Basic Research in Computer Science

BRICS RS-00-33 Danvy & Nielsen: A Higher-Order Colon Translation

A Higher-Order Colon Translation

Olivier Danvy
Lasse R. Nielsen

BRICS Report Series **RS-00-33**

ISSN 0909-0878 **December 2000**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/00/33/`

A Higher-Order Colon Translation *

Olivier Danvy and Lasse R. Nielsen

BRICS †
Department of Computer Science
University of Aarhus ‡

December 2000

Abstract

A lambda-encoding such as the CPS transformation gives rise to administrative redexes. In his seminal article "Call-by-name, call-by-value and the lambda-calculus", 25 years ago, Plotkin tackled administrative reductions using a so-called colon translation. In "Representing control, a study of the CPS transformation", 15 years later, Danvy and Filinski integrated administrative reductions in the CPS transformation, making it operate in one pass. This one-pass transformation is higher-order, and can be used for other lambda-encodings, but we do not see its associated proof technique used in practice—instead, Plotkin's colon translation appears to be favored. Therefore, in an attempt to link the higher-order transformation and Plotkin's proof technique, we recast Plotkin's proof of Indifference and Simulation in a higher-order setting. To this end, we extend the colon translation from first order to higher order.

Keywords: Call by name, call by value, λ-calculus, continuation-passing style (CPS), CPS transformation, administrative reductions, colon translation, one-pass CPS transformation, Indifference, Simulation.

*To appear in the proceedings of FLOPS 2001.
†Basic Research in Computer Science (`http://www.brics.dk/`), Centre of the Danish National Research Foundation.
‡Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
E-mail: {`danvy,lrn`}`@brics.dk`

Contents

1 Introduction **3**

2 Call-by-name and call-by-value λ-calculi **4**
2.1 Call by value 4
2.2 Call by name 4
2.3 Evaluation-order independent reduction 5
2.4 Stuck terms 5

3 The meta language of the one-pass CPS transformation **5**

4 The one-pass CPS transformation **8**

5 The goal **10**

6 Plotkin's four lemmas **11**

7 Plotkin's double proof **14**

8 Conclusion **15**

List of Figures

1 The left-to-right, call-by-value CPS transformation in one pass . 8

1 Introduction

In "Call-by-name, call-by-value and the λ-calculus" [9], Plotkin provided the first formalization of the transformation into continuation-passing style (CPS) [10, 11]. Plotkin's CPS transformation is first-order. As most λ-encodings, it is plagued with so-called *administrative redexes*, i.e., redexes that are only artefacts of the λ-encoding. Their reduction steps are interspersed with the reduction steps corresponding to actual reduction steps in the original direct-style program. In the early 90s, a higher-order version of the CPS transformation was developed that eliminates all administrative redexes at transformation time, in one pass [1, 3, 12]. This higher-order version has been formalized in several ways: using a notion of 'schematic' continuations [3], using higher-order rewriting [4], and using logical relations [2]. None of these various ways of formalizing a one-pass CPS transformation, however, match Plotkin's original proof technique.

Does it mean that Plotkin's proof technique is inherently first-order? In this article, we answer this question negatively. We adapt his proof technique to the higher-order setting of one-pass CPS transformations.

To tackle administrative reductions, in his formalization, Plotkin introduced a so-called *colon translation* that eliminates administrative redexes until a redex is reached that corresponds to an actual redex in the original program. Our goal here is to present a higher-order colon translation that yields the same effect for a one-pass CPS transformation, thus adapting Plotkin's original proof technique to a higher-order operational setting.

Prerequisites: We assume a basic familiarity with Plotkin's work, as can be gathered from his original article [9] or from Hatcliff and Danvy's revisitation [7]. The one-pass CPS transformation is presented in Danvy and Filinski's article [3] (Section 3 of that article reviews administrative redexes and the colon translation). A pictorial presentation of actual and administrative reductions can be found in Section 3 of Danvy, Dzafic, and Pfenning's 1999 article [2]. Nevertheless, we have tried to make the present article stand alone.

Overview: Section 2 reviews call by name and call by value in the untyped λ-calculus. Sections 3 and 4 present the one-pass CPS transformation and its meta-language. Section 5 states our goal (proving Plotkin's Indifference and Simulation theorems) and our means (a higher-order colon translation). Section 6 recasts Plotkin's four lemmas and Section 7 restates his proof. Section 8 concludes.

2 Call-by-name and call-by-value λ-calculi

We define the untyped λ-calculus by giving its syntax and two semantics, one for call by value (CBV) and the other for call by name (CBN).

Definition 1 (Syntax of the λ-calculus with uninterpreted constants)

$$e \quad ::= \quad x \mid \lambda x.e \mid e \mathbin{@} e \mid c$$

We only distinguish expressions up to renaming of bound variables. So for example, $\lambda x.x$ and $\lambda y.y$ are considered equal.

2.1 Call by value

The CBV semantics of closed expressions is given using evaluation contexts in the style of Felleisen [5].

The evaluation contexts are as follows.

$$\mathrm{C_v}[\,] \quad ::= \quad [\,] \mid \mathrm{C_v}[\,] \mathbin{@} e \mid v_\mathrm{v} \mathbin{@} \mathrm{C_v}[\,]$$

where v_v is a value.

Values form a subset of expressions and are defined as follows.

$$v_\mathrm{v} \quad ::= \quad \lambda x.e \mid c$$

The reduction rule is defined as follows.

$$\mathrm{C_v}[(\lambda x.e) \mathbin{@} v_\mathrm{v}] \longmapsto_\mathrm{v} \mathrm{C_v}[e[v_\mathrm{v}/x]]$$

where $e[v/x]$ is the standard capture-avoiding substitution.

$\textsc{Eval}_\mathrm{v}$ is the following partial function:

$$\begin{cases} \textsc{Eval}_\mathrm{v}(e) = v_\mathrm{v} & \text{iff } e \longmapsto_\mathrm{v}^i v_\mathrm{v} \text{ for some value } v_\mathrm{v} \text{ and integer } i \\ \textsc{Eval}_\mathrm{v}(e) \text{ is undefined} & \text{otherwise} \end{cases}$$

where $\longmapsto_\mathrm{v}^i$ denotes i iterations of $\longmapsto_\mathrm{v}$.

2.2 Call by name

We give the CBN semantics for closed expressions as in Section 2.1.

The evaluation contexts are:

$$\mathrm{C_n}[\,] \quad ::= \quad [\,] \mid \mathrm{C_n}[\,] \mathbin{@} e$$

The values are:

$$v_\mathrm{n} \quad ::= \quad \lambda x.e \mid c$$

The reduction rule is:

$$\mathrm{C_n}[(\lambda x.e) \mathbin{@} e'] \longmapsto_\mathrm{n} \mathrm{C_n}[e[e'/x]]$$

$\textsc{Eval}_\mathrm{n}$ is the following partial function:

$$\begin{cases} \textsc{Eval}_\mathrm{n}(e) = v_\mathrm{n} & \text{iff } e \longmapsto_\mathrm{n}^i v_\mathrm{n} \text{ for some value } v_\mathrm{n} \text{ and integer } i \\ \textsc{Eval}_\mathrm{n}(e) \text{ is undefined} & \text{otherwise} \end{cases}$$

2.3 Evaluation-order independent reduction

In the remainder of this article, the arrow $\longmapsto$ (without annotation) corresponds to a reduction that is legal in both call by value and call by name. In other words, the $\longmapsto$ relation is the intersection of the $\longmapsto_{\mathrm{v}}$ and $\longmapsto_{\mathrm{n}}$ relations.

2.4 Stuck terms

In both the call-by-value and the call-by-name semantics, we can write closed expressions that are not values, but for which there are no legal reductions. Plotkin said that such expressions "stick".

The stuck closed expressions w.r.t. the CBV semantics are:

$$\mathrm{Sticks_v} \quad ::= \quad c \mathbin{@} v_{\mathrm{v}} \mid \mathrm{Sticks_v} \mathbin{@} e \mid v_{\mathrm{v}} \mathbin{@} \mathrm{Sticks_v}$$

or simply $\mathrm{C_v}[c \mathbin{@} v_{\mathrm{v}}]$.

The stuck expressions are disjoint from the values. All closed expressions are either values, stuck, or there is a possible reduction.

The stuck closed expressions w.r.t. the CBN semantics are:

$$\mathrm{Sticks_n} \quad ::= \quad \mathrm{Sticks_n} \mathbin{@} e \mid c \mathbin{@} e$$

or simply $\mathrm{C_n}[c \mathbin{@} e]$.

Again the stuck expressions are disjoint from the values, and all closed expressions are either values, stuck, or allow a reduction.

3 The meta language of the one-pass CPS transformation

The one-pass CPS transformation maps direct-style λ-expressions into λ-expressions that can be reduced at transformation time, yielding λ-expressions in CPS. Therefore, the implementation language for the transformation contains two kinds of applications, two kinds of λ-abstractions, and two kinds of variables:

$$E \quad ::= \quad c \mid x \mid \lambda x.E \mid E \mathbin{@} E \mid X \mid \overline{\lambda} X.E \mid E \mathbin{\overline{@}} E$$

Following tradition we refer to overlined constructs as static and non-overlined ones as dynamic. The identifiers ranged over by X and x are called static and dynamic variables, respectively. Substituting an expression for a static variable is a static substitution, and substituting an expression for a dynamic variable is a dynamic substitution. As in Section 2, we only distinguish expressions up to renaming of bound variables, both for static and dynamic variables.

We define the following typing system for the meta language:

$$\tau \quad ::= \quad \textsc{Syntax} \mid \tau \to \tau$$

$$\frac{}{\Gamma \vdash c : \textsc{Syntax}} \qquad \frac{}{\Gamma \vdash x : \textsc{Syntax}} \qquad \frac{\Gamma \vdash E : \textsc{Syntax}}{\Gamma \vdash \lambda x.E : \textsc{Syntax}}$$

$$\frac{\Gamma \vdash E_0 : \textsc{Syntax} \qquad \Gamma \vdash E_1 : \textsc{Syntax}}{\Gamma \vdash E_0 \mathbin{@} E_1 : \textsc{Syntax}} \qquad \frac{\Gamma(X) = \tau}{\Gamma \vdash X : \tau}$$

$$\frac{\Gamma, X : \tau_1 \vdash E : \tau_2}{\Gamma \vdash \overline{\lambda} X.E : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash E_0 : \tau_1 \to \tau_2 \qquad \Gamma \vdash E_1 : \tau_1}{\Gamma \vdash E_0 \mathbin{\overline{@}} E_1 : \tau_2}$$

Any expression typeable in this typing system is simply typed and hence its reduction terminates and any evaluation strategy leads to the same normal form. We choose to use call by value, arbitrarily.

The meta-language values are:

$$v_{\mathrm{V}} \quad ::= \quad c \mid x \mid \lambda x.v_{\mathrm{V}} \mid v_{\mathrm{V}} \mathbin{@} v_{\mathrm{V}} \mid \overline{\lambda} X.E$$

N.B. If a meta-language value v_{V} has type Syntax, it is an expression in the λ-calculus as defined in Section 2. In particular, v_{V} is free of static variables.

The evaluation contexts are:

$$\mathrm{C_V}[\,] \quad ::= \quad [\,] \mid \lambda x.\mathrm{C_V}[\,] \mid \mathrm{C_V}[\,] \mathbin{@} E \mid v_{\mathrm{V}} \mathbin{@} \mathrm{C_V}[\,] \mid \mathrm{C_V}[\,] \mathbin{\overline{@}} E \mid v_{\mathrm{V}} \mathbin{\overline{@}} \mathrm{C_V}[\,]$$

The reduction rule only allows us to reduce static β-redexes:

$$\mathrm{C_V}[(\overline{\lambda} X.E) \mathbin{\overline{@}} v_{\mathrm{V}}] \longmapsto_{\mathrm{V}} \mathrm{C_V}[E[v_{\mathrm{V}}/X]]$$

Proposition 1 (Context nesting) *Since nested contexts are themselves contexts, the reduction rule satisfies the following property.*

$$E \longmapsto_{\mathrm{V}} E' \iff \mathrm{C_V}[E] \longmapsto_{\mathrm{V}} \mathrm{C_V}[E']$$

Proof: Omitted. □

We define the static evaluation function, $\textsc{Eval}_{\mathrm{V}}$, by

$$\textsc{Eval}_{\mathrm{V}}(E) = v_{\mathrm{V}} \iff E \longmapsto_{\mathrm{V}}^{*} v_{\mathrm{V}}$$

where $\longmapsto_{\mathrm{V}}^{*}$ denotes zero or more iterations of $\longmapsto_{\mathrm{V}}$. By definition reductions on well-typed expressions are strongly normalizing, so $\textsc{Eval}_{\mathrm{V}}$ is total on these.

We treat the meta language as a higher-order language in the sense that we equate expressions up to β-equivalence (written $\equiv_{\mathrm{V}}$). We only use simply typed and thus strongly normalizing expressions, so we can equate any expression to the value it reduces to.

The dynamic substitution of a *closed* value, as used for λ-expressions in Section 2, can be extended directly to meta-language terms. The interaction between static and dynamic substitutions satisfies the following property.

Proposition 2 (Substitution) *If E is a meta-language expression, v_{V} is a meta-language value, and e is a closed λ-expression, then*

$$E[v_{\mathrm{V}}/X][e/x] = E[e/x][v_{\mathrm{V}}[e/x]/X].$$

Proof: By structural induction on E.

Cases $E = c$, x, and Y $(Y \neq X)$**:** In these cases X is not free in E, and hence X is not free in $E[e/x]$. It follows that

$$E[v_{\mathrm{V}}/X][e/x] = E[e/x] = E[e/x][v_{\mathrm{V}}[e/x]/X].$$

Cases $E = E_1 \mathbin{@} E_2$ and $E_1 \mathbin{\overline{@}} E_2$**:** These cases both follow directly from the induction hypothesis and the definition of substitution.

Case $E = X$**:** $X[v_{\mathrm{V}}/X][e/x] = v_{\mathrm{V}}[e/x] = X[e/x][v_{\mathrm{V}}[e/x]/X]$

Case $E = \lambda y.E_1$ and $\overline{\lambda}Y.E_1$**:** Since we equate expressions up to alpha-renaming of bound variables, we can assume that y is not free in v_{V} and $y \neq x$ as well as $Y \neq X$. These cases are thus similar to the $E_1 \mathbin{@} E_2$ case.

□

Corollary 1 (Dynamic substitution respects static β-equivalence) *If E and E' are meta-language expressions, v is a value, and x is a dynamic variable, then*

$$E \longmapsto_{\mathrm{V}}^{*} E' \Longrightarrow E[v/x] \longmapsto_{\mathrm{V}}^{*} E'[v/x].$$

I.e., if $E \equiv_{\mathrm{V}} E'$ then $E[v/x] \equiv_{\mathrm{V}} E'[v/x]$.

Proof: It suffices to show that if $E \longmapsto_{\mathrm{V}} E'$ then $E[v/x] \longmapsto_{\mathrm{V}} E'[v/x]$. Since $E \longmapsto_{\mathrm{V}} E$ only if there exists a context $\mathrm{C_V}[\,]$ such that $E = \mathrm{C_V}[(\overline{\lambda}X.E_1) \mathbin{\overline{@}} V] \longmapsto_{\mathrm{V}} \mathrm{C_V}[E_1[V/X]] = E'$, the proof is by induction on the structure of the context.

Case $\mathrm{C_V}[\,] = [\,]$**:** In this case $E = (\overline{\lambda}X.E_1) \mathbin{\overline{@}} V$ and $E' = E_1[V/X]$.

$$\begin{array}{rcll} ((\overline{\lambda}X.E_1) \mathbin{\overline{@}} V)[v/x] & = & (\overline{\lambda}X.E_1[v/x]) \mathbin{\overline{@}} V[v/x] & \\ & \longmapsto_{\mathrm{V}} & E_1[v/x][V[v/x]/X] & \\ & = & E_1[V/X][v/x] & \text{by Proposition 2} \end{array}$$

Case $\mathrm{C_V}[\,] = \mathrm{C}_1[\,] \mathbin{\overline{@}} E_2$**:**

$$\begin{array}{rcll} E[v/x] & = & \mathrm{C}_1[(\overline{\lambda}X.E_1) \mathbin{\overline{@}} V][v/x] & \\ & = & (\mathrm{C}_1[(\overline{\lambda}X.E_1) \mathbin{\overline{@}} V] \mathbin{\overline{@}} E_2)[v/x] & \\ & = & \mathrm{C}_1[(\overline{\lambda}X.E_1) \mathbin{\overline{@}} V][v/x] \mathbin{\overline{@}} E_2[v/x] & \\ & \longmapsto_{\mathrm{V}} & \mathrm{C}_1[E_1[V/X]][v/x] \mathbin{\overline{@}} E_2[v/x] & \text{by I.H. and Prop. 1} \\ & = & (\mathrm{C}_1[E_1[V/X]] \mathbin{\overline{@}} E_2)[v/x] & \\ & = & \mathrm{C_V}[E_1[V/X]][v/x] & \\ & = & E'[v/x] & \end{array}$$

Cases $\mathrm{C_V}[\,] = \mathrm{C}_1[\,] \mathbin{@} E_2$, $V \mathbin{\overline{@}} \mathrm{C}_1[\,]$, $V \mathbin{@} \mathrm{C}_1[\,]$, and $\lambda y.\mathrm{C}_1[\,]$**:** Similar to the previous case.

□

This corollary shows that we can use an arbitrary representative for the equivalence classes up to $\equiv_{\mathrm{V}}$ when performing dynamic reductions. Therefore, in the remainder of this article, we will use $=$ instead of $\equiv_{\mathrm{V}}$.

4 The one-pass CPS transformation

We consider the one-pass version of Plotkin's (left-to-right) call-by-value CPS transformation [3]. (Many others exist, depending, e.g., on the evaluation order of the source language [6].)

$$
\begin{array}{rcl}
[\![\cdot]\!] & : & \textsc{Syntax} \to (\textsc{Syntax} \to \textsc{Syntax}) \to \textsc{Syntax} \\
[\![c]\!] & = & \overline{\lambda}\mathrm{K}.\mathrm{K} \mathbin{\overline{@}} c \\
[\![x]\!] & = & \overline{\lambda}\mathrm{K}.\mathrm{K} \mathbin{\overline{@}} x \\
[\![\lambda x.e]\!] & = & \overline{\lambda}\mathrm{K}.\mathrm{K} \mathbin{\overline{@}} \lambda x.\lambda\mathrm{k}.[\![e]\!]' \mathbin{\overline{@}} \mathrm{k} \\
[\![e_1 \mathbin{@} e_2]\!] & = & \overline{\lambda}\mathrm{K}.[\![e_1]\!] \mathbin{\overline{@}} \overline{\lambda}\mathrm{V}_1.[\![e_2]\!] \mathbin{\overline{@}} \overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\mathrm{K} \mathbin{\overline{@}} \mathrm{v}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\cdot]\!]' & : & \textsc{Syntax} \to \textsc{Syntax} \to \textsc{Syntax} \\
[\![c]\!]' & = & \overline{\lambda}\mathrm{K}.\mathrm{K} \mathbin{@} c \\
[\![x]\!]' & = & \overline{\lambda}\mathrm{K}.\mathrm{K} \mathbin{@} x \\
[\![\lambda x.e]\!]' & = & \overline{\lambda}\mathrm{K}.\mathrm{K} \mathbin{@} \lambda x.\lambda\mathrm{k}.[\![e]\!]' \mathbin{\overline{@}} \mathrm{k} \\
[\![e_1 \mathbin{@} e_2]\!]' & = & \overline{\lambda}\mathrm{K}.[\![e_1]\!] \mathbin{\overline{@}} \overline{\lambda}\mathrm{V}_1.[\![e_2]\!] \mathbin{\overline{@}} \overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \mathrm{K}
\end{array}
$$

The transformation uses some fixed variables that are assumed to be "fresh", i.e., that do not occur in the program to be transformed. These variables are denoted by 'k' and 'v', which are dynamic variables. We also fix the static variables K, V_1, and V_2. The static variables cannot clash with variables in the original program, but they must be distinct. All these fresh variables are written in Roman font.

Figure 1: The left-to-right, call-by-value CPS transformation in one pass

The one-pass CPS transformation is defined inductively with the two functions $[\![\cdot]\!]$ and $[\![\cdot]\!]'$ displayed in Figure 1. Whereas Plotkin's CPS transformation uses only one function, the one-pass transformation uses two, depending on whether the continuation is known statically ($[\![\cdot]\!]$) or not ($[\![\cdot]\!]'$). These two

functions map λ-expressions as defined in Section 2 to expressions in the meta language defined in Section 3. Their type are easily inferred using the rules of Section 3.

We define the function Ψ to map direct-style values to CPS values:

$$\begin{array}{rcl} \Psi(c) & = & c \\ \Psi(\lambda x.e) & = & \lambda x.\lambda \mathrm{k}.[\![e]\!]' \,\overline{@}\, \mathrm{k} \end{array}$$

N.B. $[\![v_\mathrm{v}]\!] = \overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,\Psi(v_\mathrm{v})$ and $[\![v_\mathrm{v}]\!]' = \overline{\lambda}\mathrm{K}.\mathrm{K}\,@\,\Psi(v_\mathrm{v})$.

The two-level η-redex $\lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v}$ coerces a meta-language expression of type Syntax $\to$ Syntax into a meta-language expression of type Syntax, and therefore, the two CPS-transformation functions are related as follows.

Proposition 3 *If e is a λ-expression and κ is a meta-language expression of type* Syntax $\to$ Syntax *then*

$$[\![e]\!]' \,\overline{@}\, \lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v} \longmapsto^* [\![e]\!]\,\overline{@}\,\kappa$$

remembering that we equate expressions up to static β-equivalence and that the identifier v *is fresh, so it is not free in κ.*

Proof: By structural induction on e.

Case $e = x$:

$$\begin{array}{rcll} [\![x]\!]' \,\overline{@}\, \lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v} & = & (\lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v})\,@\,x & \\ & \longmapsto & \kappa\,\overline{@}\,\mathrm{v}[x/\mathrm{v}] & \\ & = & \kappa\,\overline{@}\,x & \text{by Corollary 1} \\ & = & [\![x]\!]\,\overline{@}\,\kappa & \end{array}$$

Case $e = \lambda x.e_1$:

$$\begin{array}{rcl} [\![\lambda x.e_1]\!]' \,\overline{@}\, \lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v} & = & (\lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v})\,@\,\Psi(\lambda x.e_1) \\ & \longmapsto & \kappa\,\overline{@}\,\mathrm{v}[\Psi(\lambda x.e_1)/\mathrm{v}] \\ & = & \kappa\,\overline{@}\,\Psi(\lambda x.e_1) \\ & = & [\![\lambda x.e_1]\!]\,\overline{@}\,\kappa \end{array}$$

Case $e = e_1\,@\,e_2$:

$$\begin{array}{rcl} [\![e_1\,@\,e_2]\!]' \,\overline{@}\, \lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v} & = & [\![e_1]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_1.[\![e_2]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_2.\mathrm{V}_1\,@\,\mathrm{V}_2\,@\,\lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v} \\ & = & [\![e_1\,@\,e_2]\!]\,\overline{@}\,\kappa \end{array}$$

□

5 The goal

Plotkin proved three properties of the CPS transformation: Indifference, Simulation, and Translation. We prove the first two for the one-pass CPS transformation.

Theorem 1 (Indifference) $\textsc{Eval}_{\mathrm{v}}([\![e]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}.\mathrm{V}) = \textsc{Eval}_{\mathrm{n}}([\![e]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}.\mathrm{V})$.

The Indifference theorem formalizes a key property of CPS, namely that CPS programs are evaluation-order independent.

Theorem 2 (Simulation) $\textsc{Eval}_{\mathrm{v}}([\![e]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}.\mathrm{V}) = \Psi(\textsc{Eval}_{\mathrm{v}}(e))$.

where Ψ was defined in Section 4, just before Proposition 3.

The Simulation theorem formalizes the correctness of the call-by-value CPS transformation. For one point, the theorem shows that termination is preserved, but more essentially, it says that evaluating a CPS-transformed program yields the CPS counterpart of the result of evaluating the original program.

To prove Indifference and Simulation, Plotkin used four lemmas. We prove the same four lemmas for the one-pass CPS transformation. Plotkin's proof then applies as is (see Section 7).

To handle administrative redexes, Plotkin introduced a colon translation. This translation is an infix operation mapping an expression and a continuation to the CPS counterpart of that expression applied to the continuation, but bypassing the initial administrative redexes introduced by the CPS transformation. To account for meta-level reductions at CPS-transformation time, we define a higher-order version of Plotkin's colon translation.

Definition 2 (Higher-order colon translation)

$$
\begin{array}{rcll}
c : \kappa & = & \kappa\,\overline{@}\,\Psi(c) & \\
\lambda x.e : \kappa & = & \kappa\,\overline{@}\,\Psi(\lambda x.e) & \\
v_1 @ v_2 : \kappa & = & \Psi(v_1) @ \Psi(v_2) @ \lambda v.\kappa\,\overline{@}\,v & \\
v_1 @ e_2 : \kappa & = & e_2 : \overline{\lambda} V_2.\Psi(v_1) @ V_2 @ \lambda v.\kappa\,\overline{@}\,v & e_2 \textit{ not a value} \\
e_1 @ e_2 : \kappa & = & e_1 : \overline{\lambda} V_1.[\![e_2]\!]\,\overline{@}\,\overline{\lambda} V_2.\, V_1 @ V_2 @ \lambda v.\kappa\,\overline{@}\,v & e_1 \textit{ not a value}
\end{array}
$$

Unlike Plotkin's colon translation, which is first order and interspersed with the actual reductions, this colon translation is higher order (as indicated by the overlines) and its static, administrative reductions occur before the actual, dynamic reductions.

6 Plotkin's four lemmas

Lemma 1 (Substitution lemma)
For all λ-expressions e and all closed values v and variables x,

$$\begin{array}{rcl} [\![e]\!][\Psi(v)/x] & = & [\![e[v/x]]\!] \\ [\![e]\!]'[\Psi(v)/x] & = & [\![e[v/x]]\!]' \end{array}$$

Proof: By structural induction on e.

Case $e = x$:

$$\begin{array}{rcl} [\![x]\!][\Psi(v)/x] & = & (\overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,x)[\Psi(v)/x] \\ & = & \overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,\Psi(v) \\ & = & [\![v]\!] \\ & = & [\![x[v/x]]\!] \end{array}$$

Case $e = y$:

$$\begin{array}{rcl} [\![y]\!][\Psi(v)/x] & = & (\overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,y)[\Psi(v)/x] \\ & = & \overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,y \\ & = & [\![y[v/x]]\!] \end{array}$$

Case $e = \lambda y.e_1$, where $y \neq x$

$$\begin{array}{rcll} [\![\lambda y.e_1]\!][\Psi(v)/x] & = & (\overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,\lambda y.\lambda\mathrm{k}.[\![e_1]\!]'\,\overline{@}\,\mathrm{k})[\Psi(v)/x] & \\ & = & \overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,((\lambda y.\lambda\mathrm{k}.[\![e_1]\!]'\,\overline{@}\,\mathrm{k})[\Psi(v)/x]) & \\ & = & \overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,(\lambda y.\lambda\mathrm{k}.[\![e_1]\!]'[\Psi(v)/x]\,\overline{@}\,\mathrm{k}) & \\ & = & \overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,(\lambda y.\lambda\mathrm{k}.[\![e_1[v/x]]\!]'\,\overline{@}\,\mathrm{k}) & \text{by I.H.} \\ & = & [\![\lambda y.e_1[v/x]]\!] & \end{array}$$

Case $e = e_1 \mathbin{@} e_2$:

$$\begin{array}{rcl} & & [\![e_1 \mathbin{@} e_2]\!][\Psi(v)/x] \\ & = & \overline{\lambda}\mathrm{K}.([\![e_1]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_1.[\![e_2]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\mathrm{K}\,\overline{@}\,\mathrm{v})[\Psi(v)/x] \\ & = & \overline{\lambda}\mathrm{K}.([\![e_1]\!][\Psi(v)/x])\,\overline{@}\,\overline{\lambda}\mathrm{V}_1.([\![e_2]\!][\Psi(v)/x])\,\overline{@}\,\overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\mathrm{K}\,\overline{@}\,\mathrm{v} \\ & = & \overline{\lambda}\mathrm{K}.[\![e_1[v/x]]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_1.[\![e_2[v/x]]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v} \quad \text{by I.H.} \\ & = & [\![e_1[v/x] \mathbin{@} e_2[v/x]]\!] \\ & = & [\![(e_1 \mathbin{@} e_2)[v/x]]\!] \end{array}$$

The cases for $[\![\cdot]\!]'$ are similar. □

Lemma 2 (Administrative reductions)
If e is a closed λ-expression and κ is a closed meta-language expression of type Syntax → Syntax *then $[\![e]\!]\,\overline{@}\,\kappa \longmapsto_{\mathrm{V}}^{+} e : \kappa$ (i.e., they are equal modulo $\equiv_{\mathrm{V}}$).*

Proof: By structural induction on e.

Case $e = c$:

$$\begin{array}{rcl} [\![c]\!]\,\overline{@}\,\kappa & = & (\overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,c)\,\overline{@}\,\kappa \\ & \longmapsto_{\mathrm{V}} & \kappa\,\overline{@}\,c \\ & = & c : \kappa \end{array}$$

Case $e = \lambda x.e_1$**:**

$$\begin{array}{lll} [\![\lambda x.e_1]\!]\,\overline{@}\,\kappa & = & (\overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,\Psi(\lambda x.e_1))\,\overline{@}\,\kappa \\ & \longmapsto_{\mathrm{V}} & \kappa\,\overline{@}\,\Psi(\lambda x.e_1) \\ & = & \lambda x.e_1 : \kappa \end{array}$$

Case $e = v_1 \mathbin{@} v_2$**:**

$$\begin{array}{lll} & & [\![v_1 \mathbin{@} v_2]\!]\,\overline{@}\,\kappa \\ & = & (\overline{\lambda}\mathrm{K}.[\![v_1]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_1.[\![v_2]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\mathrm{K}\,\overline{@}\,\mathrm{v})\,\overline{@}\,\kappa \\ & = & (\overline{\lambda}\mathrm{K}.(\overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,\Psi(v_1))\overline{@} \\ & & \qquad \overline{\lambda}\mathrm{V}_1.(\overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,\Psi(v_2))\,\overline{@}\,\overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\mathrm{K}\,\overline{@}\,\mathrm{v})\overline{@}\kappa \\ & \longmapsto_{\mathrm{V}}^{5} & \Psi(v_1) \mathbin{@} \Psi(v_2) \mathbin{@} \lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v} \\ & = & v_1 \mathbin{@} v_2 : \kappa \end{array}$$

Case $e = v_1 \mathbin{@} e_2$**:**

$$\begin{array}{lll} & & [\![v_1 \mathbin{@} e_2]\!]\,\overline{@}\,\kappa \\ & = & (\overline{\lambda}\mathrm{K}.[\![v_1]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_1.[\![e_2]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\mathrm{K}\,\overline{@}\,\mathrm{v})\,\overline{@}\,\kappa \\ & = & (\overline{\lambda}\mathrm{K}.(\overline{\lambda}\mathrm{K}.\mathrm{K}\,\overline{@}\,\Psi(v_1))\,\overline{@}\,\overline{\lambda}\mathrm{V}_1.[\![e_2]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\mathrm{K}\,\overline{@}\,\mathrm{v})\,\overline{@}\,\kappa \\ & \longmapsto_{\mathrm{V}}^{3} & [\![e_2]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_2.\Psi(v_1) \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v} \\ & \longmapsto_{\mathrm{V}}^{*} & e_2 : \overline{\lambda}\mathrm{V}_2.\Psi(v_1) \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v} \qquad \text{by I.H.} \\ & = & v_1 \mathbin{@} e_2 : \kappa \end{array}$$

Case $e = e_1 \mathbin{@} e_2$**:**

$$\begin{array}{lll} & & [\![e_1 \mathbin{@} e_2]\!]\,\overline{@}\,\kappa \\ & = & (\overline{\lambda}\mathrm{K}.[\![e_1]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_1.[\![e_2]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\mathrm{K}\,\overline{@}\,\mathrm{v})\,\overline{@}\,\kappa \\ & \longmapsto_{\mathrm{V}} & [\![e_1]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_1.[\![e_2]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v} \\ & \longmapsto_{\mathrm{V}}^{*} & e_1 : \overline{\lambda}\mathrm{V}_1.[\![e_2]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda\mathrm{v}.\kappa\,\overline{@}\,\mathrm{v} \qquad \text{by I.H.} \\ & = & e_1 \mathbin{@} e_2 : \kappa \end{array}$$

This enumeration accounts for all expressions e. □

Lemma 3 (Single-step simulation)
If e is a closed λ-expression, κ is a closed meta-language expression of type Syntax → Syntax*, and $e \longmapsto_{\mathrm{v}} e'$ then $e : \kappa \longmapsto^{+} e' : \kappa$.*

Proof: By structural induction on the evaluation context in the derivation of $e \longmapsto_{\mathrm{v}} e'$.

Case $\mathrm{C}[\,] = [\,]$, i.e., $(\lambda x.e) \mathbin{@} v \longmapsto_{\mathrm{v}} e[v/x]$**:**

$$
\begin{array}{lll}
(\lambda x.e) \mathbin{@} v : \kappa & & \\
\quad = & \Psi(\lambda x.e) \mathbin{@} \Psi(v) \mathbin{@} \lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v} & \\
\quad = & (\lambda x.\lambda \mathrm{k}.[\![e]\!]' \mathbin{\overline{@}} \mathrm{k}) \mathbin{@} \Psi(v) \mathbin{@} \lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v} & \\
\longmapsto & ((\lambda \mathrm{k}.[\![e]\!]' \mathbin{\overline{@}} \mathrm{k})[\Psi(v)/x]) \mathbin{@} \lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v} & \\
\quad = & (\lambda \mathrm{k}.[\![e]\!]'[\Psi(v)/x] \mathbin{\overline{@}} \mathrm{k}) \mathbin{@} \lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v} & \text{by Corollary 1} \\
\quad = & (\lambda \mathrm{k}.[\![e[v/x]]\!]' \mathbin{\overline{@}} \mathrm{k}) \mathbin{@} \lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v} & \text{by Lemma 1} \\
\longmapsto & [\![e[v/x]]\!]' \mathbin{\overline{@}} (\lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v}) & \\
\longmapsto^* & [\![e[v/x]]\!] \mathbin{\overline{@}} \kappa & \text{by Proposition 3} \\
\quad = & e[v/x] : \kappa & \text{by Lemma 2}
\end{array}
$$

Case $\mathrm{C}[\,] = \mathrm{C}_1[\,] \mathbin{@} e_2$, i.e., $\mathrm{C}_1[e_1] \mathbin{@} e_2 \longmapsto_{\mathrm{v}} \mathrm{C}_1[e_1'] \mathbin{@} e_2$ derived from $e_1 \longmapsto_{\mathrm{v}} e_1'$**:**

$$
\begin{array}{lll}
\mathrm{C}_1[e_1] \mathbin{@} e_2 : \kappa & & \\
\quad = & \mathrm{C}_1[e_1] : \overline{\lambda}\mathrm{V}_1.[\![e_2]\!] \mathbin{\overline{@}} \overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v} & \\
\longmapsto^+ & \mathrm{C}_1[e_1'] : \overline{\lambda}\mathrm{V}_1.[\![e_2]\!] \mathbin{\overline{@}} \overline{\lambda}\mathrm{V}_2.\mathrm{V}_1 \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v} & \text{by I.H.} \\
\quad = & e' & \text{to give it a name}
\end{array}
$$

- If $\mathrm{C}_1[e_1']$ is not a value, then $\mathrm{C}_1[e_1'] \mathbin{@} e_2 : \kappa = e'$
- If $\mathrm{C}_1[e_1']$ is a value then

$$
\begin{array}{lll}
e' & = & [\![e_2]\!] \mathbin{\overline{@}} \overline{\lambda}\mathrm{V}_2.(\Psi(\mathrm{C}_1[e_1']) \mathbin{@} \mathrm{V}_2) \mathbin{@} \lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v} \\
 & = & e_2 : \overline{\lambda}\mathrm{V}_2.\Psi(\mathrm{C}_1[e_1']) \mathbin{@} \mathrm{V}_2 \mathbin{@} (\lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v}) \qquad \text{by Lemma 2} \\
 & = & e''
\end{array}
$$

  - If e_2 is not a value, then $\mathrm{C}_1[e_1'] \mathbin{@} e_2 : \kappa = e''$.
  - If e_2 is a value, then

$$
\begin{array}{rcl}
e'' & = & \Psi(\mathrm{C}_1[e_1']) \mathbin{@} \Psi(e_2) \mathbin{@} \lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v} \\
 & = & \mathrm{C}_1[e_1'] \mathbin{@} e_2 : \kappa
\end{array}
$$

Case $\mathrm{C}[\,] = v_1 \mathbin{@} \mathrm{C}_1[\,]$, i.e., $v_1 \mathbin{@} \mathrm{C}_1[e_2] \longmapsto_{\mathrm{v}} v_1 \mathbin{@} \mathrm{C}_1[e_2']$ derived from $e_2 \longmapsto_{\mathrm{v}} e_2'$**:**

$$
\begin{array}{lll}
v_1 \mathbin{@} \mathrm{C}_1[e_2] : \kappa & = & \mathrm{C}_1[e_2] : \overline{\lambda}\mathrm{V}_2.\Psi(v_1) \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v} \\
 & \longmapsto^+ & \mathrm{C}_1[e_2'] : \overline{\lambda}\mathrm{V}_2.\Psi(v_1) \mathbin{@} \mathrm{V}_2 \mathbin{@} \lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v} \qquad \text{by I.H.} \\
 & = & e'
\end{array}
$$

- If $\mathrm{C}_1[e_2']$ is a not value then $v_1 \mathbin{@} \mathrm{C}_1[e_2'] : \kappa = e'$.
- If $\mathrm{C}_1[e_2']$ is a value then

$$
\begin{array}{rcl}
e' & = & \Psi(v_1) \mathbin{@} \Psi(\mathrm{C}_1[e_2']) \mathbin{@} \lambda \mathrm{v}.\kappa \mathbin{\overline{@}} \mathrm{v} \\
 & = & v_1 \mathbin{@} e_2' : \kappa
\end{array}
$$

□

Lemma 4 (Coincidence of stuck expressions)
If $e \in \text{Sticks}_\text{v}$ and κ is a closed meta-language expression with type SYNTAX $\to$ SYNTAX*, then $e : \kappa \in \text{Sticks}_\text{n} \cap \text{Sticks}_\text{v}$.*

Proof: By induction on the structure of the stuck expression e.

Case $e = c \mathbin{@} v$**:**

$$\begin{array}{rcl}(c \mathbin{@} v) : \kappa & = & \Psi(c) \mathbin{@} \Psi(v) \mathbin{@} \lambda\text{v}.\kappa \mathbin{\overline{@}} \text{v} \\ & = & c \mathbin{@} \Psi(v) \mathbin{@} \lambda\text{v}.\kappa \mathbin{\overline{@}} \text{v}\end{array}$$

which is stuck since $c \mathbin{@} \Psi(v)$ is stuck both in call-by-name and call-by-value.

Case $e = v_1 \mathbin{@} e_2$ where $e_2 \in \text{Sticks}_\text{v}$**:**

$$(v_1 \mathbin{@} e_2) : \kappa \quad = \quad e_2 : \overline{\lambda}\text{V}_2.\Psi(v_1) \mathbin{@} \text{V}_2 \mathbin{@} \lambda\text{v}.\kappa \mathbin{\overline{@}} \text{v}$$

which is stuck by induction hypothesis, e_2 being closed and structurally smaller that e.

Case $e = e_1 \mathbin{@} e_2$ where $e_1 \in \text{Sticks}_\text{v}$**:**

$$(e_1 \mathbin{@} e_2) : \kappa \quad = \quad e_1 : \overline{\lambda}\text{V}_1.[\![e_2]\!] \mathbin{\overline{@}} \overline{\lambda}\text{V}_2.\text{V}_1 \mathbin{@} \text{V}_2 \mathbin{@} \lambda\text{v}.\kappa \mathbin{\overline{@}} \text{v}$$

which is stuck by induction hypothesis, e_1 being closed and structurally smaller that e.

□

7 Plotkin's double proof

Plotkin proved both Indifference and Simulation using four lemmas similar to the ones in the previous section. Let us restate his proof.
Proof: Let us show that for any e, $\text{EVAL}_\text{v}([\![e]\!] \mathbin{\overline{@}} \overline{\lambda}\text{V}.\text{V})$ and $\text{EVAL}_\text{n}([\![e]\!] \mathbin{\overline{@}} \overline{\lambda}\text{V}.\text{V})$ either are both defined and yield the same result, which is $\Psi(\text{EVAL}_\text{v}(e))$, or they are both undefined.

1. If $\text{EVAL}_\text{v}(e) = v_\text{v}$ (i.e., if $e \longmapsto_\text{v}^i v_\text{v}$ for some i, by the definition of $\text{EVAL}_\text{v}(\cdot)$ in Section 2.1) then

$$\begin{array}{rcll}[\![e]\!] \mathbin{\overline{@}} \overline{\lambda}\text{V}.\text{V} & = & e : \overline{\lambda}\text{V}.\text{V} & \text{by Lemma 2} \\ & \longmapsto^* & v_\text{v} : \overline{\lambda}\text{V}.\text{V} & \text{by repeated use of Lemma 3} \\ & = & (\overline{\lambda}\text{V}.\text{V}) \mathbin{\overline{@}} \Psi(v_\text{v}) & \text{since } v_\text{v} \text{ is a value} \\ & = & \Psi(v_\text{v}) & \\ & = & \Psi(\text{EVAL}_\text{v}(e)) & \end{array}$$

   Therefore $\text{EVAL}_\text{v}([\![e]\!] \mathbin{\overline{@}} \overline{\lambda}\text{V}.\text{V}) = \Psi(\text{EVAL}_\text{v}(e))$. Furthermore, Lemma 3 only uses reductions in $\longmapsto$, and this relation is the intersection of the $\longmapsto_\text{v}$ and $\longmapsto_\text{n}$ relations. Therefore $\text{EVAL}_\text{v}([\![e]\!] \mathbin{\overline{@}} \overline{\lambda}\text{V}.\text{V}) = \text{EVAL}_\text{n}([\![e]\!] \mathbin{\overline{@}} \overline{\lambda}\text{V}.\text{V})$.

2. If $\textsc{Eval}_{\mathrm{v}}(e)$ is undefined then it is either because e reduces to a stuck term, or because e has an infinite reduction sequence.

   (a) In the first case there exists an $e' \in \mathrm{Sticks}_{\mathrm{v}}$ such that $e \longmapsto_{\mathrm{v}}^{*} e'$. In that case

   $$\begin{array}{rcll} [\![e]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}.\mathrm{V} & = & e : \overline{\lambda}\mathrm{V}.\mathrm{V} & \text{by Lemma 2} \\ & \longmapsto^{*} & e' : \overline{\lambda}\mathrm{V}.\mathrm{V} & \text{by repeated use of Lemma 3} \\ & \in & \mathrm{Sticks}_{\mathrm{n}} \cap \mathrm{Sticks}_{\mathrm{v}} & \text{by Lemma 4} \end{array}$$

   In words, whether one uses call by name or call by value, $[\![e]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}.\mathrm{V}$ reduces to a stuck term.

   (b) In the second case there exists a sequence of expressions

   $$e \longmapsto_{\mathrm{v}} e_1 \longmapsto_{\mathrm{v}} e_2 \longmapsto_{\mathrm{v}} \cdots \longmapsto_{\mathrm{v}} e_n \longmapsto_{\mathrm{v}} \cdots$$

   In that case

   $$\begin{array}{rcll} [\![e]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}.\mathrm{V} & = & e : \overline{\lambda}\mathrm{V}.\mathrm{V} & \text{by Lemma 2} \\ & \longmapsto^{+} & e_1 : \overline{\lambda}\mathrm{V}.\mathrm{V} & \text{by Lemma 3} \\ & \longmapsto^{+} & e_2 : \overline{\lambda}\mathrm{V}.\mathrm{V} & \text{by Lemma 3} \\ & \longmapsto^{+} & \cdots & \\ & \longmapsto^{+} & e_n : \overline{\lambda}\mathrm{V}.\mathrm{V} & \text{by Lemma 3} \\ & \longmapsto^{+} & \cdots & \end{array}$$

   In words, $[\![e]\!]\,\overline{@}\,\overline{\lambda}\mathrm{V}.\mathrm{V}$ has an infinite reduction sequence too, both in call by name and in call by value.

Together these two cases prove the Simulation and Indifference theorems. □

8 Conclusion

We have adapted Plotkin's four lemmas to the one-pass CPS transformation, which required us to introduce a higher-order colon translation. Given these four lemmas, Plotkin's Indifference and Simulation theorems and their proof apply directly to validate the one-pass CPS transformation.

Other λ-encodings exist that give rise to administrative reductions—for example, in denotational semantics, the denotation of a program is obtained by a syntax-directed translation into the λ-calculus. The resulting λ-term contains many administrative redexes that need to be dealt with to reason about programs. A good semantics-directed compiler is expected to eliminate these administrative redexes at compile time. For example, an identifier is typically mapped into an application of the environment, and scope resolution is expected from a compiler, so that variables are looked up in constant time at run time. Factoring out administrative redexes at compile time (a.k.a. staging [8]) is accepted good practice. When the compiler is written in a higher-order functional language (say, ML), administrative reductions can be represented as ML reductions. What we have shown here is a way to prove the correctness of this higher-order representation in the particular case of the CPS transformation.

Acknowledgments: This work is supported by the ESPRIT Working Group APPSEM (`http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/`). Part of it was carried out while the second author was visiting John Hannan at Penn State, in the fall of 2000. We are grateful to the anonymous reviewers and to Julia Lawall for perceptive comments.

References

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[2] Olivier Danvy, Belmina Dzafic, and Frank Pfenning. On proving syntactic properties of CPS programs. In *Third International Workshop on Higher-Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*, pages 19–31, Paris, France, September 1999. Also available as the technical report BRICS RS-99-23.

[3] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[4] Olivier Danvy and Kristoffer Høgsbro Rose. Higher-order rewriting and partial evaluation. In Tobias Nipkow, editor, *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Kyoto, Japan, March 1998. Springer-Verlag. Extended version available as the technical report BRICS-RS-97-46.

[5] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.

[6] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, January 1994. ACM Press.

[7] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(2):303–319, 1997. Extended version available as the technical report BRICS RS-97-7.

[8] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In Mark Scott Johnson and Ravi Sethi, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 86–96, St. Petersburg, Florida, January 1986. ACM Press.

[9] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[10] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, December 1993.

[11] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[12] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics*, number 598 in Lecture Notes in Computer Science, pages 294–311, Pittsburgh, Pennsylvania, March 1991. Springer-Verlag.

Recent BRICS Report Series Publications

RS-00-33 **Olivier Danvy and Lasse R. Nielsen.** ***A Higher-Order Colon Translation*****. December 2000. 17 pp. To appear in** ***Fifth International Symposium on Functional and Logic Programming*****, FLOPS '01 Proceedings, LNCS, 2001.**

RS-00-32 **John C. Reynolds.** ***What do Types Mean? - From Intrinsic to Extrinsic Semantics*****. December 2000.**

RS-00-31 **Bernd Grobauer and Julia L. Lawall.** ***Partial Evaluation of Pattern Matching in Strings, revisited*****. November 2000. 48 pp.**

RS-00-30 **Ivan B. Damgård and Maciej Koprowski.** ***Practical Threshold RSA Signatures Without a Trusted Dealer*****. November 2000. 14 pp.**

RS-00-29 **Luigi Santocanale.** ***The Alternation Hierarchy for the Theory of μ-lattices*****. November 2000. 44 pp. Extended abstract appears in** ***Abstracts from the International Summer Conference in Category Theory*****, CT2000, Como, Italy, July 16–22, 2000.**

RS-00-28 **Luigi Santocanale.** ***Free μ-lattices*****. November 2000. 51 pp. Short abstract appeared in** ***Proceedings of Category Theory 99*****, Coimbra, Portugal, July 19–24, 1999. Full version to appear in a special conference issue of the** ***Journal of Pure and Applied Algebra*****.**

RS-00-27 **Zoltán Ésik and Werner Kuich.** ***Inductive -Semirings*****. October 2000. 34 pp.**

RS-00-26 **František Čapkovič.** ***Modelling and Control of Discrete Event Dynamic Systems*****. October 2000. 58 pp.**

RS-00-25 **Zoltán Ésik.** ***Continuous Additive Algebras and Injective Simulations of Synchronization Trees*****. September 2000. 41 pp.**

RS-00-24 **Claus Brabrand and Michael I. Schwartzbach.** ***Growing Languages with Metamorphic Syntax Macros*****. September 2000.**

RS-00-23 **Luca Aceto, Anna Ingólfsdóttir, Mikkel Lykke Pedersen, and Jan Poulsen.** ***Characteristic Formulae for Timed Automata*****. September 2000. 23 pp.**