mapsto (y_1 \Leftarrow \{y_1,\, y_2\}) \downarrow c][\overline{\mathcal{L}}[\![q_2]\!] \mapsto (y_2 \Leftarrow \{y_1,\, y_2\}) \downarrow c] \\
[\overline{\mathcal{L}}[\![\lfloor g \rfloor]\!] \mapsto c] & = & [\underline{c}/g] \\
[\overline{\mathcal{L}}[\![\{\}]\!] \mapsto c] & = & [] \\
[\overline{\mathcal{L}}[\![y]\!] \mapsto c] & = & [c/y]
\end{array}$$

Figure 6: Definition of Eager and Lazy Substitution

$$\begin{array}{lll}
 & \langle\ \lceil x \Rightarrow e \rceil \mid \textsf{call/cc} \mid c\ \rangle & \\
\rightsquigarrow & \langle\ \lceil x \Rightarrow e \rceil \mid (\lceil g \rceil \Rightarrow \lceil y \Leftarrow \_ \rceil \uparrow g) \downarrow y \Leftarrow y \mid c\ \rangle & (\text{definition of } \textsf{call/cc}) \\
\rightsquigarrow & \langle\ \lceil x \Rightarrow e \rceil \mid (\lceil g \rceil \Rightarrow \lceil c \Leftarrow \_ \rceil \uparrow g) \downarrow c\ \rangle & (\overline{\beta R}) \\
\rightsquigarrow & \langle\ \lceil x \Rightarrow e \rceil \mid \lceil g \rceil \Rightarrow \lceil c \Leftarrow \_ \rceil \uparrow g \mid c\ \rangle & (\overline{\mathit{pop}}) \\
\rightsquigarrow & \langle\ \lceil c \Leftarrow \_ \rceil \uparrow (x \Rightarrow e) \mid c\ \rangle & (\beta R) \\
\rightsquigarrow & \langle\ \lceil c \Leftarrow \_ \rceil \mid x \Rightarrow e \mid c\ \rangle & (\mathit{pop}) \\
\rightsquigarrow & \langle\ e[\lceil c \Leftarrow \_ \rceil/x] \mid c\ \rangle & (\beta R)
\end{array}$$

Figure 7: Example Reduction of call/cc

There can be multiple rules that are applicable to the same configuration. Furthermore, the reduction rules are not Church-Rosser, that is, the result of evaluation can be different values. For example, the configuration $\langle\ 1 \uparrow (\bullet \Leftarrow y) \mid (x \Rightarrow 2) \downarrow \bullet\ \rangle$ has the following two different reductions:

$$\begin{array}{lllllll}
 & \langle\ 1 \uparrow (\bullet \Leftarrow y) \mid (x \Rightarrow 2) \downarrow \bullet\ \rangle & & & \langle\ 1 \uparrow (\bullet \Leftarrow y) \mid (x \Rightarrow 2) \downarrow \bullet\ \rangle & \\
\rightsquigarrow & \langle\ 1 \mid \bullet \Leftarrow y \mid (x \Rightarrow 2) \downarrow \bullet\ \rangle & (\mathit{pop}) & \rightsquigarrow & \langle\ 1 \uparrow (\bullet \Leftarrow y) \mid (x \Rightarrow 2) \mid \bullet\ \rangle & (\overline{\mathit{pop}}) \\
\rightsquigarrow & \langle\ 1 \mid \bullet\ \rangle & (\overline{\beta}) & \rightsquigarrow & \langle\ 2 \mid \bullet\ \rangle & (\beta) \\
\rightsquigarrow & 1 & (\overline{\mathit{end}}) & \rightsquigarrow & 2 & (\overline{\mathit{end}})
\end{array}$$

This is not surprising. Without fixing the evaluation strategy, the result can be different. In a functional language, for example, $(\lambda x.1)(3/0)$ is reduced to 1 in CBN and an error in CBV. To recover uniqueness of evaluation, we introduce CBV evaluation strategy into SLC in Section 3 and CBN one in Section 4.

2.5 Properties of non-deterministic SLC

Without specifying the evaluation strategy, the small-step reduction semantics for non-deterministic SLC is sound with respect to the type system. Let $\langle ... \rangle$ (possibly with a subscript) denote either a two-place configuration $\langle\ e \mid c\ \rangle$ or a three-place configuration $\langle\ e \mid f \mid c\ \rangle$. We can show the progress by simple case analysis and the preservation using the substitution lemma.

Theorem 2.1 (Progress)
If $\vdash \langle ... \rangle_1$, then $\langle ... \rangle_1 \rightsquigarrow \langle ... \rangle_2$ for some $\langle ... \rangle_2$ or $\langle ... \rangle_1 = \langle\ n \mid \bullet\ \rangle$ for some n.

$$\begin{array}{rrcl}
\text{value} & v & ::= & n \mid x \mid () \mid (v,\, v) \mid \lceil f \rceil \mid [v \uparrow \mathit{inl}] \mid [v \uparrow \mathit{inr}] \mid [v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)] \\
\text{expression} & e & ::= & v \mid (e,\, e) \mid e \uparrow f \\
\text{function} & f & ::= & g \mid p \Rightarrow e \mid c \Leftarrow q \mid \overline{e} \mid \underline{c} \\
\text{continuation} & c & ::= & \bullet \mid y \mid \{\} \mid \{c,\, c\} \mid f \downarrow c \mid \lfloor f \rfloor \mid [\mathit{inl} \downarrow c] \mid [\mathit{inr} \downarrow c]
\end{array}$$

Figure 8: Syntax of CBV SLC

$$\frac{\Gamma \vdash v : +A}{\Gamma \vdash [v \uparrow \mathit{inl}] : +(A \vee B)}\ \texttt{TInl} \qquad \frac{\Gamma \vdash c : \neg(A \vee B)}{\Gamma \vdash [\mathit{inl} \downarrow c] : \neg A}\ \overline{\texttt{TInl}}$$

$$\frac{\Gamma \vdash v : +B}{\Gamma \vdash [v \uparrow \mathit{inr}] : +(A \vee B)}\ \texttt{TInr} \qquad \frac{\Gamma \vdash c : \neg(A \vee B)}{\Gamma \vdash [\mathit{inr} \downarrow c] : \neg B}\ \overline{\texttt{TInr}}$$

$$\frac{\Gamma \vdash v : +A \quad \Gamma \vdash c : \neg B}{\Gamma \vdash [v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)] : +(A - B)}\ \texttt{TCtx}$$

Figure 9: Additional Typing Rules for CBV SLC

Theorem 2.2 (Preservation)
Assume $\vdash \langle ... \rangle_1$. If $\langle ... \rangle_1 \rightsquigarrow \langle ... \rangle_2$, then $\vdash \langle ... \rangle_2$.

3 CBV SLC

The reduction rules in Figure 5 are non-deterministic. In this section, we introduce the CBV evaluation strategy into SLC. Under CBV, an argument is evaluated to a value before β-reduction. In SLC, it means that the evaluation goes from the expression side of the configuration $\langle ... \rangle$, or from left to right of $\langle ... \rangle$. To enforce this evaluation strategy, we first introduce a value into the syntax of SLC as in Figure 8. In addition to integers, variables, units, pairs of values, and higher-order functions, the value contains three *frozen* values that are expressions enclosed in brackets. In the non-deterministic setting, they are operationally the same as the expressions without brackets using the following interpretation:

$$\begin{array}{rcl}
\mathit{inl} & = & y_1 \Leftarrow \{y_1,\, y_2\} \\
\mathit{inr} & = & y_2 \Leftarrow \{y_1,\, y_2\}
\end{array}$$

In the CBV setting, they are used to control the order of evaluation. Likewise, we have introduced two frozen continuations. The typing rules for the frozen values and frozen continuations are the same as those without brackets. They are summarized in Figure 9.

Currently, values include a function value of the form $\lceil \overline{e} \rceil$. If we want to exclude this case from values, we could separate f into value functions and non-value functions, and include only the former into values.

Figure 10 shows the reduction rules for CBV SLC. The rule names with the subscript v indicate the rules that are changed from non-deterministic ones (Figure 5). The rule names with primes are the reduction rules for frozen values/continuations. The rules marked with $*$ are ones directly obtained from Filinski's denotational semantics. We will explain it in detail in Section 5.2. Rules in Figure 10 are restriction of the reduction rules for the non-deterministic SLC in a way the evaluation order is fixed to CBV: evaluation goes from left to right in the configuration $\langle ... \rangle$. For example, the rule (βR_{v}) requires that the argument is fully evaluated (and hence the use of eager pattern deconstruction $\mathcal{R}$); the rules ($\mathit{left}_{\mathrm{v}}$) and ($\mathit{right}_{\mathrm{v}}$) force the left-to-right evaluation order for pairs. Rules for frozen values/continuations first arise in ($\overline{\mathit{exc}_{\mathrm{v}}}$) when a continuation appears as a function. In this case, we freeze v and c into $[v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)]$ and evaluate c'. Here, we cannot remove the bracket (as in ($\overline{\mathit{exc}}$) in

$$\begin{array}{clrcll}
 & (\mathit{begin}) & e : +\mathsf{int} & \rightsquigarrow & \langle\, e \mid \bullet \,\rangle & \\
 & (\mathit{left}_{\mathrm{v}}) & \langle\, (e_1,\, e_2) \mid c \,\rangle & \rightsquigarrow & \langle\, e_1 \mid (x \Rightarrow (x,\, e_2)) \downarrow c \,\rangle & \text{if } e_1 \neq v \\
 & (\mathit{right}_{\mathrm{v}}) & \langle\, (v_1,\, e_2) \mid c \,\rangle & \rightsquigarrow & \langle\, e_2 \mid (x \Rightarrow (v_1,\, x)) \downarrow c \,\rangle & \text{if } e_2 \neq v \\
 & (\mathit{pop}) & \langle\, e \uparrow f \mid c \,\rangle & \rightsquigarrow & \langle\, e \mid f \mid c \,\rangle & \\
 & (\mathit{push}_{\mathrm{v}}) & \langle\, e \mid \underline{f} \mid c \,\rangle & \rightsquigarrow & \langle\, e \mid f \downarrow c \,\rangle & \text{if } e \neq v \\
* & (\mathit{exc}_{\mathrm{v}}) & \langle\, v \mid \overline{e'} \mid c \,\rangle & \rightsquigarrow & \langle\, e' \mid (\lceil g \rceil \Rightarrow v \uparrow g) \downarrow c \,\rangle & \\
* & (\mathit{inl}') & \langle\, [v \uparrow \mathit{inl}] \mid \{c_1,\, c_2\} \,\rangle & \rightsquigarrow & \langle\, v \mid c_1 \,\rangle & \\
* & (\mathit{inr}') & \langle\, [v \uparrow \mathit{inr}] \mid \{c_1,\, c_2\} \,\rangle & \rightsquigarrow & \langle\, v \mid c_2 \,\rangle & \\
* & (\mathit{contx}') & \langle\, [v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)] \mid \lfloor f \rfloor \,\rangle & \rightsquigarrow & \langle\, v \mid f \mid c \,\rangle & \\
* & (\beta R_{\mathrm{v}}) & \langle\, v \mid p \Rightarrow e' \mid c \,\rangle & \rightsquigarrow & \langle\, e' \, [\mathcal{R}[\![p]\!] \mapsto v] \mid c \,\rangle & \\
* & (\overline{\beta L_{\mathrm{v}}}) & \langle\, v \mid c' \Leftarrow q \mid c \,\rangle & \rightsquigarrow & \langle\, v \mid c' \, [\overline{\mathcal{L}}_{\mathrm{v}}[\![q]\!] \mapsto c] \,\rangle & \\
* & (\overline{\mathit{inr}'}) & \langle\, v \mid [\mathit{inr} \downarrow c] \,\rangle & \rightsquigarrow & \langle\, [v \uparrow \mathit{inr}] \mid c \,\rangle & \\
* & (\overline{\mathit{inl}'}) & \langle\, v \mid [\mathit{inl} \downarrow c] \,\rangle & \rightsquigarrow & \langle\, [v \uparrow \mathit{inl}] \mid c \,\rangle & \\
* & (\overline{\mathit{exc}_{\mathrm{v}}}) & \langle\, v \mid \underline{c'} \mid c \,\rangle & \rightsquigarrow & \langle\, [v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)] \mid c' \,\rangle & \\
* & (\overline{\mathit{pop}_{\mathrm{v}}}) & \langle\, v \mid f \downarrow c \,\rangle & \rightsquigarrow & \langle\, v \mid f \mid c \,\rangle & \\
* & (\overline{\mathit{end}}) & \langle\, n \mid \bullet \,\rangle & \rightsquigarrow & n &
\end{array}$$

Figure 10: Small-Step Semantics for CBV SLC

$$\begin{array}{rcl}
[\overline{\mathcal{L}}_{\mathrm{v}}[\![\{q_1,\, q_2\}]\!] \mapsto c] & = & [\overline{\mathcal{L}}_{\mathrm{v}}[\![q_1]\!] \mapsto [\mathit{inl} \downarrow c]][\overline{\mathcal{L}}_{\mathrm{v}}[\![q_2]\!] \mapsto [\mathit{inr} \downarrow c]] \\
[\overline{\mathcal{L}}_{\mathrm{v}}[\![\lfloor g \rfloor]\!] \mapsto c] & = & [\underline{c}/g] \\
[\overline{\mathcal{L}}_{\mathrm{v}}[\![\{\}]\!] \mapsto c] & = & [] \\
[\overline{\mathcal{L}}_{\mathrm{v}}[\![y]\!] \mapsto c] & = & [c/y]
\end{array}$$

Figure 11: Definition of Lazy Substitution for CBV SLC

Figure 5), because it would lead to non-termination in CBV:

$$\begin{array}{rcll}
\langle\, v \mid \underline{c'} \mid c \,\rangle & \rightsquigarrow & \langle\, v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor) \mid c' \,\rangle & (\overline{\mathit{exc}}) \\
 & \rightsquigarrow & \langle\, v \mid g \downarrow c \Leftarrow \lfloor g \rfloor \mid c' \,\rangle & (\mathit{pop}) \\
 & \rightsquigarrow & \langle\, v \mid \underline{c'} \downarrow c \,\rangle & (\overline{\beta L_{\mathrm{v}}}) \\
 & \rightsquigarrow & \langle\, v \mid \underline{c'} \mid c \,\rangle & (\overline{\mathit{pop}_{\mathrm{v}}})
\end{array}$$

By temporarily freezing the application $[v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)]$ as a value, we enforce the evaluation of c'. The frozen value is destructed in (contx') only when the evaluation of c' is finished. Filinski called this frozen value $[v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)]$ a *context*.

Because the evaluation goes from left to right, an interesting asymmetry arises between expression abstractions and continuation abstractions: although the argument to an expression abstraction is always a value in CBV, the argument continuation to a continuation abstraction is not a value in general. In other words, an expression abstraction is evaluated in CBV but a continuation abstraction is evaluated in CBN.

Because the argument continuation in $(\overline{\beta L_{\mathrm{v}}})$ is not evaluated yet, we cannot use the eager pattern deconstruction here. Instead, we use the CBV lazy pattern deconstruction shown in Figure 11. Unlike $\overline{\mathcal{L}}$ (in Figure 6), $\overline{\mathcal{L}}_{\mathrm{v}}$ introduces frozen injections ($[\mathit{inl} \downarrow c]$ and $[\mathit{inr} \downarrow c]$) to defer the pattern deconstruction. They force the evaluation of continuations only when the pattern deconstruction is actually needed in $(\overline{\mathit{inl}'})$ or $(\overline{\mathit{inr}'})$.

3.1 Properties of CBV SLC

Since the CBV reduction semantics is a special case of the non-deterministic reduction semantics, the preservation holds for CBV reduction semantics. As for the progress, we can show by simple case analysis that the CBV reduction semantics satisfies the following stronger property:

$$\begin{array}{rlll}
\text{expression} & e & ::= & n \mid x \mid () \mid (e,\, e) \mid e \uparrow f \mid \lceil f \rceil \mid [e \uparrow \mathit{fst}] \mid [e \uparrow \mathit{snd}] \\
\text{function} & f & ::= & g \mid p \Rightarrow e \mid c \Leftarrow q \mid \overline{e} \mid \underline{c} \\
\text{continuation} & c & ::= & k \mid \{c,\, c\} \mid f \downarrow c \\
\text{covalue} & k & ::= & \bullet \mid y \mid \{\} \mid \{k,\, k\} \mid \lfloor f \rfloor \mid [(\lceil g \rceil \Rightarrow e \uparrow g) \downarrow k] \mid [\mathit{fst} \downarrow k] \mid [\mathit{snd} \downarrow k]
\end{array}$$

Figure 12: Syntax of CBN SLC

$$\frac{\Gamma \vdash e : +(A \wedge B)}{\Gamma \vdash [e \uparrow \mathit{fst}] : +A}\ \texttt{TFst} \qquad \frac{\Gamma \vdash k : \neg A}{\Gamma \vdash [\mathit{fst} \downarrow k] : \neg(A \wedge B)}\ \overline{\texttt{TFst}}$$

$$\frac{\Gamma \vdash e : +(A \wedge B)}{\Gamma \vdash [e \uparrow \mathit{snd}] : +B}\ \texttt{TSnd} \qquad \frac{\Gamma \vdash k : \neg B}{\Gamma \vdash [\mathit{snd} \downarrow k] : \neg(A \wedge B)}\ \overline{\texttt{TSnd}}$$

$$\frac{\Gamma \vdash e : +A \quad \Gamma \vdash k : \neg B}{\Gamma \vdash [(\lceil g \rceil \Rightarrow k \uparrow g) \downarrow e] : \neg(A \rightarrow B)}\ \overline{\texttt{TCtx}}$$

Figure 13: Additional Typing Rules of CBN SLC

Theorem 3.1 (The Uniqueness of Reduction in CBV SLC)
Under CBV, if $\vdash \langle ... \rangle$, then there is exactly one reduction rule applicable to $\langle ... \rangle$.

Furthermore, using the standard logical relation argument (as found in [9, Section 12]) tailored for SLC, we can show the following theorem:

Theorem 3.2 (Termination of Evaluation in CBV SLC)
Under CBV, if $\vdash e : +\mathsf{int}$, then there exists a unique n such that $e \rightsquigarrow^* n$.

4 CBN SLC

The CBN evaluation strategy evaluates expressions only when it is needed by its context. In other words, the evaluation goes from the continuation side, or from right to left of $\langle ... \rangle$. Because of this duality, CBN SLC can be mechanically obtained by repeating the construction of CBV SLC in the previous section with the roles of expressions and continuations swapped: the evaluation goes from the continuation side of the configuration $\langle ... \rangle$, or from right to left of $\langle ... \rangle$. The result is a mirror image of CBV SLC. Furthermore, the same properties hold for CBN SLC.

Dually to CBV SLC, we first introduce a *covalue* (that is a value of continuations) into the syntax of SLC as in Figure 12. It is defined as completely dual notion of the value v in CBV SLC. Therefore the covalue contains three frozen covalues that are continuations enclosed in brackets. In the non-deterministic setting, they are operationally the same as the continuations without brackets using the following interpretation:

$$\begin{array}{rcl}
\mathit{fst} & = & (x_1,\, x_2) \Rightarrow x_1 \\
\mathit{snd} & = & (x_1,\, x_2) \Rightarrow x_1
\end{array}$$

In the CBN setting, they are used to control the order of evaluation. Likewise, we have introduced two frozen expressions. Figure 13 shows typing rules for frozen covalues and frozen expressions.

Figure 14 shows the reduction rules for CBN SLC. The rule names with the subscript n indicate the rules that are changed from non-deterministic ones (Figure 5). The rule names with primes are the reduction rules for frozen expressions/covalues. The rules marked with $*$ are the ones directly obtained from Filinski's denotational semantics. We will explain it in detail in Section 6.2. Rules in Figure 14 are restriction of the reduction rules for the non-deterministic SLC in a way the evaluation order is fixed to CBN: evaluation goes from right to left in the configuration $\langle ... \rangle$.

$$
\begin{array}{clrcll}
 & (\mathit{begin}) & e : +\mathsf{int} & \rightsquigarrow & \langle\, e \mid \bullet \,\rangle & \\
* & (\mathit{pop}_{\mathrm{n}}) & \langle\, e \uparrow \underline{f} \mid k \,\rangle & \rightsquigarrow & \langle\, e \mid f \mid k \,\rangle & \\
* & (\mathit{exc}_{\mathrm{n}}) & \langle\, e \mid \overline{e'} \mid k \,\rangle & \rightsquigarrow & \langle\, e' \mid [(\lceil g \rceil \Rightarrow e \uparrow g) \downarrow k] \,\rangle & \\
* & (\mathit{fst}') & \langle\, [e \uparrow \mathit{fst}] \mid k \,\rangle & \rightsquigarrow & \langle\, e \mid [\mathit{fst} \downarrow k] \,\rangle & \\
* & (\mathit{snd}') & \langle\, [e \uparrow \mathit{snd}] \mid k \,\rangle & \rightsquigarrow & \langle\, e \mid [\mathit{snd} \downarrow k] \,\rangle & \\
* & (\beta L_{\mathrm{n}}) & \langle\, e \mid p \Rightarrow e' \mid k \,\rangle & \rightsquigarrow & \langle\, e'[\mathcal{L}_{\mathrm{n}}[\![p]\!] \mapsto e] \mid k \,\rangle & \\
* & (\overline{\beta R_{\mathrm{n}}}) & \langle\, e \mid c' \Leftarrow q \mid k \,\rangle & \rightsquigarrow & \langle\, e \mid c'[\overline{\mathcal{R}}[\![q]\!] \mapsto k] \,\rangle & \\
* & (\overline{\mathit{contx}'}) & \langle\, \lceil f \rceil \mid [(\lceil g \rceil \Rightarrow e \uparrow g) \downarrow k] \,\rangle & \rightsquigarrow & \langle\, e \mid f \mid k \,\rangle & \\
* & (\overline{\mathit{snd}'}) & \langle\, (e_1,\, e_2) \mid [\mathit{snd} \downarrow k] \,\rangle & \rightsquigarrow & \langle\, e_2 \mid k \,\rangle & \\
* & (\overline{\mathit{fst}'}) & \langle\, (e_1,\, e_2) \mid [\mathit{fst} \downarrow k] \,\rangle & \rightsquigarrow & \langle\, e_1 \mid k \,\rangle & \\
* & (\overline{\mathit{exc}_{\mathrm{n}}}) & \langle\, e \mid \underline{c'} \mid k \,\rangle & \rightsquigarrow & \langle\, e \uparrow (g \downarrow k \Leftarrow \lfloor g \rfloor) \mid c' \,\rangle & \\
 & (\overline{\mathit{push}_{\mathrm{n}}}) & \langle\, e \mid f \mid c \,\rangle & \rightsquigarrow & \langle\, e \uparrow f \mid c \,\rangle & \text{if } c \neq k \\
 & (\overline{\mathit{pop}}) & \langle\, e \mid f \downarrow c \,\rangle & \rightsquigarrow & \langle\, e \mid f \mid c \,\rangle & \\
 & (\overline{\mathit{right}_{\mathrm{n}}}) & \langle\, e \mid \{c_1,\, c_2\} \,\rangle & \rightsquigarrow & \langle\, e \uparrow (\{c_1,\, y\} \Leftarrow y) \mid c_2 \,\rangle & \text{if } c_2 \neq k \\
 & (\overline{\mathit{left}_{\mathrm{n}}}) & \langle\, e \mid \{c_1,\, k_2\} \,\rangle & \rightsquigarrow & \langle\, e \uparrow (\{y,\, k_2\} \Leftarrow y) \mid c_1 \,\rangle & \text{if } c_1 \neq k \\
* & (\overline{\mathit{end}}) & \langle\, n \mid \bullet \,\rangle & \rightsquigarrow & n &
\end{array}
$$

Figure 14: Small-Step Semantics for CBN SLC

$$
\begin{array}{rcl}
[\mathcal{L}_{\mathrm{n}}[\![x]\!] \mapsto e] & = & [e/x] \\
[\mathcal{L}_{\mathrm{n}}[\![()]\!] \mapsto e] & = & [] \\
[\mathcal{L}_{\mathrm{n}}[\![\lceil g \rceil]\!] \mapsto e] & = & [\overline{e}/g] \\
[\mathcal{L}_{\mathrm{n}}[\![(p_1,\, p_2)]\!] \mapsto e] & = & [\mathcal{L}_{\mathrm{n}}[\![p_1]\!] \mapsto [e \uparrow \mathit{fst}]][\mathcal{L}_{\mathrm{n}}[\![p_2]\!] \mapsto [e \uparrow \mathit{snd}]]
\end{array}
$$

Figure 15: Definition of Lazy Substitution for CBN SLC

4.1 Properties of CBN SLC

Since the CBN reduction semantics is a special case of the non-deterministic reduction semantics, the preservation holds for CBN reduction semantics. As for the progress, we can show by simple case analysis that the CBN reduction semantics satisfies the following stronger property:

Theorem 4.1 (The Uniqueness of Reduction in CBN SLC)
Under CBN, if $\vdash \langle \ldots \rangle$, then there is exactly one reduction rule applicable to $\langle \ldots \rangle$.

Furthermore, using the standard logical relation argument (as found in [9, Section 12]) tailored for SLC, we can show the following theorem:

Theorem 4.2 (Termination of Evaluation in CBN SLC)
Under CBN, if $\vdash e : +\mathsf{int}$, then there exists a unique n such that $e \rightsquigarrow^* n$.

5 Functional Correspondence for CBV SLC

In this section, we show that the CBV small-step semantics of SLC presented in Section 3 corresponds to Filinski's original definition of CBV SLC given as a denotational semantics. The method we use is the functional correspondence of interpreters and abstract machines developed by Danvy and his group [1, 4]: an abstract machine is a CPS-transformed defunctionalized interpreter. In our case, since Filinski's denotational semantics is already in (a kind of) CPS, the small-step semantics (an abstract machine) can be obtained by defunctionalizing the denotational semantics.

Figure 16 shows Filinski's denotational semantics for CBV SLC. Since the syntax of SLC consists of three kinds, the semantics also consists of three functions: $\mathcal{E}$, $\mathcal{C}$, and $\mathcal{F}$. In Filinski's formulation, the patterns do not contain the function pattern. The semantic functions are not expressed in the symmetric manner, because SLC is mapped into the standard (asymmetric) lambda-calculus. In the semantics, ρ

$$\begin{array}{rcl}
\mathit{Val} & = & \mathit{Int} + \mathit{Unit}() + \mathit{Pair}(\mathit{Val} \times \mathit{Val}) + \mathit{In}_1(\mathit{Val}) + \mathit{In}_2(\mathit{Val}) + \\
 & & \mathit{Closr}(\mathit{Val} \to \mathit{Cnt} \to \mathit{Ans}) + \mathit{Contx}(\mathit{Val} \times \mathit{Cont}) \\
\mathit{Cnt} & = & \mathit{Val} \to \mathit{Ans} \\
\mathit{Env} & = & \mathit{Ide} \to (\mathit{Val} + \mathit{Cnt})
\end{array}$$

$$\begin{array}{rcl}
\mathcal{E} : e \to \mathit{Env} \to \mathit{Cnt} & \to & \mathit{Ans} \\
\mathcal{E}\llbracket n \rrbracket \rho\kappa & = & \kappa\ n \\
\mathcal{E}\llbracket x \rrbracket \rho\kappa & = & \mathit{let}\ \mathit{val}(v) = \rho\ x\ \mathit{in}\ \kappa\ v \\
\mathcal{E}\llbracket (e_1,\, e_2) \rrbracket \rho\kappa & = & \mathcal{E}\llbracket e_1 \rrbracket\, \rho\, (\lambda v_1.\mathcal{E}\llbracket e_2 \rrbracket\, \rho\, (\lambda v_2.\kappa\ \mathit{pair}(v_1, v_2))) \\
\mathcal{E}\llbracket () \rrbracket \rho\kappa & = & \kappa\ \mathit{unit}() \\
\mathcal{E}\llbracket e \uparrow f \rrbracket \rho\kappa & = & \mathcal{E}\llbracket e \rrbracket\, \rho\, (\lambda v.\mathcal{F}\llbracket f \rrbracket \rho v \kappa) \\
\mathcal{E}\llbracket \lceil f \rceil \rrbracket \rho\kappa & = & \kappa\ \mathit{closr}(\lambda v \kappa'.\mathcal{F}\llbracket f \rrbracket \rho v \kappa')
\end{array}$$

$$\begin{array}{rcl}
\mathcal{C} : c \to \mathit{Env} \to \mathit{Val} & \to & \mathit{Ans} \\
\mathcal{C}\llbracket y \rrbracket \rho v & = & \mathit{let}\ \mathit{cnt}(\kappa) = \rho\ y\ \mathit{in}\ \kappa\ v \\
\mathcal{C}\llbracket \{c_1,\, c_2\} \rrbracket \rho v & = & \mathit{case}\ v\ \mathit{of}\ \mathit{in}_1(v') : \mathcal{C}\llbracket c_1 \rrbracket \rho v' \mid \mathit{in}_2(v') : \mathcal{C}\llbracket c_2 \rrbracket \rho v'\ \mathit{esac} \\
\mathcal{C}\llbracket \{\} \rrbracket \rho v & = & \mathit{case}\ v\ \mathit{of}\ \mathit{esac} \\
\mathcal{C}\llbracket f \downarrow c \rrbracket \rho v & = & \mathcal{F}\llbracket f \rrbracket\, \rho\, v\, (\lambda v'.\mathcal{C}\llbracket c \rrbracket \rho v') \\
\mathcal{C}\llbracket \lfloor f \rfloor \rrbracket \rho v & = & \mathit{let}\ \mathit{contx}(v', \kappa) = v\ \mathit{in}\ \mathcal{F}\llbracket f \rrbracket \rho v' \kappa
\end{array}$$

$$\begin{array}{rcl}
\mathcal{F} : f \to \mathit{Env} \to \mathit{Val} \to \mathit{Cnt} & \to & \mathit{Ans} \\
\mathcal{F}\llbracket X \Rightarrow e \rrbracket \rho v \kappa & = & \mathcal{E}\llbracket e \rrbracket\, ([\mathcal{X}\llbracket X \rrbracket \mapsto v]\rho)\, \kappa \\
\mathcal{F}\llbracket c \Leftarrow Y \rrbracket \rho v \kappa & = & \mathcal{C}\llbracket c \rrbracket\, ([\mathcal{Y}\llbracket Y \rrbracket \mapsto \kappa]\rho)\, v \\
\mathcal{F}\llbracket \overline{e} \rrbracket \rho v \kappa & = & \mathcal{E}\llbracket e \rrbracket\, \rho\, (\lambda v'.\mathit{let}\ \mathit{closr}(h) = v'\ \mathit{in}\ h\, v\, \kappa) \\
\mathcal{F}\llbracket \underline{c} \rrbracket \rho v \kappa & = & \mathcal{C}\llbracket c \rrbracket\, \rho\, \mathit{contx}(v, \kappa)
\end{array}$$

$$\begin{array}{rcl}
\mathcal{X} : X \to \mathit{Val} \to \mathit{Env} & \to & \mathit{Env} \\
[\mathcal{X}\llbracket x \rrbracket \mapsto v]\rho & = & \rho[x \mapsto \mathit{val}(v)] \\
[\mathcal{X}\llbracket () \rrbracket \mapsto v]\rho & = & \mathit{let}\ \mathit{unit}() = v\ \mathit{in}\ \rho \\
[\mathcal{X}\llbracket (X_1,\, X_2) \rrbracket \mapsto v]\rho & = & \mathit{let}\ \mathit{pair}(v_1, v_2) = v\ \mathit{in}\ [\mathcal{X}\llbracket X_1 \rrbracket \mapsto v_1]([\mathcal{X}\llbracket X_2 \rrbracket \mapsto v_2]\rho)
\end{array}$$

$$\begin{array}{rcl}
\mathcal{Y} : Y \to \mathit{Cnt} \to \mathit{Env} & \to & \mathit{Env} \\
[\mathcal{Y}\llbracket y \rrbracket \mapsto \kappa]\rho & = & \rho[y \mapsto \mathit{cnt}(\kappa)] \\
[\mathcal{Y}\llbracket \{\} \rrbracket \mapsto \kappa]\rho & = & \rho \\
[\mathcal{Y}\llbracket \{Y_1,\, Y_2\} \rrbracket \mapsto \kappa]\rho & = & [\mathcal{Y}\llbracket Y_1 \rrbracket \mapsto \lambda v.\kappa\ \mathit{in}_1(v)]\ ([\mathcal{Y}\llbracket Y_2 \rrbracket \mapsto \lambda v.\kappa\ \mathit{in}_2(v)]\rho)
\end{array}$$

Figure 16: Filinski's Denotational Semantics for CBV SLC

represents an environment, v represents a value (the result of evaluation), and κ represents a *semantic* continuation. Filinski introduces a domain for each value type (e.g., *Pair*) together with a constructor for it (in lower case letter, e.g., *pair*).

We will not go into details of this denotational semantics. Rather, we regard this semantics as a definitional interpreter for SLC and defunctionalize it. Defunctionalization, introduced by Reynolds [10], is a whole-program transformation to remove higher-order functions. Every higher-order function in a program is replaced with a unique constructor whose arguments are free variables of the higher-order function. When the higher-order function is applied, a newly introduced apply function $\mathcal{A}$ is used instead, which, given the actual argument, will execute the body of the higher-order function. This way, all the higher-order functions are replaced with first-order data.

5.1 Defunctionalization

We defunctionalize two kinds of higher-order functions in Figure 16: continuations (κ of type Cnt) and closures (the argument of *closr*).[3] The result of defunctionalization is found in the upper left of Figure 17.

[3]The latter is sometimes called closure conversion.

Defunctionalized Semantics	Corresponding Abstract Machine
$\mathcal{E} : e \to Env \to Cnt \to Ans$	The rules coming from $\mathcal{E}$
$\mathcal{E}\llbracket n \rrbracket \rho\kappa = \mathcal{A}\ \kappa\ n$	$\langle\ \llbracket n \rrbracket\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ n \mid \kappa\ \rangle$
$\mathcal{E}\llbracket x \rrbracket \rho\kappa = let\ val(v) = \rho\ x\ in\ \mathcal{A}\ \kappa\ v$	$\langle\ \llbracket x \rrbracket\rho \mid \kappa\ \rangle \rightsquigarrow let\ val(v) = \rho\ x\ in\ \langle\ v \mid \kappa\ \rangle$
$\mathcal{E}\llbracket (e_1,\ e_2) \rrbracket \rho\kappa = \mathcal{E}\llbracket e_1 \rrbracket\ \rho\,(\mathsf{Ep1}(e_2, \rho, \kappa))$	$\diamond\ \ \langle\ \llbracket (e_1,\ e_2) \rrbracket\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ \llbracket e_1 \rrbracket\rho \mid \mathsf{Ep1}(e_2, \rho, \kappa)\ \rangle$
$\mathcal{E}\llbracket () \rrbracket \rho\kappa = \mathcal{A}\ \kappa\ unit()$	$\langle\ \llbracket () \rrbracket\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ unit() \mid \kappa\ \rangle$
$\mathcal{E}\llbracket e \uparrow f \rrbracket \rho\kappa = \mathcal{E}\llbracket e \rrbracket\ \rho\,(\mathsf{Epp}(f, \rho, \kappa))$	$\diamond\ \ \langle\ \llbracket e \uparrow f \rrbracket\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ \llbracket e \rrbracket\rho \mid \mathsf{Epp}(f, \rho, \kappa)\ \rangle$
$\mathcal{E}\llbracket \lceil f \rceil \rrbracket \rho\kappa = \mathcal{A}\ \kappa\ \mathsf{Closr}(f, \rho)$	$\langle\ \llbracket \lceil f \rceil \rrbracket\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ \mathsf{Closr}(f, \rho) \mid \kappa\ \rangle$
$\mathcal{C} : c \to Env \to Val \to Ans$	The rules coming from $\mathcal{C}$
$\mathcal{C}\llbracket y \rrbracket \rho v = let\ cnt(\kappa) = \rho\ y\ in\ \mathcal{A}\ \kappa\ v$	$\langle\ v \mid \llbracket y \rrbracket\rho\ \rangle \rightsquigarrow let\ cnt(\kappa) = \rho\ y\ in\ \langle\ v \mid \kappa\ \rangle$
$\mathcal{C}\llbracket \{c_1,\ c_2\} \rrbracket \rho\ in_1(v) = \mathcal{C}\llbracket c_1 \rrbracket \rho v$	$*\ \ \langle\ in_1(v) \mid \llbracket \{c_1,\ c_2\} \rrbracket\rho\ \rangle \rightsquigarrow \langle\ v \mid \llbracket c_1 \rrbracket\rho\ \rangle$
$\mathcal{C}\llbracket \{c_1,\ c_2\} \rrbracket \rho\ in_2(v) = \mathcal{C}\llbracket c_2 \rrbracket \rho v$	$*\ \ \langle\ in_2(v) \mid \llbracket \{c_1,\ c_2\} \rrbracket\rho\ \rangle \rightsquigarrow \langle\ v \mid \llbracket c_2 \rrbracket\rho\ \rangle$
$\mathcal{C}\llbracket \{\} \rrbracket \rho v = case\ v\ of\ esac$	$\langle\ v \mid \llbracket \{\} \rrbracket\rho\ \rangle \rightsquigarrow case\ v\ of\ esac$
$\mathcal{C}\llbracket f \downarrow c \rrbracket \rho v = \mathcal{F}\llbracket f \rrbracket\ \rho\, v\,(\mathsf{Cpp}(c, \rho))$	$*\ \ \langle\ v \mid \llbracket f \downarrow c \rrbracket\rho\ \rangle \rightsquigarrow \langle\ v \mid \llbracket f \rrbracket\rho \mid \mathsf{Cpp}(c, \rho)\ \rangle$
$\mathcal{C}\llbracket \lfloor f \rfloor \rrbracket \rho\ contx(v, \kappa) = \mathcal{F}\llbracket f \rrbracket \rho v\kappa$	$*\ \ \langle\ contx(v, \kappa) \mid \llbracket \lfloor f \rfloor \rrbracket\rho\ \rangle \rightsquigarrow \langle\ v \mid \llbracket f \rrbracket\rho \mid \kappa\ \rangle$
$\mathcal{F} : f \to Env \to Val \to Cnt \to Ans$	The rules coming from $\mathcal{F}$
$\mathcal{F}\llbracket X \Rightarrow e \rrbracket \rho v\kappa = \mathcal{E}\llbracket e \rrbracket\,([\mathcal{X}\llbracket X \rrbracket \mapsto v]\rho)\,\kappa$	$*\ \ \langle\ v \mid \llbracket X \Rightarrow e \rrbracket\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ \llbracket e \rrbracket\,([\mathcal{X}\llbracket X \rrbracket \mapsto v]\rho) \mid \kappa\ \rangle$
$\mathcal{F}\llbracket c \Leftarrow Y \rrbracket \rho v\kappa = \mathcal{C}\llbracket c \rrbracket\,([\mathcal{Y}\llbracket Y \rrbracket \mapsto \kappa]\rho)\, v$	$*\ \ \langle\ v \mid \llbracket c \Leftarrow Y \rrbracket\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ v \mid \llbracket c \rrbracket\,([\mathcal{Y}\llbracket Y \rrbracket \mapsto \kappa]\rho)\ \rangle$
$\mathcal{F}\llbracket \overline{e} \rrbracket \rho v\kappa = \mathcal{E}\llbracket e \rrbracket\ \rho\,(\mathsf{Openr}(v, \kappa))$	$*\ \ \langle\ v \mid \llbracket \overline{e} \rrbracket\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ \llbracket e \rrbracket\rho \mid \mathsf{Openr}(v, \kappa)\ \rangle$
$\mathcal{F}\llbracket \underline{c} \rrbracket \rho v\kappa = \mathcal{C}\llbracket c \rrbracket\ \rho\ contx(v, \kappa)$	$*\ \ \langle\ v \mid \llbracket \underline{c} \rrbracket\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ contx(v, \kappa) \mid \llbracket c \rrbracket\rho\ \rangle$
$\mathcal{A} : Cnt \to Val \to Ans$	The rules coming from $\mathcal{A}$
$\mathcal{A}\ (\mathsf{Ep1}(e_2, \rho, \kappa))\ v_1 = \mathcal{E}\llbracket e_2 \rrbracket\ \rho\,(\mathsf{Ep2}(v_1, \kappa))$	$\diamond\ \ \langle\ v_1 \mid \mathsf{Ep1}(e_2, \rho, \kappa)\ \rangle \rightsquigarrow \langle\ \llbracket e_2 \rrbracket\rho \mid \mathsf{Ep2}(v_1, \kappa)\ \rangle$
$\mathcal{A}\ (\mathsf{Ep2}(v_1, \kappa))\ v_2 = \mathcal{A}\ \kappa\ pair(v_1, v_2)$	$\diamond\ \ \langle\ v_2 \mid \mathsf{Ep2}(v_1, \kappa)\ \rangle \rightsquigarrow \langle\ pair(v_1, v_2) \mid \kappa\ \rangle$
$\mathcal{A}\ (\mathsf{Epp}(f, \rho, \kappa))\ v = \mathcal{F}\llbracket f \rrbracket \rho v\kappa$	$*\ \ \langle\ v \mid \mathsf{Epp}(f, \rho, \kappa)\ \rangle \rightsquigarrow \langle\ v \mid \llbracket f \rrbracket\rho \mid \kappa\ \rangle$
$\mathcal{A}\ (\mathsf{Cpp}(c, \rho))\ v = \mathcal{C}\llbracket c \rrbracket \rho v$	$\langle\ v \mid \mathsf{Cpp}(c, \rho)\ \rangle \rightsquigarrow \langle\ v \mid \llbracket c \rrbracket\rho\ \rangle$
$\mathcal{A}\ (\mathsf{Openr}(v, \kappa))\ \mathsf{Closr}(f, \rho) = \mathcal{F}\llbracket f \rrbracket \rho v\kappa$	$\diamond\ \ \langle\ \mathsf{Closr}(f, \rho) \mid \mathsf{Openr}(v, \kappa)\ \rangle \rightsquigarrow \langle\ v \mid \llbracket f \rrbracket\rho \mid \kappa\ \rangle$
$\mathcal{A}\ (\mathsf{Inl}(\kappa))\ v = \mathcal{A}\ \kappa\ in_1(v)$	$*\ \ \langle\ v \mid \mathsf{Inl}(\kappa)\ \rangle \rightsquigarrow \langle\ in_1(v) \mid \kappa\ \rangle$
$\mathcal{A}\ (\mathsf{Inr}(\kappa))\ v = \mathcal{A}\ \kappa\ in_2(v)$	$*\ \ \langle\ v \mid \mathsf{Inr}(\kappa)\ \rangle \rightsquigarrow \langle\ in_2(v) \mid \kappa\ \rangle$
$\mathcal{A}\ \mathsf{Init}\ n = n$	$*\ \ \langle\ n \mid \mathsf{Init}\ \rangle \rightsquigarrow n$

$$
\begin{array}{rcl}
\mathcal{X} : X \to Val \to Env & \to & Env \\
[\mathcal{X}\llbracket x \rrbracket \mapsto v]\rho & = & \rho[x \mapsto val(v)] \\
[\mathcal{X}\llbracket () \rrbracket \mapsto unit()]\rho & = & \rho \\
[\mathcal{X}\llbracket (X_1,\ X_2) \rrbracket \mapsto pair(v_1, v_2)]\rho & = & [\mathcal{X}\llbracket X_1 \rrbracket \mapsto v_1]([\mathcal{X}\llbracket X_2 \rrbracket \mapsto v_2]\rho)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{Y} : Y \to Cnt \to Env & \to & Env \\
[\mathcal{Y}\llbracket y \rrbracket \mapsto \kappa]\rho & = & \rho[y \mapsto cnt(\kappa)] \\
[\mathcal{Y}\llbracket \{\} \rrbracket \mapsto \kappa]\rho & = & \rho \\
[\mathcal{Y}\llbracket \{Y_1,\ Y_2\} \rrbracket \mapsto \kappa]\rho & = & [\mathcal{Y}\llbracket Y_1 \rrbracket \mapsto \mathsf{Inl}(\kappa)]([\mathcal{Y}\llbracket Y_2 \rrbracket \mapsto \mathsf{Inr}(\kappa)]\rho)
\end{array}
$$

Figure 17: Defunctionalized Semantics and Corresponding Abstract Machine (CBV)

n	$\mapsto$	n	Init	$\mapsto$	$\bullet$
$unit()$	$\mapsto$	$()$	$\mathsf{Ep1}(e_2, \kappa)$	$\mapsto$	$(x_1 \Rightarrow (x_1,\, e_2)) \downarrow \kappa$
$pair(v_1, v_2)$	$\mapsto$	$(v_1,\, v_2)$	$\mathsf{Ep2}(v_1, \kappa)$	$\mapsto$	$(x_2 \Rightarrow (v_1,\, x_2)) \downarrow \kappa$
$in_1(v)$	$\mapsto$	$[v \uparrow inl]$	$\mathsf{Epp}(f, \kappa)$	$\mapsto$	$f \downarrow \kappa$
$in_2(v)$	$\mapsto$	$[v \uparrow inr]$	$\mathsf{Cpp}(c)$	$\mapsto$	c
$\mathsf{Closr}(f)$	$\mapsto$	$\lceil f \rceil$	$\mathsf{Openr}(v, \kappa)$	$\mapsto$	$(\lceil g \rceil \Rightarrow \kappa \uparrow g) \downarrow v$
$contx(v, \kappa)$	$\mapsto$	$[\kappa \uparrow (g \downarrow v \Leftarrow \lfloor g \rfloor)]$	$\mathsf{Inl}(\kappa)$	$\mapsto$	$[inl \downarrow \kappa]$
			$\mathsf{Inl}(\kappa)$	$\mapsto$	$[inl \downarrow \kappa]$

Figure 18: Term Transformation for CBV Derivation

The translation is mechanical: whenever λ is used in Figure 16, it is replaced with a first-order data and the corresponding clause is added to the apply function $\mathcal{A}$. For example, $\lambda v.\mathcal{F}[\![f]\!]\rho v\kappa$ appearing in $\mathcal{E}[\![e \uparrow f]\!]\rho\kappa$ of Figure 16 is translated to $\mathsf{Epp}(f, \rho, \kappa)$ where the free variables of $\lambda v.\mathcal{F}[\![f]\!]\rho v\kappa$ becomes the arguments to Epp. The corresponding clause is added to the definition of $\mathcal{A}$:

$$\mathcal{A}\ (\mathsf{Epp}(f, \rho, \kappa))\ v = \mathcal{F}[\![f]\!]\rho v\kappa$$

so that the original body of $\lambda v.\mathcal{F}[\![f]\!]\rho v\kappa$ is executed. Since all the κ's now become a first-order data, its application is replaced with a call to the apply function $\mathcal{A}$. For example, $\kappa\ n$ in $\mathcal{E}[\![n]\!]\rho\kappa$ in Figure 16 is translated to $\mathcal{A}\ \kappa\ n$.

5.2 Rewriting to Abstract Machine Style

Because the four semantic functions, $\mathcal{E}$, $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{A}$, in the defunctionalized interpreter are mutually tail-recursive, the resulting interpreter can be directly regarded as an abstract machine. The right column of Figure 17 shows the corresponding abstract machine. It is obtained by mechanically changing the notation from $\mathcal{E}[\![e]\!]\rho\kappa$, $\mathcal{C}[\![c]\!]\rho v$, $\mathcal{F}[\![f]\!]\rho v\kappa$, and $\mathcal{A}\ \kappa\ v$, to $\langle\ [\![e]\!]\rho \mid \kappa\ \rangle$, $\langle\ v \mid [\![c]\!]\rho\ \rangle$, $\langle\ v \mid [\![f]\!]\rho \mid \kappa\ \rangle$, and $\langle\ v \mid \kappa\ \rangle$, respectively.

From this abstract machine, we can derive our small-step reduction semantics by two more simple transformations: replacing the environment with substitution and rewriting defunctionalized first-order data in SLC syntax.

The abstract machine in Figure 17 uses an environment to realize substitution. The role of an environment is to defer the substitution until the substituted variable is found. (As a slogan, "an environment is a lazy substitution.") We can simply remove all the environments (and semantic brackets) by performing substitution whenever the environment is extended, namely, at the first two rules for $\mathcal{F}$ in Figure 17. With this transformation, the first two rules for $\mathcal{F}$ becomes identical to the corresponding rules ((βR_{v}) and $(\overline{\beta L_{\mathrm{v}}})$) in Figure 10. After removing the environments, all the rules for variables become useless, because they will never be used. Furthermore, the rule for a number becomes useless, because it now becomes an identity transition: $\langle\ n \mid \kappa\ \rangle \rightsquigarrow \langle\ n \mid \kappa\ \rangle$.

The second transformation is to rewrite defunctionalized first-order data in SLC syntax. Rather than writing $\mathsf{Epp}(f, \kappa)$, for example, we can instead write $f \downarrow \kappa$, because the rule for $\mathsf{Epp}(f, \kappa)$ is:

$$\langle\ v \mid \mathsf{Epp}(f, \kappa)\ \rangle \rightsquigarrow \langle\ v \mid f \mid \kappa\ \rangle$$

By writing $\mathsf{Epp}(f, \kappa)$ as $f \downarrow \kappa$, the rule becomes identical to $(\overline{pop_{\mathrm{v}}})$. Likewise, we change the notation for all the first-order data as shown in Figure 18.

With the above two transformations, all the rules marked with $*$ in Figure 17 coincide with the marked rules in the CBV small-step reduction semantics in Figure 10. The rules with $\diamond$ do not coincide with the rules in Figure 10, but we can confirm that they can all be simulated by one or more reduction steps of Figure 10. Furthermore, since $\mathcal{X}$ and $\mathcal{Y}$ are transformed to $\mathcal{R}$ and $\overline{\mathcal{L}}_{\mathrm{v}}$ (except for function patterns), we conclude that the CBV small-step semantics in Figure 10 correctly implements Filinski's SLC.

$$
\begin{array}{rcl}
Val & = & Cnt \to Ans \\
Cnt & = & Bcont + Zero + Case(Cnt \times Cnt) + Pr_1(Cnt) + Pr_2(Cnt) + \\
 & & Contx(Val \to Cnt \to Ans) + Closr(Val \times Cnt) \\
Env & = & Ide \to Val + Cnt
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E} : e \to Env \to Cnt & \to & Ans \\
\mathcal{E}[\![n]\!]\rho k & = & let\ bcont = k\ in\ n \\
\mathcal{E}[\![x]\!]\rho k & = & let\ val(\nu) = \rho\ x\ in\ \nu\ k \\
\mathcal{E}[\![(e_1,\, e_2)]\!]\rho k & = & case\ k\ of\ pr_1(k') : \mathcal{E}[\![e_1]\!]\rho k' \mid pr_2(k') : \mathcal{E}[\![e_2]\!]\rho k' \\
\mathcal{E}[\![()]\!]\rho k & = & case\ k\ of\ esac \\
\mathcal{E}[\![e \uparrow f]\!]\rho k & = & \mathcal{F}[\![f]\!]\,\rho\,(\lambda k'.\mathcal{E}[\![e]\!]\rho k')\,k \\
\mathcal{E}[\![\lceil f \rceil]\!]\rho k & = & let\ closr(\nu, k') = k\ in\ \mathcal{F}[\![f]\!]\rho\nu k'
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C} : c \to Env \to Val & \to & Ans \\
\mathcal{C}[\![y]\!]\rho\nu & = & let\ cnt(k) = \rho\ y\ in\ \nu\ k \\
\mathcal{C}[\![\{c_1,\, c_2\}]\!]\rho\nu & = & \mathcal{C}[\![c_2]\!]\,\rho\,(\lambda k_2.\mathcal{C}[\![c_1]\!]\,\rho\,(\lambda k_1.\nu\ case(k_1, k_2))) \\
\mathcal{C}[\![\{\}]\!]\rho\nu & = & \nu\ zero() \\
\mathcal{C}[\![f \downarrow c]\!]\rho\nu & = & \mathcal{C}[\![c]\!]\,\rho\,(\lambda k.\mathcal{F}[\![f]\!]\rho\nu k) \\
\mathcal{C}[\![\lfloor f \rfloor]\!]\rho\nu & = & \nu\ contx(\lambda\nu' k.\mathcal{F}[\![f]\!]\rho\nu' k)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F} : f \to Env \to Val \to Cnt & \to & Ans \\
\mathcal{F}[\![p \Rightarrow e]\!]\rho\nu k & = & \mathcal{E}[\![e]\!]\,([\mathcal{X}[\![p]\!] \mapsto \nu]\rho)\,k \\
\mathcal{F}[\![c \Leftarrow Y]\!]\rho\nu k & = & \mathcal{C}[\![c]\!]\,([\mathcal{Y}[\![Y]\!] \mapsto k]\rho)\,\nu \\
\mathcal{F}[\![\overline{e}]\!]\rho\nu k & = & \mathcal{E}[\![e]\!]\,\rho\ closr(\nu, k) \\
\mathcal{F}[\![\underline{c}]\!]\rho\nu k & = & \mathcal{C}[\![c]\!]\,\rho\,(\lambda k'.let\ contx(h) = t\ in\ h\,\nu\,k')
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{X} : p \to Val \to Env & \to & Env \\
[\mathcal{X}[\![x]\!] \mapsto \nu]\rho & = & \rho[x \mapsto val(\nu)] \\
[\mathcal{X}[\![()]\!] \mapsto \nu]\rho & = & \rho \\
[\mathcal{X}[\![(p_1,\, p_2)]\!] \mapsto \nu]\rho & = & [\mathcal{X}[\![p_1]\!] \mapsto \lambda k.\nu\ pr_1(k)]\ ([\mathcal{X}[\![p_2]\!] \mapsto \lambda k.\nu\ pr_2(k)]\rho)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{Y} : q \to Cnt \to Env & \to & Env \\
[\mathcal{Y}[\![y]\!] \mapsto k]\rho & = & \rho[y \mapsto cnt(k)] \\
[\mathcal{Y}[\![\{\}]\!] \mapsto k]\rho & = & let\ zero() = k\ in\ \rho \\
[\mathcal{Y}[\![\{q_1,\, q_2\}]\!] \mapsto k]\rho & = & let\ case(c_1, c_2) = k\ in\ [\mathcal{Y}[\![q_1]\!] \mapsto c_1]\ ([\mathcal{Y}[\![q_2]\!] \mapsto c_2]\rho)
\end{array}
$$

Figure 19: Filinski's Denotational Semantics for CBN SLC

6 Functional Correspondence for CBN SLC

Filinski also presented denotational semantics for CBN SLC. Using the same method presented in the previous section, we can obtain the CBN small-step semantics. Although denotational semantics for CBV SLC and CBN SLC look quite different (because they are encoded in a standard non-symmetric lambda-calculus), the present results confirm that they are actually dual to each other. In this section, we show that the CBN small-step semantics of SLC presented in Section 4 corresponds to Filinski's original definition of CBN SLC given as a denotational semantics.

Figure 19 shows denotational semantics of Filinski's CBN SLC. In the semantics, k represents *covalues* (the result of evaluation), and ν represents *semantic* values (that is *not* a result). Filinski introduces a domain for each covalue type (e.g., *Case*) together with a constructor for it (in lower case letter, e.g., *case*).

Defunctionalized Semantics	Corresponding Abstract Machine
$\mathcal{E} : e \to Env \to Cnt \to Ans$	The rules coming from $\mathcal{E}$
$\mathcal{E}[\![n]\!]\rho\ bcont = n$	$*\ \ \langle\ [\![n]\!]\rho \mid bcont\ \rangle \rightsquigarrow n$
$\mathcal{E}[\![x]\!]\rho k = let\ val(\nu) = \rho\ x\ in\ \mathcal{A}\ \nu\ k$	$\langle\ [\![x]\!]\rho \mid k\ \rangle \rightsquigarrow let\ val(\nu) = \rho\ x\ in\ \langle\ \nu \mid k\ \rangle$
$\mathcal{E}[\![(e_1,\, e_2)]\!]\rho\ pr_1(k) = \mathcal{E}[\![e_1]\!]\rho k$	$*\ \ \langle\ [\![(e_1,\, e_2)]\!]\rho \mid pr_1(k)\ \rangle \rightsquigarrow \langle\ [\![e_1]\!]\rho \mid k\ \rangle$
$\mathcal{E}[\![(e_1,\, e_2)]\!]\rho\ pr_2(k) = \mathcal{E}[\![e_2]\!]\rho k$	$*\ \ \langle\ [\![(e_1,\, e_2)]\!]\rho \mid pr_2(k)\ \rangle \rightsquigarrow \langle\ [\![e_2]\!]\rho \mid k\ \rangle$
$\mathcal{E}[\![()]\!]\rho k = case\ k\ of\ esac$	$\langle\ [\![()]\!]\rho \mid k\ \rangle \rightsquigarrow case\ k\ of\ esac$
$\mathcal{E}[\![e \uparrow f]\!]\rho k = \mathcal{F}[\![f]\!]\,\rho\,(\mathsf{Epp}(e,\rho))\,k$	$*\ \ \langle\ [\![e \uparrow f]\!]\rho \mid k\ \rangle \rightsquigarrow \langle\ \mathsf{Epp}(e,\rho) \mid [\![f]\!]\rho \mid k\ \rangle$
$\mathcal{E}[\![\lceil f \rceil]\!]\rho\ closr(a,c) = \mathcal{F}[\![f]\!]\rho ac$	$*\ \ \langle\ [\![\lceil f \rceil]\!]\rho \mid closr(\nu,k)\ \rangle \rightsquigarrow \langle\ \nu \mid [\![f]\!]\rho \mid k\ \rangle$
$\mathcal{C} : c \to Env \to Val \to Ans$	The rules coming from $\mathcal{C}$
$\mathcal{C}[\![y]\!]\rho\nu = let\ cnt(k) = \rho\ y\ in\ \mathcal{A}\ \nu\ k$	$\langle\ \nu \mid [\![y]\!]\rho\ \rangle \rightsquigarrow let\ cnt(k) = \rho\ y\ in\ \langle\ \nu \mid k\ \rangle$
$\mathcal{C}[\![\{c_1,\, c_2\}]\!]\rho\nu = \mathcal{C}[\![c_2]\!]\,\rho\,(\mathsf{Cp2}(c_1,\rho,\nu))$	$\diamond\ \ \langle\ \nu \mid [\![\{c_1,\, c_2\}]\!]\rho\ \rangle \rightsquigarrow \langle\ \mathsf{Cp2}(c_1,\rho,\nu) \mid [\![c_2]\!]\rho\ \rangle$
$\mathcal{C}[\![\{\}]\!]\rho\nu = \mathcal{A}\ \nu\ zero()$	$\langle\ \nu \mid [\![\{\}]\!]\rho\ \rangle \rightsquigarrow \langle\ \nu \mid zero()\ \rangle$
$\mathcal{C}[\![f \downarrow c]\!]\rho\nu = \mathcal{C}[\![c]\!]\,\rho\,(\mathsf{Cpp}(f,\rho,\nu))$	$\diamond\ \ \langle\ \nu \mid [\![f \downarrow c]\!]\rho\ \rangle \rightsquigarrow \langle\ \mathsf{Cpp}(f,\rho,\nu) \mid [\![c]\!]\rho\ \rangle$
$\mathcal{C}[\![\lfloor f \rfloor]\!]\rho\nu = \mathcal{A}\ \nu\ \mathsf{Contx}(f,\rho)$	$\langle\ \nu \mid [\![\lfloor f \rfloor]\!]\rho\ \rangle \rightsquigarrow \langle\ \nu \mid \mathsf{Contx}(f,\rho)\ \rangle$
$\mathcal{F} : f \to Env \to Val \to Cnt \to Ans$	The rules coming from $\mathcal{F}$
$\mathcal{F}[\![p \Rightarrow e]\!]\rho\nu k = \mathcal{E}[\![e]\!]\,([\mathcal{X}[\![p]\!] \mapsto \nu]\rho)\,k$	$*\ \ \langle\ \nu \mid [\![p \Rightarrow e]\!]\rho \mid k\ \rangle \rightsquigarrow \langle\ [\![e]\!]\,([\mathcal{X}[\![p]\!] \mapsto \nu]\rho) \mid k\ \rangle$
$\mathcal{F}[\![c \Leftarrow q]\!]\rho\nu k = \mathcal{C}[\![c]\!]\,([\mathcal{Y}[\![q]\!] \mapsto k]\rho)\,\nu$	$*\ \ \langle\ \nu \mid [\![c \Leftarrow q]\!]\rho \mid k\ \rangle \rightsquigarrow \langle\ \nu \mid [\![c]\!]\,([\mathcal{Y}[\![q]\!] \mapsto k]\rho)\ \rangle$
$\mathcal{F}[\![\overline{e}]\!]\rho\nu k = \mathcal{E}[\![e]\!]\,\rho\,closr(\nu,k)$	$*\ \ \langle\ \nu \mid [\![\overline{e}]\!]\rho \mid k\ \rangle \rightsquigarrow \langle\ [\![e]\!]\rho \mid contx(\nu,k)\ \rangle$
$\mathcal{F}[\![\underline{c}]\!]\rho\nu k = \mathcal{C}[\![c]\!]\,\rho\,(\mathsf{Plug}(\nu,k))$	$*\ \ \langle\ \nu \mid [\![\underline{c}]\!]\rho \mid k\ \rangle \rightsquigarrow \langle\ \mathsf{Plug}(\nu,k) \mid [\![c]\!]\rho\ \rangle$
$\mathcal{A} : Val \to Cnt \to Ans$	The rules coming from $\mathcal{A}$
$\mathcal{A}\ (\mathsf{Epp}(e,\rho))\ k = \mathcal{E}[\![e]\!]\rho k$	$\langle\ \mathsf{Epp}(e,\rho) \mid k\ \rangle \rightsquigarrow \langle\ [\![e]\!]\rho \mid k\ \rangle$
$\mathcal{A}\ (\mathsf{Cp2}(c_1,\rho,\nu))\ k_2 = \mathcal{C}[\![c_1]\!]\,\rho\,(\mathsf{Cp1}(k_2,\nu))$	$\diamond\ \ \langle\ \mathsf{Cp1}(c_2,\rho,\nu) \mid k_1\ \rangle \rightsquigarrow \langle\ \mathsf{Cp2}(k_1,\nu) \mid [\![c_2]\!]\rho\ \rangle$
$\mathcal{A}\ (\mathsf{Cp1}(k_2,\nu))\ k_1 = \mathcal{A}\ \nu\ case(k_1,k_2)$	$\diamond\ \ \langle\ \mathsf{Cp2}(k_1,\nu) \mid k_2\ \rangle \rightsquigarrow \langle\ \nu \mid case(k_1,k_2)\ \rangle$
$\mathcal{A}\ (\mathsf{Cpp}(f,\rho,\nu))\ k = \mathcal{F}[\![f]\!]\rho\nu k$	$*\ \ \langle\ \mathsf{Cpp}(f,\rho,\nu) \mid k\ \rangle \rightsquigarrow \langle\ \nu \mid [\![f]\!]\rho \mid k\ \rangle$
$\mathcal{A}\ (\mathsf{Plug}(\nu,k))\ \mathsf{Contx}(f,\rho) = \mathcal{F}[\![f]\!]\rho k\nu$	$\diamond\ \ \langle\ \mathsf{Plug}(\nu,k) \mid \mathsf{Contx}(f,\rho)\ \rangle \rightsquigarrow \langle\ \nu \mid [\![f]\!]\rho \mid k\ \rangle$
$\mathcal{A}\ (\mathsf{Fst}(\nu))\ k = \mathcal{A}\ \nu\ pr_1(k)$	$*\ \ \langle\ \mathsf{Fst}(\nu) \mid k\ \rangle \rightsquigarrow \langle\ \nu \mid pr_1(k)\ \rangle$
$\mathcal{A}\ (\mathsf{Snd}(\nu))\ k = \mathcal{A}\ \nu\ pr_2(k)$	$*\ \ \langle\ \mathsf{Snd}(\nu) \mid k\ \rangle \rightsquigarrow \langle\ \nu \mid pr_2(k)\ \rangle$

$$\begin{array}{rcl}
\mathcal{X} : p \to Val \to Env & \to & Env \\
{[\mathcal{X}[\![x]\!] \mapsto \nu]\rho} & = & \rho[x \mapsto val(\nu)] \\
{[\mathcal{X}[\![()]\!] \mapsto \nu]\rho} & = & \rho \\
{[\mathcal{X}[\![(p_1,\, p_2)]\!] \mapsto \nu]\rho} & = & [\mathcal{X}[\![p_1]\!] \mapsto \mathsf{Fst}(\nu)]([\mathcal{X}[\![p_2]\!] \mapsto \mathsf{Snd}(\nu)]\rho)
\end{array}$$

$$\begin{array}{rcl}
\mathcal{Y} : q \to Cnt \to Env & \to & Env \\
{[\mathcal{Y}[\![y]\!] \mapsto k]\rho} & = & \rho[y \mapsto cnt(\nu)] \\
{[\mathcal{Y}[\![\{\}]\!] \mapsto \{\}]\rho} & = & \rho \\
{[\mathcal{Y}[\![\{q_1,\, q_2\}]\!] \mapsto case(k_1,k_2)]\rho} & = & [\mathcal{Y}[\![q_1]\!] \mapsto k_1]\ ([\mathcal{Y}[\![q_2]\!] \mapsto k_2]\rho)
\end{array}$$

Figure 20: Defunctionalized Semantics and Corresponding Abstract Machine (CBN)

$$\begin{array}{rcl@{\qquad}rcl}
n & \mapsto & n & \mathit{bcont} & \mapsto & \bullet \\
\mathsf{Epp}(e) & \mapsto & e & \mathit{zero}() & \mapsto & \{\} \\
\mathsf{Cp1}(k_2, \nu) & \mapsto & \nu \uparrow (\{y_1,\, k_2\} \Leftarrow y_1) & \mathit{pair}(k_1, k_2) & \mapsto & \{k_1,\, k_2\} \\
\mathsf{Cp2}(c_1, \nu) & \mapsto & \nu \uparrow (\{c_1,\, y_2\} \Leftarrow y_2) & \mathit{pr}_1(k) & \mapsto & [\mathit{fst} \downarrow k] \\
\mathsf{Cpp}(f, \nu) & \mapsto & \nu \uparrow f & \mathit{pr}_2(k) & \mapsto & [\mathit{snd} \downarrow k] \\
\mathsf{Plug}(\nu, k) & \mapsto & \nu \uparrow (g \downarrow k \Leftarrow \lfloor g \rfloor) & \mathsf{Contx}(f) & \mapsto & \lfloor f \rfloor \\
\mathsf{Fst}(\nu) & \mapsto & [\nu \uparrow \mathit{fst}] & \mathit{closr}(\nu, k) & \mapsto & [(\lceil g \rceil \Rightarrow k \uparrow g) \downarrow \nu] \\
\mathsf{Snd}(\nu) & \mapsto & [\nu \uparrow \mathit{snd}] & & &
\end{array}$$

Figure 21: Term Transformations for CBN Derivation

6.1 Defunctionalization

We defunctionalize two kinds of higher-order functions in Figure 19: values (ν of type Val) and contexts (the argument of *contx*). The result of defunctionalization is found in the upper left of Figure 20. The translation is mechanical and dual to the CBV derivation in Section 5.1. For example, though we have defunctionalized a semantic continuation for $e \uparrow f$ in Section 5.1, here, we need to defunctionalize a semantic value for $f \downarrow c$. We translate the semantic value $\lambda k.\mathcal{F}[\![F]\!]\rho\nu k$ to $\mathsf{Cpp}(f, \rho, \nu)$. The corresponding clause is added to the definition of $\mathcal{A}$:

$$\mathcal{A}\ (\mathsf{Cpp}(f, \rho, \nu))\ k = \mathcal{F}[\![f]\!]\rho\nu k$$

6.2 Rewriting to Abstract Machine Style

Because the four semantic functions, $\mathcal{E}$, $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{A}$, in the defunctionalized interpreter are mutually tail-recursive, the resulting interpreter can be directly regarded as an abstract machine. The right column of Figure 20 shows the corresponding abstract machine. It is obtained by mechanically changing the notation from $\mathcal{E}[\![e]\!]\rho k$, $\mathcal{C}[\![c]\!]\rho\nu$, $\mathcal{F}[\![f]\!]\rho\nu k$, and $\mathcal{A}\ \nu\ k$, to $\langle\ [\![e]\!]\rho \mid k\ \rangle$, $\langle\ \nu \mid [\![c]\!]\rho\ \rangle$, $\langle\ \nu \mid [\![f]\!]\rho \mid k\ \rangle$, and $\langle\ \nu \mid k\ \rangle$, respectively.

From this abstract machine, we can derive our small-step reduction semantics by two more simple transformations: replacing the environment with substitution and rewriting defunctionalized first-order data in SLC syntax. First, we simply remove all the environments (and semantic brackets) by performing substitution whenever the environment is extended. After removing the environments, all the rules for variables become useless, because they will never be used. Secondly, we change the notation for all the first-order data as shown in Figure 21.

With the above two transformations, all the rules marked with $*$ in Figure 20 coincide with the marked rules in the CBN small-step reduction semantics in Figure 14. The rules with $\diamond$ do not coincide with the rules in Figure 14, but we can confirm that they can all be simulated by one or more reduction steps of Figure 14. Furthermore, since $\mathcal{X}$ and $\mathcal{Y}$ are transformed to $\mathcal{L}_\mathrm{n}$ and $\overline{\mathcal{R}}$ (except for function patterns), we conclude that the CBN small-step semantics in Figure 14 correctly implements Filinski's SLC.

7 Future Direction

In this paper, we have presented a small-step reduction semantics for Filinski's symmetric lambda calculus (SLC). Now that we obtained a small-step reduction semantics for SLC, we can express and compare various other calculi in terms of SLC. Our preliminary work shows that we can naturally encode Felleisen's $\mathcal{C}$ operator as well as $\lambda\mu$-calculus into SLC [11]. We are also interested in if delimited continuations can be introduced into the SLC framework.

References

[1] Ager, M. S., D. Biernacki, O. Danvy, and J. Midtgaard "A functional correspondence between evaluators and abstract machines," *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pp. 8–19 (September 2003).

[2] Barbanera, F., and S. Berardi "A Symmetric Lambda Calculus for Classical Program Extraction," *Information and Computation*, Vol. 125, No. 2, pp. 103–117, Academic Press (March 1996).

[3] Curien, P.-L., and H. Herbelin "The Duality of Computation," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pp. 233–243 (September 2000).

[4] Danvy, O. "Defunctionalized interpreters for programming languages," *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pp. 131–142 (September 2008).

[5] Felleisen, M., and R. Hieb "The Revised Report on the Syntactic Theories of Sequential Control and State," *Theoretical Computer Science*, Vol. 103, No. 2, pp. 235–271 (September 1992).

[6] Filinski, A. "Declarative Continuations and Categorical Duality," Master's thesis, DIKU Report 89/11, University of Copenhagen (August 1989).

[7] Griffin, T. "A Formulae-as-Types Notion of Control," *Conference Record of the 17th ACM Symposium on Principles of Programming Languages*, pp. 47–58 (January 1990).

[8] Parigot, M. "$\lambda\mu$-calculus: An Algorithmic Interpretation of Classical Natural Deduction," In A. Voronkov, editor, *Logic Programming and Automated Reasoning (LNCS 624)*, pp. 190–201 (July 1992).

[9] Pierce, B. C. *Types and Programming Languages*, Cambridge: MIT Press (2002).

[10] Reynolds, J. C. "Definitional Interpreters for Higher-Order Programming Languages," *Proceedings of the ACM National Conference*, Vol. 2, pp. 717–740, (August 1972), reprinted in *Higher-Order and Symbolic Computation*, Vol. 11, No. 4, pp. 363–397, Kluwer Academic Publishers (December 1998).

[11] Sakaue, S., and K. Asai "The Foundations of Symmetric Lambda Calculus," *Computer Software*, Vol. 26, No. 2, pp. 3-17 (May 2009). (in Japanese).

[12] Selinger, P. "Control Categories and Duality: on the Categorical Semantics of the Lambda-Mu Calculus," *Mathematical Structures in Computer Science*, Vol. 11, No. 2, pp. 207-260 (March 2001).

[13] Tzevelekos, N. "Investigations on the Dual Calculus," *Theoretical Computer Science*, Vol. 360, Nos. 1–3, pp. 289–326 (August 2006).

[14] Wadler, P. "Call-by-value is dual to call-by-name," *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pp. 189–201 (August 2003).