BRICS

Basic Research in Computer Science

BRICS RS-97-7 Hatcliff & Danvy: Thunks and the λ-Calculus

Thunks and the λ-Calculus

(Extended Version)

John Hatcliff
Olivier Danvy

BRICS Report Series **RS-97-7**

ISSN 0909-0878 **March 1997**

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/97/7/`

Thunks and the λ-calculus (extended version)

John Hatcliff

Oklahoma State University †

Olivier Danvy

BRICS*

University of Aarhus ‡

March 1997

Abstract

Plotkin, in his seminal article *Call-by-name, call-by-value and the λ-calculus*, formalized evaluation strategies and simulations using operational semantics and continuations. In particular, he showed how call-by-name evaluation could be simulated under call-by-value evaluation and vice versa. Since Algol 60, however, call-by-name is both implemented and simulated with thunks rather than with continuations. We recast this folk theorem in Plotkin's setting, and show that thunks, even though they are simpler than continuations, are sufficient for establishing all the correctness properties of Plotkin's call-by-name simulation.

Furthermore, we establish a new relationship between Plotkin's two continuation-based simulations $\mathcal{C}_{\mathrm{n}}$ and $\mathcal{C}_{\mathrm{v}}$, by *deriving* $\mathcal{C}_{\mathrm{n}}$ as the composition of our thunk-based simulation $\mathcal{T}$ and of $\mathcal{C}_{\mathrm{v}}^{+}$ — an extension of $\mathcal{C}_{\mathrm{v}}$ handling thunks. Almost all of the correctness properties of $\mathcal{C}_{\mathrm{n}}$ follow from the properties of $\mathcal{T}$ and $\mathcal{C}_{\mathrm{v}}^{+}$. This simplifies reasoning about call-by-name continuation-passing style.

We also give several applications involving factoring continuation-based transformations using thunks.

*Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

†Computer Science Department
219 Mathematical Sciences, Stillwater, OK 74078-1053, USA.
E-mail: hatcliff@a.cs.okstate.edu

‡Department of Computer Science
Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
E-mail: danvy@brics.dk

Contents

List of Figures

1 Introduction and Background

1.1 Motivation

Plotkin, in his seminal article *Call-by-name, call-by-value and the λ-calculus* [23], formalizes both call-by-name and call-by-value procedure calling mechanisms for λ-calculi. Call-by-name evaluation is described with a standardization theorem for the $\lambda\beta$-calculus. Call-by-value evaluation is described with a standardization theorem for a new calculus (the $\lambda\beta_v$-calculus). Plotkin then shows that call-by-name can be simulated by call-by-value and vice versa. The simulations also give interpretations of each calculus in terms of the other.

Both of Plotkin's simulations rely on *continuations* — a technique used earlier to model the meaning of jumps in the denotational-semantics approach to programming languages [33] and to express relationships between memory-management techniques [12], among other things [27]. Since Algol 60, however, programming wisdom has it that *thunks*[1] can be used to obtain a simpler simulation of call-by-name by call-by-value.[2]

Our aim is to clarify the properties of thunks with respect to Plotkin's classic study of evaluation strategies and continuation-passing styles [23]. We begin by defining a thunk-introducing transformation $\mathcal{T}$ and prove that thunks are sufficient for establishing all the technical properties Plotkin considered for his continuation-based call-by-name simulation $\mathcal{C}_{\mathrm{n}}$.[3]

Given this, one may question what rôle continuations actually play in $\mathcal{C}_{\mathrm{n}}$ since they are unnecessary for achieving a simulation. We show that the continuation-passing structure of $\mathcal{C}_{\mathrm{n}}$ can actually be obtained by extending

[1]The term "thunk" was coined to describe the compiled representation of delayed expressions in implementations of Algol 60 [17]. The terminology has been carried over and applied to various methods of delaying the evaluation of expressions [25].

[2]Plotkin acknowledges that thunks provide some simulation properties but states that "...these 'protecting by a λ' techniques do not seem to be extendable to a complete simulation and it is fortunate that the technique of continuations is available." [23, p. 147]. By "protecting by a λ", Plotkin refers to a representation of thunks as λ-abstractions with a dummy parameter. When we discussed our investigation of thunks with him, Plotkin told us that he had also found recently the "protecting by a λ" technique to be sufficient for a complete simulation [24].

[3]Plotkin actually gives a slightly different simulation $\mathcal{P}_{\mathrm{n}}$ [23, p. 153]. We note in Section 1.5.1 that Plotkin's **Translation** theorem for $\mathcal{P}_{\mathrm{n}}$ does not hold. A slight modification to $\mathcal{P}_{\mathrm{n}}$ gives the translation $\mathcal{C}_{\mathrm{n}}$ which does satisfy the **Translation** theorem. Therefore, in the present work, we will take $\mathcal{C}_{\mathrm{n}}$ along with Plotkin's original call-by-value continuation-based simulation $\mathcal{C}_{\mathrm{v}}$ as the canonical continuation-based simulations.

Plotkin's call-by-value continuation-based simulation $\mathcal{C}_{\mathrm{v}}$ to process the abstract representation of thunks and composing this extended transformation $\mathcal{C}_{\mathrm{v}}^{+}$ with $\mathcal{T}$, *i.e.*,[4]

$$\lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{n}}\langle\!\langle e \rangle\!\rangle \;=\; (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\langle e \rangle\!\rangle.$$

This establishes a previously unrecognized connection between $\mathcal{C}_{\mathrm{n}}$ and $\mathcal{C}_{\mathrm{v}}$ and gives insight into the structural similarities between call-by-name and call-by-value continuation-passing style.

We show that almost all of the technical properties that Plotkin established for $\mathcal{C}_{\mathrm{n}}$ follow from the properties of $\mathcal{C}_{\mathrm{v}}^{+}$ and $\mathcal{T}$. So as a byproduct, when reasoning about $\mathcal{C}_{\mathrm{n}}$ and $\mathcal{C}_{\mathrm{v}}$, it is often sufficient to reason about $\mathcal{C}_{\mathrm{v}}^{+}$ and the simpler simulation $\mathcal{T}$. We give several applications involving deriving optimized continuation-based simulations for call-by-name and call-by-need languages.

1.2 An example

Consider the program $(\lambda x_1.(\lambda x_2.x_1)\,\Omega)\,b$ where Ω represents some term whose evaluation diverges under any evaluation strategy and where b represents some basic constant. Call-by-name evaluation dictates that arguments be passed unevaluated to functions. Thus, call-by-name evaluation of the example program proceeds as follows:

$$\begin{array}{rcl} (\lambda x_1.(\lambda x_2.x_1)\,\Omega)\,b & \longmapsto_{\mathrm{n}} & (\lambda x_2.b)\,\Omega \\ & \longmapsto_{\mathrm{n}} & b \end{array} \tag{1}$$

Call-by-value evaluation dictates that arguments be simplified to values (*i.e.*, constants or abstractions) before being passed to functions. Thus, call-by-value evaluation of the example program proceeds as follows:

$$\begin{array}{rcl} (\lambda x_1.(\lambda x_2.x_1)\,\Omega)\,b & \longmapsto_{\mathrm{v}} & (\lambda x_2.b)\,\Omega \\ & \longmapsto_{\mathrm{v}} & (\lambda x_2.b)\,\Omega' \\ & \longmapsto_{\mathrm{v}} & (\lambda x_2.b)\,\Omega'' \\ & \longmapsto_{\mathrm{v}} & \ldots \end{array}$$

Since the term Ω never reduces to a value, $\lambda x_2.b$ cannot be applied — and the evaluation does not terminate.

[4]In fact, $\mathcal{C}_{\mathrm{n}}$ and $\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T}$ only differ by "administrative reductions" [23, p. 149] (*i.e.*, reductions introduced by the transformations that implement continuation-passing). Thus, for optimizing transformations $\mathcal{C}_{\mathrm{n}.opt}$ and $\mathcal{C}_{\mathrm{v}.opt}^{+}$ that produce CPS terms without administrative reductions [8], the output of $\mathcal{C}_{\mathrm{n}.opt}$ is identical to the output of $\mathcal{C}_{\mathrm{v}.opt}^{+} \circ \mathcal{T}$.

The difference between call-by-name and call-by-value evaluation lies in how arguments are treated. To simulate call-by-name with call-by-value evaluation, one needs a mechanism for turning arbitrary arguments into values. This can be accomplished using a suspension constructor *delay*. *delay* e turns the expression e into a value and thus suspends its evaluation. The suspension destructor *force* triggers the evaluation of an expression suspended by *delay*. Accordingly, suspensions have the following evaluation property.

$$\mathit{force}\;(\mathit{delay}\;e) \longmapsto_{\mathrm{v}} e$$

Introducing *delay* and *force* in the example program *via* a thunking transformation $\mathcal{T}$ provides a simulation of call-by-name under call-by-value evaluation.

$$\begin{array}{ll}
(\lambda x_1.(\lambda x_2.\mathit{force}\;x_1)\,(\mathit{delay}\;\Omega))\,(\mathit{delay}\;b) & \\
\qquad \longmapsto_{\mathrm{v}} & (\lambda x_2.\mathit{force}\;(\mathit{delay}\;b))\,(\mathit{delay}\;\Omega) \\
\qquad \longmapsto_{\mathrm{v}} & \mathit{force}\;(\mathit{delay}\;b) \\
\qquad \longmapsto_{\mathrm{v}} & b
\end{array} \qquad (2)$$

Applying Plotkin's call-by-name continuation-passing transformation $\mathcal{C}_{\mathrm{n}}$ to the example program also gives a simulation of call-by-name under call-by-value evaluation [23].

$$\begin{array}{l}
(\lambda k.(\lambda k.k\;(\lambda x_1.\lambda k.(\lambda k.k\;(\lambda x_2.\lambda k.x_1\;k))\,(\lambda y_1.y_1\,\mathcal{C}_{\mathrm{n}}\langle\!\lbrack\Omega\rbrack\!\rangle\;k)))\\
\quad (\lambda y_2.y_2\;(\lambda k.k\;b)\;k))\\
(\lambda y_3.y_3)
\end{array} \qquad (3)$$

The resulting program is said to be in *continuation-passing style* (CPS). A tedious but straightforward rewriting shows that the call-by-value evaluation of the CPS program above yields b — the result of the original program when evaluated under call-by-name. Even after optimizing the CPS program by performing "administrative reductions" (*i.e.*, reductions of abstractions that implement continuation-passing and do not appear in the original program such as the $\lambda k....$ and $\lambda y_i....$ of line (3)) [23, p. 149],

$$(\lambda x_1.\lambda k.(\lambda x_2.\lambda k.x_1\;k)\,\mathcal{C}_{\mathrm{n}}\langle\!\lbrack\Omega\rbrack\!\rangle\;k)\,(\lambda k.k\;b)\,(\lambda y_3.y_3) \qquad (4)$$

the evaluation is still more involved than for the thunked program.[5]

[5]The original term at line (1) requires 2 evaluation steps. The thunked version at line (2) requires 3 steps. The unoptimized CPS version at line (3) requires 11 steps.

1.3 Overview

The remainder of this section gives necessary background material covering the syntax and semantics of λ-terms and Plotkin's continuation-passing simulations. Section 2 presents the thunk-based simulation $\mathcal{T}$ and associated correctness results. Section 3 presents the factoring of $\mathcal{C}_\mathrm{n}$ *via* thunks and gives several applications. Section 4 recasts the results of the previous sections in a typed setting. Section 5 gives a discussion of related work. Section 6 concludes.

1.4 Syntax and semantics of λ-terms

This section briefly reviews the syntax, equational theories, and operational semantics associated with λ-terms. The notation used is essentially Barendregt's [3]. The presentation of calculi in Section 1.4.3 follows Sabry and Felleisen [31] and the presentation of operational semantics in Section 1.4.4 is adapted from Plotkin [23].

1.4.1 The language Λ

Figure 1 presents the syntax of the language Λ. The language is a pure untyped functional language including constants, identifiers, λ-abstractions (functions), and applications. To simplify substitution, we follow Barendregt's variable convention[6] and work with the quotient of Λ under α-equivalence [3]. We write $e_1 \equiv e_2$ when e_1 and e_2 are α-equivalent.

The notation $FV(e)$ denotes the set of free variables in e and $e_1[x := e_2]$ denotes the result of the capture-free substitution of all free occurences of x in e_1 by e_2. A *context* C is a term with a "hole" $[\cdot]$. The operation of *filling* the context C with a term e yields the term $C[e]$, possibly capturing some free variables of e in the process. *Contexts*$[l]$ denotes the set of contexts from some language l. Closed terms — terms with no free variables — are called *programs*. *Programs*$[l]$ denotes the set of programs from some language l.

The optimized CPS version at line (4) requires 6 steps. As Sabry and Felleisen note [31, p. 302], this last program can be optimized further by unfolding source reductions, eliminating administrative reductions exposed by the unfolding, and then expanding back the source reductions. However, an optimized version of $\mathcal{C}_\mathrm{n}$ capturing these additional steps would be significantly more complicated than $\mathcal{T}$ (making it much harder to reason about its correctness). Moreover, the resulting CPS program would still require more evaluation steps in general than the corresponding program in the image of $\mathcal{T}$.

[6] In terms occurring in definitions and proofs *etc.*, all bound variables are chosen to be different from free variables [3, p. 26].

$$
\begin{array}{rcl}
e & \in & \Lambda \\
e & ::= & b \;\mid\; x \;\mid\; \lambda x.e \;\mid\; e_0\, e_1
\end{array}
$$

Figure 1: Abstract syntax of the language Λ

1.4.2 Values

Certain terms of Λ are designated as *values.* Values roughly correspond to terms that may be results of the operational semantics presented below. The sets $\mathit{Values}_{\mathrm{n}}[\Lambda]$ and $\mathit{Values}_{\mathrm{v}}[\Lambda]$ below represent the set of values from the language Λ under call-by-name and call-by-value evaluation respectively.

$$
\begin{array}{rcl}
v & \in & \mathit{Values}_{\mathrm{n}}[\Lambda] \\
v & ::= & b \mid \lambda x.e
\end{array}
\qquad
\begin{array}{rcl}
v & \in & \mathit{Values}_{\mathrm{v}}[\Lambda] \\
v & ::= & b \mid x \mid \lambda x.e
\end{array}
\qquad \ldots \textit{where } e \in \Lambda
$$

Note that identifiers are included in $\mathit{Values}_{\mathrm{v}}[\Lambda]$ since only values will be substituted for identifiers under call-by-value evaluation. We use v as a meta-variable for values and where no ambiguity results we will ignore the distinction between call-by-name and call-by-value values.

1.4.3 Calculi

λ-calculi are formal theories of equations between λ-terms. We consider calculi generated by one or more of the following principal axiom schemata (also called notions of reduction) along with the logical axioms and inference rules presented below.

Notions of reduction

$$
\begin{array}{rcllr}
(\lambda x.e_1)\, e_2 & \longrightarrow_{\beta} & e_1[x := e_2] & & (\beta) \\
(\lambda x.e)\, v & \longrightarrow_{\beta_{\mathrm{v}}} & e[x := v] & v \in \mathit{Values}_{\mathrm{v}}[\Lambda] & (\beta_{\mathrm{v}}) \\
\lambda x.e\, x & \longrightarrow_{\eta} & e & x \notin FV(e) & (\eta) \\
\lambda x.v\, x & \longrightarrow_{\eta_{\mathrm{v}}} & v & v \in \mathit{Values}_{\mathrm{v}}[\Lambda] \;\wedge\; x \notin FV(v) & (\eta_{\mathrm{v}})
\end{array}
$$

Logical axioms and inference rules

$$
\begin{array}{rcllr}
e_1 \longrightarrow e_2 & \Rightarrow & C[e_1] = C[e_2] & \forall \textit{ contexts } C & \textit{(Compatibility)} \\
 & & e = e & & \textit{(Reflexivity)} \\
e_1 = e_2,\; e_2 = e_3 & \Rightarrow & e_1 = e_3 & & \textit{(Transitivity)} \\
e_1 = e_2 & \Rightarrow & e_2 = e_1 & & \textit{(Symmetry)}
\end{array}
$$

Call-by-name:

$$(\lambda x.e_0)\, e_1 \longmapsto_{\mathrm{n}} e_0[x := e_1] \qquad \frac{e_0 \longmapsto_{\mathrm{n}} e_0'}{e_0\, e_1 \longmapsto_{\mathrm{n}} e_0'\, e_1}$$

Call-by-value:

$$(\lambda x.e)\, v \longmapsto_{\mathrm{v}} e[x := v] \qquad \frac{e_0 \longmapsto_{\mathrm{v}} e_0'}{e_0\, e_1 \longmapsto_{\mathrm{v}} e_0'\, e_1}$$

$$\frac{e_1 \longmapsto_{\mathrm{v}} e_1'}{(\lambda x.e_0)\, e_1 \longmapsto_{\mathrm{v}} (\lambda x.e_0)\, e_1'}$$

Figure 2: Single-step evaluation rules

The underlying notions of reduction completely identify a theory. For example, β generates the theory $\lambda\beta$ and β_{v} generates the theory $\lambda\beta_{\mathrm{v}}$. In general, we write λA to refer to the theory generated by a set of axioms A. When a theory λA proves an equation $e_1 = e_2$, we write $\lambda A \vdash e_1 = e_2$. If the proof does not involve the inference rule *(Symmetry)*, we write $\lambda A \vdash e_1 \longrightarrow\!\!\!\!\rightarrow e_2$, and if the proof only involves the rule *(Compatibility)* we write $\lambda A \vdash e_1 \longrightarrow e_2$. Reductions in calculational style proofs are denoted by subscripting reduction symbols (*e.g.*, $\longrightarrow_{\beta}$, $\longrightarrow\!\!\!\!\rightarrow_{\eta_{\mathrm{v}}}$). If a property holds for both $\lambda\beta$ and $\lambda\beta_{\mathrm{v}}$, we say the property holds for $\lambda\beta_i$.

1.4.4 Operational semantics

Figure 2 presents single-step evaluation rules which define the call-by-name and call-by-value operational semantics of Λ programs.[7] The (partial) evaluation functions $eval_{\mathrm{n}}$ and $eval_{\mathrm{v}}$ are defined in terms of the reflexive, transitive closure (denoted $\longmapsto^*$) of the single-step evaluation rules.

$$\begin{aligned} eval_{\mathrm{n}}(e) &= v \quad \text{iff} \quad e \longmapsto_{\mathrm{n}}^* v \\ eval_{\mathrm{v}}(e) &= v \quad \text{iff} \quad e \longmapsto_{\mathrm{v}}^* v \end{aligned}$$

[7]The rules of Figure 2 are a simplified version of Plotkin's [23, pp. 146 and 136]. To simplify the presentation, we do not consider evaluation rules defined over open terms or functional constants (*i.e.*, δ-rules).

We write $e \longmapsto_i e'$ when both $e \longmapsto_{\mathrm{n}} e'$ and $e \longmapsto_{\mathrm{v}} e'$ (similarly for eval_i). Given meta-language expressions E_1 and E_2 where one or both may be undefined, we write $E_1 \simeq E_2$ when E_1 and E_2 are both undefined, or else both are defined and denote α-equivalent terms. Similarly, for any notion of reduction r, we write $E_1 \simeq_r E_2$ when E_1 and E_2 are both undefined, or else are both defined and denote r-equivalent terms.

An evaluation $\mathit{eval}(e)$ may be undefined for two reasons:

1. e heads an infinite evaluation sequence, *i.e.*, $e \longmapsto e_1 \longmapsto e_2 \longmapsto \ldots$,
2. e heads an evaluation sequence which ends in a *stuck* term — a non-value which cannot be further evaluated (*e.g.*, the application of a basic constant to some argument).

The following definition gives programs that are stuck under call-by-name and call-by-value evaluation.

$$\begin{array}{lcl@{\qquad}lcl}
s & \in & \mathit{Stuck}_{\mathrm{n}}[\Lambda] & s & \in & \mathit{Stuck}_{\mathrm{v}}[\Lambda] \\
s & ::= & b\,e \mid s\,e & s & ::= & b\,e \mid s\,e \mid (\lambda x.e)\,s \quad \ldots\textit{where } e \in \Lambda
\end{array}$$

A simple induction over the structure of $e \in \mathit{Programs}[\Lambda]$ shows that either $e \in \mathit{Values}_{\mathrm{n}}[\Lambda]$, or $e \in \mathit{Stuck}_{\mathrm{n}}[\Lambda]$, or $e \longmapsto_{\mathrm{n}} e'$. A similar property holds for call-by-value.

1.4.5 Operational equivalence

Plotkin's definitions of call-by-name and call-by-value operational equivalence are as follows [23, pp. 147 and 144].

Definition 1 (CBN operational equivalence) *For all $e_1, e_2 \in \Lambda$, $e_1 \approx_{\mathrm{n}} e_2$ iff for any context $C \in \mathit{Contexts}[\Lambda]$ such that $C[e_1]$ and $C[e_2]$ are programs, $\mathit{eval}_{\mathrm{n}}(C[e_1])$ and $\mathit{eval}_{\mathrm{n}}(C[e_2])$ are either both undefined, or else both defined and one is a given basic constant b iff the other is.*

Definition 2 (CBV operational equivalence) *For all $e_1, e_2 \in \Lambda$, $e_1 \approx_{\mathrm{v}} e_2$ iff for any context $C \in \mathit{Contexts}[\Lambda]$ such that $C[e_1]$ and $C[e_2]$ are programs, $\mathit{eval}_{\mathrm{v}}(C[e_1])$ and $\mathit{eval}_{\mathrm{v}}(C[e_2])$ are either both undefined, or else both defined and one is a given basic constant b iff the other is.*

The calculi of Section 1.4.3 can be used to reason about operational behavior. To establish the operational equivalence two terms, it is sufficient to show that the terms are convertible in an appropriate calculus.

Theorem 1 (Soundness of calculi for Λ) *For all* $e_1, e_2 \in \Lambda$,

$$\begin{array}{rcl} \lambda\beta \vdash e_1 = e_2 & \Rightarrow & e_1 \approx_{\mathrm{n}} e_2 \\ \lambda\beta_{\mathrm{v}} \vdash e_1 = e_2 & \Rightarrow & e_1 \approx_{\mathrm{v}} e_2 \end{array}$$

Proof: See [23, pp. 147 and 144] ∎

Note that η is unsound for both call-by-name and call-by-value since it does not preserve termination properties.[8] Termination properties can be preserved by requiring the contractum of an η-redex to be a value. For example, η_{v} preserves call-by-value termination properties. However, even these restricted forms are unsound *in an untyped setting* due to "improper" uses of basic constants. For example,

$$\lambda x.b\,x \longrightarrow_{\eta_{\mathrm{v}}} b$$

but $\lambda x.b\,x \not\approx_v b$ (take $C = [\cdot]$). Thus, extending the setting considered by Plotkin (*i.e.*, untyped terms with basic constants) to include an elegant theory of η-like reduction seems problematic.[9] However, in specific settings where constraints on the structure of terms disallow such problematic cases, limited forms of η reduction can be applied soundly.[10]

1.5 Continuation-based simulations

This section presents Plotkin's continuation-based simulations of call-by-name in call-by-value and vice versa [23]. As characterized by Meyer and Wand [18], "CPS terms are tail-recursive: no argument is an application. Therefore there is at most one redex which is not inside the scope of an abstraction, and thus call-by-value evaluation coincides with outermost or call-by-name evaluation."

1.5.1 Call-by-name continuation-passing style

Figure 3 gives Plotkin's call-by-name CPS transformation $\mathcal{P}_{\mathrm{n}}$ where the k's and the y's are fresh variables (*i.e.*, variables not appearing free in the argument of $\mathcal{P}_{\mathrm{n}}$). The transformation is defined using two translation functions:

[8] For example, $\lambda x.\Omega\, x \longrightarrow_{\eta} \Omega$ but $eval_i(\lambda x.\Omega\, x)$ is defined whereas $eval_i(\Omega)$ is undefined.

[9] Sabry and Felleisen similarly discuss problems with η and η_{v} reduction [30, p. 5] [31, p. 322].

[10] This is the case with the languages of terms in the image of CPS transformations presented in the following section.

$$
\begin{array}{rcl}
\mathcal{P}_{\mathrm{n}}\langle\!|\cdot|\!\rangle & : & \Lambda \to \Lambda \\
\mathcal{P}_{\mathrm{n}}\langle\!|v|\!\rangle & = & \lambda k.k\,\mathcal{P}_{\mathrm{n}}\langle v\rangle \\
\mathcal{P}_{\mathrm{n}}\langle\!|x|\!\rangle & = & x \\
\mathcal{P}_{\mathrm{n}}\langle\!|e_0\,e_1|\!\rangle & = & \lambda k.\mathcal{P}_{\mathrm{n}}\langle\!|e_0|\!\rangle\,(\lambda y_0.y_0\,\mathcal{P}_{\mathrm{n}}\langle\!|e_1|\!\rangle\,k) \\
\\
\mathcal{P}_{\mathrm{n}}\langle\cdot\rangle & : & \mathit{Values}_{\mathrm{n}}[\Lambda] \to \Lambda \\
\mathcal{P}_{\mathrm{n}}\langle b\rangle & = & b \\
\mathcal{P}_{\mathrm{n}}\langle \lambda x.e\rangle & = & \lambda x.\mathcal{P}_{\mathrm{n}}\langle\!|e|\!\rangle
\end{array}
$$

Figure 3: Plotkin's call-by-name CPS transformation

$\mathcal{P}_{\mathrm{n}}\langle\!|\cdot|\!\rangle$ is the general translation function for terms of Λ; $\mathcal{P}_{\mathrm{n}}\langle\cdot\rangle$ is the translation function for call-by-name values. The following theorem given by Plotkin [23, p. 153] captures correctness properties of the transformation.

Theorem 2 (Plotkin 1975) *For all $e \in \mathit{Programs}[\Lambda]$ and $e_1, e_2 \in \Lambda$,*

1. **Indifference:** $\mathit{eval}_{\mathrm{v}}(\mathcal{P}_{\mathrm{n}}\langle\!|e|\!\rangle\, I) \simeq \mathit{eval}_{\mathrm{n}}(\mathcal{P}_{\mathrm{n}}\langle\!|e|\!\rangle\, I)$
2. **Simulation:** $\mathcal{P}_{\mathrm{n}}\langle \mathit{eval}_{\mathrm{n}}(e)\rangle \simeq \mathit{eval}_{\mathrm{v}}(\mathcal{P}_{\mathrm{n}}\langle\!|e|\!\rangle\, I)$
3. **Translation:**

$$
\begin{array}{rll}
\lambda\beta \vdash e_1 = e_2 & \mathit{iff} & \lambda\beta_{\mathrm{v}} \vdash \mathcal{P}_{\mathrm{n}}\langle\!|e_1|\!\rangle = \mathcal{P}_{\mathrm{n}}\langle\!|e_2|\!\rangle \\
& \mathit{iff} & \lambda\beta \vdash \mathcal{P}_{\mathrm{n}}\langle\!|e_1|\!\rangle = \mathcal{P}_{\mathrm{n}}\langle\!|e_2|\!\rangle \\
& \mathit{iff} & \lambda\beta_{\mathrm{v}} \vdash \mathcal{P}_{\mathrm{n}}\langle\!|e_1|\!\rangle\, I = \mathcal{P}_{\mathrm{n}}\langle\!|e_2|\!\rangle\, I \\
& \mathit{iff} & \lambda\beta \vdash \mathcal{P}_{\mathrm{n}}\langle\!|e_1|\!\rangle\, I = \mathcal{P}_{\mathrm{n}}\langle\!|e_2|\!\rangle\, I
\end{array}
$$

The **Indifference** property states that, given the identity function $I = \lambda y.y$ as the initial continuation, the result of evaluating a CPS term using call-by-value evaluation is the same as the result of using call-by-name evaluation. In other words, terms in the image of the transformation are evaluation-order independent. This follows because all function arguments are values in the image of the transformation (and this condition is preserved under β_i reductions).

The **Simulation** property states that, given the identity function as an initial continuation, evaluating a CPS term using call-by-value evaluation simulates the evaluation of the original term using call-by-name evaluation.

The **Translation** property purports that β-equivalence classes are preserved and reflected by $\mathcal{P}_{\rm n}$. However, the property does not hold because[11]

$$\lambda\beta \vdash e_1 = e_2 \;\;\not\Leftrightarrow\;\; \lambda\beta_i \vdash \mathcal{P}_{\rm n}\langle\!\langle e_1\rangle\!\rangle = \mathcal{P}_{\rm n}\langle\!\langle e_2\rangle\!\rangle.$$

In some cases, $\eta_{\rm v}$ is needed to establish the equivalence of the CPS-images of two β-convertible terms. For example, $\lambda x.(\lambda z.z)\,x \longrightarrow_\beta \lambda x.x$ but

$$\begin{array}{lcll}
\mathcal{P}_{\rm n}\langle\!\langle \lambda x.(\lambda z.z)\,x\rangle\!\rangle & = & \lambda k.k\,(\lambda x.\lambda k.(\lambda k.k\,(\lambda z.z))\,(\lambda y.y\,x\,k)) & \\
& \longrightarrow_{\beta_{\rm v}} & \lambda k.k\,(\lambda x.\lambda k.(\lambda y.y\,x\,k)\,(\lambda z.z)) & \\
& \longrightarrow_{\beta_{\rm v}} & \lambda k.k\,(\lambda x.\lambda k.(\lambda z.z)\,x\,k) & \\
& \longrightarrow_{\beta_{\rm v}} & \lambda k.k\,(\lambda x.\lambda k.x\,k) & \\
& \longrightarrow_{\eta_{\rm v}} & \lambda k.k\,(\lambda x.x) & \ldots\eta_{\rm v}\ \textit{is needed for this step} \\
& = & \mathcal{P}_{\rm n}\langle\!\langle \lambda x.x\rangle\!\rangle. &
\end{array}$$

In practice, $\eta_{\rm v}$ reductions such as those required in the example above are unproblematic if they are embedded in proper CPS contexts. When $\lambda k.k\,(\lambda x.\lambda k.x\,k)$ is embedded in a CPS context, x will always bind to a term of the form $\lambda k.e$ during evaluation. However, if the term is not embedded in a CPS context (*e.g.*, $[\cdot]\,(\lambda y.y\,b)$), the $\eta_{\rm v}$ reduction is unsound.

The simplest solution for recovering the **Translation** property is to change the translation of identifiers from $\mathcal{P}_{\rm n}\langle\!\langle x\rangle\!\rangle = x$ to $\lambda k.x\,k$. Let $\mathcal{C}_{\rm n}$ be the modified translation which is identical to $\mathcal{P}_{\rm n}$ except that

$$\mathcal{C}_{\rm n}\langle\!\langle x\rangle\!\rangle \;=\; \lambda k.x\,k$$

For the example above, the new translation gives

$$\lambda\beta_i \vdash \mathcal{C}_{\rm n}\langle\!\langle \lambda x.(\lambda z.z)\,x\rangle\!\rangle = \mathcal{C}_{\rm n}\langle\!\langle \lambda x.x\rangle\!\rangle.$$

The following theorem gives the correctness properties for $\mathcal{C}_{\rm n}$.

Theorem 3 *For all $e \in Programs[\Lambda]$ and $e_1, e_2 \in \Lambda$,*

1. **Indifference:** $eval_{\rm v}(\mathcal{C}_{\rm n}\langle\!\langle e\rangle\!\rangle\,I) \simeq eval_{\rm n}(\mathcal{C}_{\rm n}\langle\!\langle e\rangle\!\rangle\,I)$
2. **Simulation:** $\mathcal{C}_{\rm n}\langle eval_{\rm n}(e)\rangle \simeq_{\beta_i} eval_{\rm v}(\mathcal{C}_{\rm n}\langle\!\langle e\rangle\!\rangle\,I)$

[11] The proof given in [23, p. 158] breaks down where it is stated "It is straightforward to show that $\lambda\beta \vdash e_1 = e_2$ implies $\lambda\beta_{\rm v} \vdash \mathcal{P}_{\rm n}\langle\!\langle e_1\rangle\!\rangle = \mathcal{P}_{\rm n}\langle\!\langle e_2\rangle\!\rangle$ ...".

3. **Translation:**

$$\begin{aligned} \lambda\beta \vdash e_1 = e_2 \quad & \textit{iff} \quad \lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{n}}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{C}_{\mathrm{n}}\langle\!\langle e_2 \rangle\!\rangle \\ & \textit{iff} \quad \lambda\beta \vdash \mathcal{C}_{\mathrm{n}}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{C}_{\mathrm{n}}\langle\!\langle e_2 \rangle\!\rangle \\ & \textit{iff} \quad \lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{n}}\langle\!\langle e_1 \rangle\!\rangle\, I = \mathcal{C}_{\mathrm{n}}\langle\!\langle e_2 \rangle\!\rangle\, I \\ & \textit{iff} \quad \lambda\beta \vdash \mathcal{C}_{\mathrm{n}}\langle\!\langle e_1 \rangle\!\rangle\, I = \mathcal{C}_{\mathrm{n}}\langle\!\langle e_2 \rangle\!\rangle\, I \end{aligned}$$

The **Indifference** and **Translation** properties for $\mathcal{C}_{\mathrm{n}}$ are identical to those of $\mathcal{P}_{\mathrm{n}}$. However, the **Simulation** property for $\mathcal{C}_{\mathrm{n}}$ holds up to β_i-equivalence[12] while **Simulation** for $\mathcal{P}_{\mathrm{n}}$ holds up to α-equivalence.[13]

We show in Section 3.3.1 that proofs of **Indifference**, **Simulation**, and most of the **Translation** can be derived from the correctness properties of $\mathcal{C}_{\mathrm{v}}^{+}$ and $\mathcal{T}$ (as discussed in Section 1). All that remains of **Translation** is the $\Leftarrow$ direction of the first bi-implication; this follows in a straightforward manner from Plotkin's original proof for $\mathcal{P}_{\mathrm{n}}$ (see Appendix A.1.3).

1.5.2 Call-by-value continuation-passing style

Figure 4 gives Plotkin's call-by-value CPS transformation. The following theorem captures correctness properties of the translation.

Theorem 4 (Plotkin 1975) *For all $e \in \mathit{Programs}[\Lambda]$ and $e_1, e_2 \in \Lambda$,*

1. **Indifference:** $\mathit{eval}_{\mathrm{n}}(\mathcal{C}_{\mathrm{v}}\langle\!\langle e \rangle\!\rangle\, I) \simeq \mathit{eval}_{\mathrm{v}}(\mathcal{C}_{\mathrm{v}}\langle\!\langle e \rangle\!\rangle\, I)$

2. **Simulation:** $\mathcal{C}_{\mathrm{v}}\langle \mathit{eval}_{\mathrm{v}}(e) \rangle \simeq \mathit{eval}_{\mathrm{n}}(\mathcal{C}_{\mathrm{v}}\langle\!\langle e \rangle\!\rangle\, I)$

3. **Translation:**

$$\begin{aligned} & \textit{If}\ \lambda\beta_{\mathrm{v}} \vdash e_1 = e_2 \ \textit{then}\ \lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{v}}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\langle e_2 \rangle\!\rangle \\ & \textit{Also}\ \lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{v}}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\langle e_2 \rangle\!\rangle \ \textit{iff}\ \lambda\beta \vdash \mathcal{C}_{\mathrm{v}}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\langle e_2 \rangle\!\rangle \end{aligned}$$

---

[12] For example, $\mathcal{C}_{\mathrm{n}}\langle \mathit{eval}_{\mathrm{n}}((\lambda z.\lambda y.z)\, b) \rangle = \lambda y.\lambda k.k\, b$ whereas $\mathit{eval}_{\mathrm{v}}(\mathcal{C}_{\mathrm{n}}\langle\!\langle (\lambda z.\lambda y.z)\, b \rangle\!\rangle\, I) = \lambda y.\lambda k.(\lambda k.k\, b)\, k$.

[13] This is because $\mathcal{P}_{\mathrm{n}}$ commutes with substitution up to α-equivalence, *i.e.*, $\mathcal{C}_{\mathrm{n}}\langle\!\langle e_0[x := e_1] \rangle\!\rangle \equiv \mathcal{C}_{\mathrm{n}}\langle\!\langle e_0 \rangle\!\rangle[x := \mathcal{C}_{\mathrm{n}}\langle\!\langle e_1 \rangle\!\rangle]$ whereas $\mathcal{C}_{\mathrm{n}}$ commutes with substitution only up to β_i-equivalence, *i.e.*, $\mathcal{C}_{\mathrm{n}}\langle\!\langle e_0[x := e_1] \rangle\!\rangle =_{\beta_i} \mathcal{C}_{\mathrm{n}}\langle\!\langle e_0 \rangle\!\rangle[x := \mathcal{C}_{\mathrm{n}}\langle\!\langle e_1 \rangle\!\rangle]$. This renders the usual *colon translation* technique [23, p. 154] insufficient for proving **Simulation** for $\mathcal{C}_{\mathrm{n}}$. Evaluation steps involving substitution lead to terms which lie outside the image of the colon translation associated with $\mathcal{C}_{\mathrm{n}}$. A similar situation occurs with the thunk-based simulation $\mathcal{T}$ introduced in Section 2 (see Section 2.3.2, Footnote 17).

$$
\begin{array}{rcl}
\mathcal{C}_{\mathrm{v}}\langle\!\langle\cdot\rangle\!\rangle & : & \Lambda \to \Lambda \\
\mathcal{C}_{\mathrm{v}}\langle\!\langle v\rangle\!\rangle & = & \lambda k.k\, \mathcal{C}_{\mathrm{v}}\langle v\rangle \\
\mathcal{C}_{\mathrm{v}}\langle\!\langle e_0\, e_1\rangle\!\rangle & = & \lambda k.\mathcal{C}_{\mathrm{v}}\langle\!\langle e_0\rangle\!\rangle\, (\lambda y_0.\mathcal{C}_{\mathrm{v}}\langle\!\langle e_1\rangle\!\rangle\, (\lambda y_1.y_0\, y_1\, k)) \\
\\
\mathcal{C}_{\mathrm{v}}\langle\cdot\rangle & : & \mathit{Values}_{\mathrm{v}}[\Lambda] \to \Lambda \\
\mathcal{C}_{\mathrm{v}}\langle b\rangle & = & b \\
\mathcal{C}_{\mathrm{v}}\langle x\rangle & = & x \\
\mathcal{C}_{\mathrm{v}}\langle \lambda x.e\rangle & = & \lambda x.\mathcal{C}_{\mathrm{v}}\langle\!\langle e\rangle\!\rangle
\end{array}
$$

Figure 4: Plotkin's call-by-value CPS transformation

The intuition behind the **Indifference** and **Simulation** properties is the same as for $\mathcal{C}_{\mathrm{n}}$. The **Translation** property states that β_{v}-convertible terms are also convertible in the image of $\mathcal{C}_{\mathrm{v}}$. In contrast to the theory $\lambda\beta$ appearing in the **Translation** property for $\mathcal{C}_{\mathrm{n}}$ (Theorem 3), the theory $\lambda\beta_{\mathrm{v}}$ is *incomplete* in the sense that it cannot establish the convertibility of some pairs of terms in the image of the CPS transformation [31].[14]

Finally, note that neither $\mathcal{C}_{\mathrm{n}}$ nor $\mathcal{C}_{\mathrm{v}}$ (nor $\mathcal{P}_{\mathrm{n}}$) are fully abstract (*i.e.*, they do not preserve operational equivalence) [23, pp. 154 and 148]. Specifically, $e_1 \approx_{\mathrm{n}} e_2$ does not imply $\mathcal{C}_{\mathrm{n}}\langle\!\langle e_1\rangle\!\rangle \approx_{\mathrm{v}} \mathcal{C}_{\mathrm{n}}\langle\!\langle e_2\rangle\!\rangle$ (and similarly for $\mathcal{C}_{\mathrm{v}}$).[15]

[14]Plotkin gives the following example of the incompleteness [23, p. 153]. Let $e_1 \equiv ((\lambda x.x\,x)\,(\lambda x.x\,x))\,y$ and $e_2 \equiv (\lambda x.x\,y)\,((\lambda x.x\,x)\,(\lambda x.x\,x))$. Then $\lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{v}}\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\langle e_2\rangle\!\rangle$ and $\lambda\beta \vdash \mathcal{C}_{\mathrm{v}}\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\langle e_2\rangle\!\rangle$ but $\lambda\beta_{\mathrm{v}} \not\vdash e_1 = e_2$. Sabry and Felleisen [31] give an equational theory λA (where A is a set of axioms including $\beta_{\mathrm{v}}\eta_{\mathrm{v}}$) and show it complete in the sense that $\lambda A \vdash e_1 = e_2$ *iff* $\lambda\beta\eta \vdash \mathcal{F}\langle\!\langle e_1\rangle\!\rangle = \mathcal{F}\langle\!\langle e_2\rangle\!\rangle$. $\mathcal{F}$ is Fischer's call-by-value CPS transformation [12] where continuations are the first arguments to functions (instead of the second arguments as in Plotkin's $\mathcal{C}_{\mathrm{v}}$). Note that their results cannot be immediately carried over to this setting since the reduction properties of terms generated by the transformation $\mathcal{F}$ are sufficiently different from the reduction properties of terms generated by Plotkin's transformation $\mathcal{C}_{\mathrm{v}}$ (see [31, p. 314]).

[15]For examples of why full abstraction fails, see [23, pp. 154 and 149] and [30, p. 30]. For a detailed presentation of fully abstract translations in a typed setting, see the work of Riecke [28, 29].

$$
\begin{array}{rcl}
\mathcal{T} & : & \Lambda \to \Lambda_\tau \\
\mathcal{T}\langle\!\langle b \rangle\!\rangle & = & b \\
\mathcal{T}\langle\!\langle x \rangle\!\rangle & = & \mathit{force}\ x \\
\mathcal{T}\langle\!\langle \lambda x.e \rangle\!\rangle & = & \lambda x.\mathcal{T}\langle\!\langle e \rangle\!\rangle \\
\mathcal{T}\langle\!\langle e_0\, e_1 \rangle\!\rangle & = & \mathcal{T}\langle\!\langle e_0 \rangle\!\rangle\, (\mathit{delay}\ \mathcal{T}\langle\!\langle e_1 \rangle\!\rangle)
\end{array}
$$

Figure 5: Thunk introduction

2 Thunks

2.1 Thunk introduction

To establish the simulation properties of thunks, we extend the language Λ to the language Λ_τ that includes suspension operators.

$$
\begin{array}{rcl}
e & \in & \Lambda_\tau \\
e & ::= & \ldots \quad | \quad \mathit{delay}\ e \quad | \quad \mathit{force}\ e
\end{array}
$$

The operator *delay* suspends the computation of an expression — thereby coercing an expression to a value. Therefore, *delay* e is added to the value sets in Λ_τ.

$$
\begin{array}{rcl}
v & \in & \mathit{Values}_{\mathrm{n}}[\Lambda_\tau] \\
v & ::= & \ldots \quad | \quad \mathit{delay}\ e
\end{array}
\qquad
\begin{array}{rcl}
v & \in & \mathit{Values}_{\mathrm{v}}[\Lambda_\tau] \\
v & ::= & \ldots \quad | \quad \mathit{delay}\ e
\end{array}
\quad \ldots\mathit{where}\ e \in \Lambda_\tau
$$

Figure 5 presents the definition of the thunk-based simulation $\mathcal{T}$.

2.2 Reduction of thunked terms

2.2.1 τ-reduction

The operator *force* triggers the evaluation of a suspension created by *delay*. This is formalized by the following notion of reduction.

Definition 3 (τ-reduction) $\mathit{force}\ (\mathit{delay}\ e) \longrightarrow_\tau e$

The notion of reduction τ generates the theory $\lambda\tau$ as outlined in Section 1.4.3. Combining reductions β and τ generates the theory $\lambda\beta\tau$. Similarly, β_{v} and τ give $\lambda\beta_{\mathrm{v}}\tau$.

It is easy to show that τ is Church-Rosser. The Church-Rosser property for $\beta\tau$ and $\beta_v\tau$ follows by the Hindley-Rosen Lemma [3, pp. 53 – 65] since β [3, p. 62] and β_v [23, p. 135] are also Church-Rosser, and it can be shown that $\longrightarrow\!\!\!\!\!\rightarrow_\tau$ commutes with $\longrightarrow\!\!\!\!\!\rightarrow_\beta$ and $\longrightarrow\!\!\!\!\!\rightarrow_{\beta_v}$.

The evaluation rules for Λ_τ are obtained by adding the following rules to both the call-by-name and call-by-value evaluation rules of Figure 2.

$$\frac{e \longmapsto e'}{\mathit{force}\ e \longmapsto \mathit{force}\ e'} \qquad \mathit{force}\ (\mathit{delay}\ e) \longmapsto e$$

2.2.2 A language closed under reductions

To determine the correctness properties of thunks, we consider the set of terms $T \subset \Lambda_\tau$ which are reachable from the image of $\mathcal{T}$ *via* β and τ reductions.

$$T \stackrel{\text{def}}{=} \{t \in \Lambda_\tau \mid \exists e \in \Lambda\,.\,\lambda\beta\tau \vdash \mathcal{T}\langle\!\![e]\!\!\rangle \longrightarrow\!\!\!\!\!\rightarrow t\}$$

The set of terms T can be described with the following grammar.

$$\begin{array}{lcl} t & \in & \mathcal{T}\langle\!\![\Lambda]\!\!\rangle^* \\ t & ::= & b \mid \mathit{force}\ x \mid \mathit{force}\ (\mathit{delay}\ t) \mid \lambda x.t \mid t_0\ (\mathit{delay}\ t_1) \end{array}$$

Appendix A.2.2 shows that the language $\mathcal{T}\langle\!\![\Lambda]\!\!\rangle^* = T$. Note that every β-redex in $\mathcal{T}\langle\!\![\Lambda]\!\!\rangle^*$ is also a β_v-redex (since all function arguments are suspensions).

2.3 A thunk-based simulation

We want to show that thunks are sufficient for establishing a call-by-name simulation satisfying all of the correctness properties of the continuation-passing simulation $\mathcal{C}_n$. Specifically, we prove the following theorem which recasts the correctness theorem for $\mathcal{C}_n$ (Theorem 3) in terms of $\mathcal{T}$.[16]

Theorem 5 *For all* $e \in \mathit{Programs}[\Lambda]$ *and* $e_1, e_2 \in \Lambda$*,*

1. **Indifference:** $\mathit{eval}_v(\mathcal{T}\langle\!\![e]\!\!\rangle) \simeq \mathit{eval}_n(\mathcal{T}\langle\!\![e]\!\!\rangle)$
2. **Simulation:** $\mathcal{T}\langle\!\![\mathit{eval}_n(e)]\!\!\rangle \simeq_\tau \mathit{eval}_v(\mathcal{T}\langle\!\![e]\!\!\rangle)$

[16]The last two assertions of the **Translation** component of Theorem 3 do not appear here since the identity function as the initial continuation only plays a role in CPS evaluation.

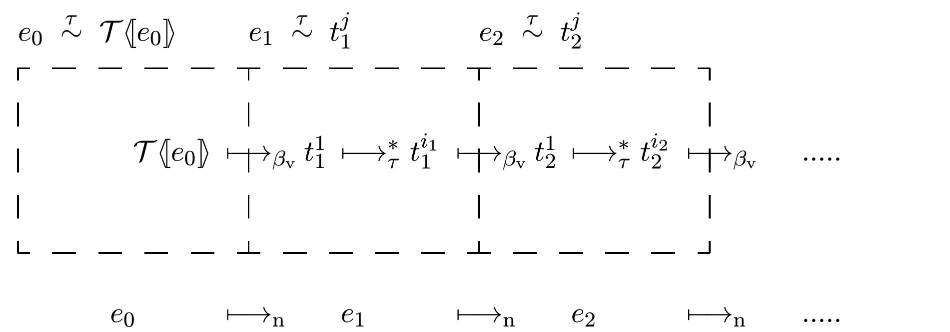

Figure 6: Evaluation of thunked terms

3. **Translation:** $\lambda\beta \vdash e_1 = e_2$ *iff* $\lambda\beta_\mathrm{v}\tau \vdash \mathcal{T}\langle\!|e_1|\!\rangle = \mathcal{T}\langle\!|e_2|\!\rangle$ *iff* $\lambda\beta\tau \vdash \mathcal{T}\langle\!|e_1|\!\rangle = \mathcal{T}\langle\!|e_2|\!\rangle$

2.3.1 Indifference

The **Indifference** property for $\mathcal{T}$ is immediate since all function arguments are values (specifically *suspensions*) in the language $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$.

2.3.2 Simulation

In general, the steps involved in $\mathcal{T}\langle\!|eval_\mathrm{n}(e)|\!\rangle$ and $eval_\mathrm{v}(\mathcal{T}\langle\!|e|\!\rangle)$ can be pictured as in Figure 6 (in the figure, $\longmapsto_\tau$ and $\longmapsto_{\beta_\mathrm{v}}$ denote $\longmapsto_\mathrm{v}$ steps which correspond to τ and β_v reduction, respectively).[17] Initially, $\mathcal{T}\langle\!|e_0|\!\rangle \longmapsto_{\beta_\mathrm{v}} t_1^1$ where t_1^1 is related to e_1 by the following inductively defined relation $\stackrel{\tau}{\sim}$.

$$\stackrel{\tau}{\sim}.1 \quad b \stackrel{\tau}{\sim} b$$

$$\stackrel{\tau}{\sim}.2 \quad x \stackrel{\tau}{\sim} force\ x$$

$$\stackrel{\tau}{\sim}.3 \quad \frac{e \stackrel{\tau}{\sim} t}{\lambda x.e \stackrel{\tau}{\sim} \lambda x.t}$$

$$\stackrel{\tau}{\sim}.4 \quad \frac{e_0 \stackrel{\tau}{\sim} t_0 \qquad e_1 \stackrel{\tau}{\sim} t_1}{e_0\, e_1 \stackrel{\tau}{\sim} t_0\,(delay\ t_1)}$$

$$\stackrel{\tau}{\sim}.5 \quad \frac{e \stackrel{\tau}{\sim} t}{e \stackrel{\tau}{\sim} force\ (delay\ t)}$$

[17] Note that **Simulation** for $\mathcal{T}$ holds up to τ-equivalence because $\mathcal{T}$ commutes with substitution up to τ-equivalence. Taking $e = (\lambda x.\lambda y.x)\, b$ illustrates that $\mathcal{T}\langle\!|eval_\mathrm{n}(e)|\!\rangle$ may be in τ-normal form where $eval_\mathrm{v}(\mathcal{T}\langle\!|e|\!\rangle)$ may contain τ-redexes inside the body of a resulting abstraction.

$$
\begin{array}{rcl}
\mathcal{T}^{-1} & : & \mathcal{T}\langle\!\![\Lambda]\!\!\rangle^* \to \Lambda \\
\mathcal{T}^{-1}\langle\!\![b]\!\!\rangle & = & b \\
\mathcal{T}^{-1}\langle\!\![\mathit{force}\ x]\!\!\rangle & = & x \\
\mathcal{T}^{-1}\langle\!\![\mathit{force}\ (\mathit{delay}\ t)]\!\!\rangle & = & \mathcal{T}^{-1}\langle\!\![t]\!\!\rangle \\
\mathcal{T}^{-1}\langle\!\![\lambda x.t]\!\!\rangle & = & \lambda x.\mathcal{T}^{-1}\langle\!\![t]\!\!\rangle \\
\mathcal{T}^{-1}\langle\!\![t_0\,(\mathit{delay}\ t_1)]\!\!\rangle & = & \mathcal{T}^{-1}\langle\!\![e_0]\!\!\rangle\,\mathcal{T}^{-1}\langle\!\![e_1]\!\!\rangle
\end{array}
$$

Figure 7: Thunk elimination

Simple inductions show that $e \overset{\mathcal{T}}{\sim} \mathcal{T}\langle\!\![e]\!\!\rangle$, and that $e \overset{\mathcal{T}}{\sim} t$ implies $\mathcal{T}\langle\!\![e]\!\!\rangle$ is τ-equivalent to t.

Now for the remaining steps in Figure 6, the following property states that each $\longmapsto_{\mathrm{n}}$ step on a Λ term implies corresponding $\longmapsto_{\mathrm{v}}$ steps on appropriately related thunked terms.

Property 1 *For all $e_0, e_1 \in Programs[\Lambda]$ and $t_0 \in Programs[\mathcal{T}\langle\!\![\Lambda]\!\!\rangle^*]$ such that $e_0 \overset{\mathcal{T}}{\sim} t_0$,*

$$e_0 \longmapsto_{\mathrm{n}} e_1 \ \Rightarrow\ \exists t_1 \in \mathcal{T}\langle\!\![\Lambda]\!\!\rangle^* .\ t_0 \longmapsto_{\mathrm{v}}^{+} t_1 \ \wedge\ e_1 \overset{\mathcal{T}}{\sim} t_1$$

It is also the case that every terminating evaluation sequence over Λ terms corresponds to a terminating evaluation sequence over thunked terms (and vice-versa). These properties are sufficient for establishing the **Simulation** property for $\mathcal{T}$ (see Appendix A.2.3).

2.3.3 Translation

To prove the **Translation** for $\mathcal{T}$, we establish an equational correspondence between the language Λ under theory $\lambda\beta$ and language $\mathcal{T}\langle\!\![\Lambda]\!\!\rangle^*$ under theory $\lambda\beta_i\tau$ (*i.e.*, $\lambda\beta_{\mathrm{v}}\tau$ as well as $\lambda\beta\tau$). Basically, equational correspondence holds when a one-to-one correspondence exists between equivalence classes of the two theories.

The thunk introduction $\mathcal{T}$ of Figure 5 establishes a mapping from Λ to $\mathcal{T}\langle\!\![\Lambda]\!\!\rangle^*$. For the reverse direction, the thunk elimination $\mathcal{T}^{-1}$ of Figure 7 establishes a mapping from $\mathcal{T}\langle\!\![\Lambda]\!\!\rangle^*$ back to Λ.

The relationship between equational theories for source terms and thunked terms is as follows.

Theorem 6 (Equational Correspondence) *For all* $e, e_1, e_2 \in \Lambda$ *and* $t, t_1, t_2 \in \mathcal{T}\langle\!\lbrack\Lambda\rbrack\!\rangle^*$,

1. $\lambda\beta \vdash e = (\mathcal{T}^{-1} \circ \mathcal{T})\langle\!\lbrack e\rbrack\!\rangle$
2. $\lambda\beta_i\tau \vdash t = (\mathcal{T} \circ \mathcal{T}^{-1})\langle\!\lbrack t\rbrack\!\rangle$
3. $\lambda\beta \vdash e_1 = e_2$ *iff* $\lambda\beta_i\tau \vdash \mathcal{T}\langle\!\lbrack e_1\rbrack\!\rangle = \mathcal{T}\langle\!\lbrack e_2\rbrack\!\rangle$
4. $\lambda\beta_i\tau \vdash t_1 = t_2$ *iff* $\lambda\beta \vdash \mathcal{T}^{-1}\langle\!\lbrack t_1\rbrack\!\rangle = \mathcal{T}^{-1}\langle\!\lbrack t_2\rbrack\!\rangle$

Note that component 3 of Theorem 6 corresponds to the thunk **Translation** property (component 3 of Theorem 5).

The proof of Theorem 6 follows the outline of a proof with similar structure given by Sabry and Felleisen [31]. First, we characterize the interaction of $\mathcal{T}$ and $\mathcal{T}^{-1}$ (components 1 and 2 of Theorem 6). Then, we examine the relation between reductions in the theories $\lambda\beta$ and $\lambda\beta_i\tau$ (components 3 and 4 of Theorem 6).

The following property states that $\mathcal{T}^{-1} \circ \mathcal{T}$ is the identity function over Λ.

Property 2 *For all* $e \in \Lambda$, $e = (\mathcal{T}^{-1} \circ \mathcal{T})\langle\!\lbrack e\rbrack\!\rangle$.

This follows from the fact that $\mathcal{T}^{-1}$ simply removes all suspension operators. However, removing suspension operators has the effect of collapsing τ-redexes. This leads to a slightly weaker condition for the opposite direction.

Property 3 *For all* $t \in \mathcal{T}\langle\!\lbrack\Lambda\rbrack\!\rangle^*$, $\lambda\tau \vdash t = (\mathcal{T} \circ \mathcal{T}^{-1})\langle\!\lbrack t\rbrack\!\rangle$.

In other words, $\mathcal{T} \circ \mathcal{T}^{-1}$ is not the identity function, but maintains τ-equivalence. For example,

$$\begin{array}{rcl}(\mathcal{T} \circ \mathcal{T}^{-1})\langle\!\lbrack(\lambda x.\mathit{force}\ (\mathit{delay}\ b))\ (\mathit{delay}\ b)\rbrack\!\rangle & = & \mathcal{T}\langle\!\lbrack(\lambda x.b)\ b\rbrack\!\rangle \\ & = & (\lambda x.b)\ (\mathit{delay}\ b).\end{array}$$

Components 1 and 2 of Theorem 6 follow immediately from Properties 2 and 3.

For components 3 and 4 of Theorem 6, it is sufficient to establish the following two properties. The first property shows that any reduction in Λ corresponds to *one or more* reductions in $\mathcal{T}\langle\!\lbrack\Lambda\rbrack\!\rangle^*$.

Property 4 *For all $e_1, e_2 \in \Lambda$,*

$$\lambda\beta \vdash e_1 \longrightarrow e_2 \;\Rightarrow\; \lambda\beta_i\tau \vdash \mathcal{T}\langle\!\!\{ e_1 \}\!\!\rangle \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}\langle\!\!\{ e_2 \}\!\!\rangle.$$

For example, the β-reduction

$$\lambda\beta \vdash e_1 \equiv (\lambda x.x\, b)\,(\lambda y.y) \longrightarrow (\lambda y.y)\, b \equiv e_2$$

corresponds to the β_i-reduction

$$\begin{aligned}\lambda\beta_i\tau \vdash \mathcal{T}\langle\!\!\{ e_1 \}\!\!\rangle &\equiv (\lambda x.(\mathit{force}\ x)\,(\mathit{delay}\ b))\,(\mathit{delay}\ (\lambda y.\mathit{force}\ y))\\ &\longrightarrow (\mathit{force}\ (\mathit{delay}\ (\lambda y.\mathit{force}\ y)))\,(\mathit{delay}\ b)\end{aligned}$$

However, an additional τ-reduction (and in general multiple τ-reductions) is needed to reach $\mathcal{T}\langle\!\!\{ e_2 \}\!\!\rangle$, *i.e.*,

$$\begin{aligned}\lambda\beta_i\tau \vdash &(\mathit{force}\ (\mathit{delay}\ (\lambda y.\mathit{force}\ y)))\,(\mathit{delay}\ b)\ .\\ &\longrightarrow (\lambda y.\mathit{force}\ y)\,(\mathit{delay}\ b) \equiv \mathcal{T}\langle\!\!\{ e_2 \}\!\!\rangle\end{aligned}$$

For the other direction, the following property states that any reduction in $\mathcal{T}\langle\!\!\{ \Lambda \}\!\!\rangle^*$ corresponds to *zero or one* reductions in Λ.

Property 5 *For all $t_1, t_2 \in \mathcal{T}\langle\!\!\{ \Lambda \}\!\!\rangle^*$,*

$$\lambda\beta_i\tau \vdash t_1 \longrightarrow t_2 \;\Rightarrow\; \lambda\beta \vdash \mathcal{T}^{-1}\langle\!\!\{ t_1 \}\!\!\rangle \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}^{-1}\langle\!\!\{ t_2 \}\!\!\rangle.$$

Specifically, a τ-reduction in $\mathcal{T}\langle\!\!\{ \Lambda \}\!\!\rangle^*$ implies no reductions in Λ. This is because $\mathcal{T}^{-1}$ collapses τ-redexes. For example,

$$\lambda\beta_i\tau \vdash t_1 \equiv \mathit{force}\ (\mathit{delay}\ b) \longrightarrow b \equiv t_2,$$

but $\mathcal{T}^{-1}\langle\!\!\{ t_1 \}\!\!\rangle = b = \mathcal{T}^{-1}\langle\!\!\{ t_2 \}\!\!\rangle$, so no reductions occur.

A β_i-reduction in $\mathcal{T}\langle\!\!\{ \Lambda \}\!\!\rangle^*$ implies one β-reduction in Λ. For example, the β_i-reduction

$$\begin{aligned}\lambda\beta_i\tau \vdash t_1 &\equiv (\lambda x.(\mathit{force}\ x)\,(\mathit{delay}\ b))\,(\mathit{delay}\ (\lambda y.\mathit{force}\ y))\\ &\longrightarrow (\mathit{force}\ (\mathit{delay}\ \lambda y.\mathit{force}\ y))\,(\mathit{delay}\ b) \equiv t_2\end{aligned}$$

corresponds to the β-reduction

$$\lambda\beta \vdash \mathcal{T}^{-1}\langle\!\!\{ t_1 \}\!\!\rangle \equiv (\lambda x.x\, b)\,(\lambda y.y) \longrightarrow (\lambda y.y)\, b \equiv \mathcal{T}^{-1}\langle\!\!\{ t_2 \}\!\!\rangle.$$

Given these properties, components 3 and 4 of Theorem 6 can be proved in a straightforward manner by appealing to Church-Rosser and compatibility properties of β and $\beta_i\tau$ reduction (see Appendix A.2.4).

2.4 Thunks implemented in Λ

Representing thunks *via* abstract suspension operators *delay* and *force* simplifies the technical presentation and enables the connection between $\mathcal{C}_{\rm n}$ and $\mathcal{C}_{\rm v}$ presented in the next section. Elsewhere [15], we show that the *delay*/*force* representation of thunks and associated properties (*i.e.*, reduction properties and translation into CPS) are not arbitrary, but are determined by the relationship between strictness and continuation monads [19].

However, thunks can be implemented directly in Λ using what Plotkin described as the "protecting by a λ" technique [23, p. 147]. Specifically, an expression is delayed by wrapping it in an abstraction with a dummy parameter. A suspension is forced by applying it to a dummy argument. The following transformation encodes Λ_τ terms using this technique (we only show the transformation on suspension operators).

$$\begin{array}{rcll}\mathcal{L} & : & \Lambda_\tau \to \Lambda & \\ & \dots & & \\ \mathcal{L}\langle\!\!\langle \mathit{delay}\ e\rangle\!\!\rangle & = & \lambda z.\mathcal{L}\langle\!\!\langle e\rangle\!\!\rangle & \quad \dots \mathit{where}\ z \notin FV(e) \\ \mathcal{L}\langle\!\!\langle \mathit{force}\ e\rangle\!\!\rangle & = & e\, b & \end{array}$$

This implementation of *delay* and *force* preserves the two basic properties of suspensions:

1. $\mathcal{L}\langle\!\!\langle \mathit{delay}\ e\rangle\!\!\rangle \;=\; \lambda z.\mathcal{L}\langle\!\!\langle e\rangle\!\!\rangle$ is a value; and
2. τ-reduction is faithfully implemented in both the call-by-name and call-by-value calculi, *i.e.*,

$$\mathcal{L}\langle\!\!\langle \mathit{force}\ (\mathit{delay}\ e)\rangle\!\!\rangle \;=\; (\lambda z.\mathcal{L}\langle\!\!\langle e\rangle\!\!\rangle)\, b \;\longrightarrow_{\beta_i}\; \mathcal{L}\langle\!\!\langle e\rangle\!\!\rangle.$$

Now, by composing $\mathcal{L}$ with $\mathcal{T}$ we obtain the thunk-introducing transformation $\mathcal{T}_{\mathcal{L}}$ of Figure 8 that implements thunks directly in Λ. The following theorem recasts the correctness theorem for $\mathcal{C}_{\rm n}$ (Theorem 3) in terms of $\mathcal{T}_{\mathcal{L}}$.

Theorem 7 *For all $e \in \mathit{Programs}[\Lambda]$ and $e_1, e_2 \in \Lambda$,*

1. **Indifference:** $\mathit{eval}_{\rm v}(\mathcal{T}_{\mathcal{L}}\langle\!\!\langle e\rangle\!\!\rangle) \simeq \mathit{eval}_{\rm n}(\mathcal{T}_{\mathcal{L}}\langle\!\!\langle e\rangle\!\!\rangle)$
2. **Simulation:** $\mathcal{T}_{\mathcal{L}}\langle\!\!\langle \mathit{eval}_{\rm n}(e)\rangle\!\!\rangle \simeq_{\beta_i} \mathit{eval}_{\rm v}(\mathcal{T}_{\mathcal{L}}\langle\!\!\langle e\rangle\!\!\rangle)$
3. **Translation:** $\lambda\beta \vdash e_1 = e_2$ *iff* $\lambda\beta_{\rm v} \vdash \mathcal{T}_{\mathcal{L}}\langle\!\!\langle e_1\rangle\!\!\rangle = \mathcal{T}_{\mathcal{L}}\langle\!\!\langle e_2\rangle\!\!\rangle$ *iff* $\lambda\beta \vdash \mathcal{T}_{\mathcal{L}}\langle\!\!\langle e_1\rangle\!\!\rangle = \mathcal{T}_{\mathcal{L}}\langle\!\!\langle e_2\rangle\!\!\rangle$

$$
\begin{array}{rcll}
\mathcal{T}_{\mathcal{L}} & : & \Lambda \rightarrow \Lambda & \\
\mathcal{T}_{\mathcal{L}}\langle\!\langle b \rangle\!\rangle & = & b & \\
\mathcal{T}_{\mathcal{L}}\langle\!\langle x \rangle\!\rangle & = & x\, b & \textit{...for some arbitrary basic constant } b \\
\mathcal{T}_{\mathcal{L}}\langle\!\langle \lambda x.e \rangle\!\rangle & = & \lambda x.\mathcal{T}_{\mathcal{L}}\langle\!\langle e \rangle\!\rangle & \\
\mathcal{T}_{\mathcal{L}}\langle\!\langle e_0\, e_1 \rangle\!\rangle & = & \mathcal{T}_{\mathcal{L}}\langle\!\langle e_0 \rangle\!\rangle\, (\lambda z.\mathcal{T}_{\mathcal{L}}\langle\!\langle e_1 \rangle\!\rangle) & \textit{...where } z \notin FV(e_1)
\end{array}
$$

Figure 8: Thunk introduction implemented in Λ

Proof: The proofs for $\mathcal{T}_{\mathcal{L}}$ may be carried out directly using the same techniques as for $\mathcal{T}$. It is simpler, however, to take advantage of the fact that $\mathcal{T}_{\mathcal{L}} = \mathcal{L} \circ \mathcal{T}$ and reason indirectly. Specifically, one can show that for all $t \in \mathit{Programs}[\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*]$, $\mathcal{L}\langle\!\langle \mathit{eval}_{\mathrm{v}}(t) \rangle\!\rangle \simeq \mathit{eval}_i(\mathcal{L}\langle\!\langle t \rangle\!\rangle)$. Additionally, $\mathcal{L}$ and its inverse $\mathcal{L}^{-1}$ establish an equational correspondence between $\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$ and $\mathcal{T}_{\mathcal{L}}\langle\!\langle \Lambda \rangle\!\rangle^*$ (terms in the image of $\mathcal{T}_{\mathcal{L}}$ closed under β_i reduction). Now composing these results for $\mathcal{L}$ with Theorem 5 for $\mathcal{T}$ establishes each component of the current theorem (see Appendix A.3). ∎

3 Connecting the Thunk-based and the Continuation-based Simulations

We now extend Plotkin's $\mathcal{C}_{\mathrm{v}}$ to a call-by-value CPS transformation $\mathcal{C}_{\mathrm{v}}^{+}$ that handles suspension operators *delay* and *force*. Clearly $\mathcal{C}_{\mathrm{v}}^{+}$ should preserve call-by-value meaning, but in the case of thunked terms, call-by-value evaluation gives call-by-name meaning. Therefore, one would expect the result of $\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T}$ to be continuation-passing terms that encode call-by-name meaning. In fact, we show that for all $e \in \Lambda$, $(\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\langle e \rangle\!\rangle$ is identical to $\mathcal{C}_{\mathrm{n}}\langle\!\langle e \rangle\!\rangle$ modulo administrative reductions. As a byproduct, $\mathcal{C}_{\mathrm{n}}$ can be factored as $\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T}$ as captured by the following diagram.

$$
\begin{tikzcd}
\Lambda \arrow[r, "\mathcal{T}"] \arrow[dr, "\mathcal{C}_{\mathrm{n}}"'] & \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^{*} \arrow[d, "\mathcal{C}_{\mathrm{v}}^{+}"] \\
& \text{CPS terms}
\end{tikzcd}
$$

We give several applications of this factorization.

3.1 CPS transformation of thunk constructs

We begin by extending $\mathcal{C}_{\mathrm{v}}$ to transform *delay* and *force* — thereby obtaining the transformation $\mathcal{C}_{\mathrm{v}}^{+}$. The definitions follow directly from the two basic properties of thunks: *delay* e is a value; and *force* (*delay* e) $\longrightarrow_{\tau}$ e.

First, since *delay* $e \in \mathit{Values}_{\mathrm{v}}[\Lambda_{\tau}]$, $\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle \mathit{delay}\ e \rangle\!\rangle \ = \ \lambda k.k\,(\mathcal{C}_{\mathrm{v}}^{+}\langle \mathit{delay}\ e \rangle)$. Notice that in the definition of $\mathcal{C}_{\mathrm{v}}$ (see Figure 4) all expressions $\mathcal{C}_{\mathrm{v}}\langle\!\langle e \rangle\!\rangle$ require a continuation for evaluation. Therefore, an expression is delayed by simply not passing it a continuation, *i.e.*, $\mathcal{C}_{\mathrm{v}}^{+}\langle \mathit{delay}\ e \rangle \ = \ \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle$. As required, $\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle$ is a value. This effectively implements *delay* by "protecting by a λ". However, the "protecting λ" is not associated with a dummy parameter but with the continuation parameter in $\mathcal{C}_{\mathrm{v}}^{+}\langle \mathit{delay}\ e \rangle \ = \ \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle$.

Since the suspension of an expression is achieved by depriving it of a continuation, a suspension is naturally forced by supplying it with a continuation.[18] This leads to the following definition.

$$\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle \mathit{force}\ e \rangle\!\rangle \ = \ \lambda k.\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle\,(\lambda v.v\,k)$$

The following property shows that $\mathcal{C}_{\mathrm{v}}^{+}$ faithfully implements τ-reduction.

Property 6 *For all $e \in \Lambda_{\tau}$, $\lambda\beta_i \vdash \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle \mathit{force}\ (\mathit{delay}\ e) \rangle\!\rangle \ = \ \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle$*

Proof:

$$
\begin{array}{lcl}
\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle \mathit{force}\ (\mathit{delay}\ e) \rangle\!\rangle & = & \lambda k.(\lambda k.k\,(\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle))\,(\lambda v.v\,k) \\
& \longrightarrow_{\beta_i} & \lambda k.(\lambda v.v\,k)\,\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle \\
& \longrightarrow_{\beta_i} & \lambda k.\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle\,k \\
& \longrightarrow_{\beta_i} & \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle
\end{array}
$$

[18] These encodings of thunks with continuations are well-known to functional programmers. For example, they can be found in Dupont's PhD thesis [11].

$$
\begin{array}{rcl}
\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle \cdot \rangle\!\rangle & : & \Lambda_\tau \to \Lambda \\
 & \dots & \\
\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle \mathit{force}\ e \rangle\!\rangle & = & \lambda k.\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle\ (\lambda y.y\ k) \\
 & & \\
\mathcal{C}_{\mathrm{v}}^{+}\langle \cdot \rangle & : & \mathit{Values}_{\mathrm{v}}[\Lambda_\tau] \to \Lambda \\
 & \dots & \\
\mathcal{C}_{\mathrm{v}}^{+}\langle \mathit{delay}\ e \rangle & = & \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle
\end{array}
$$

Figure 9: Call-by-value CPS transformation (extended to thunks)

The last step holds since a straightforward case analysis shows that $\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle$ always has the form $\lambda k.e'$ for some $e' \in \Lambda$. ∎

The clauses of Figure 9 extend the definition of $\mathcal{C}_{\mathrm{v}}$ in Figure 4. The properties of $\mathcal{C}_{\mathrm{v}}$ as stated in Theorem 4 can be extended to the transformation $\mathcal{C}_{\mathrm{v}}^{+}$.[19]

Theorem 8 *For all $t \in \mathit{Programs}[\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*]$ and $t_1, t_2 \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$,*

1. **Indifference:** $\mathit{eval}_{\mathrm{n}}(\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t \rangle\!\rangle\ I) \simeq \mathit{eval}_{\mathrm{v}}(\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t \rangle\!\rangle\ I)$

2. **Simulation:** $\mathcal{C}_{\mathrm{v}}^{+}\langle \mathit{eval}_{\mathrm{v}}(t) \rangle \simeq \mathit{eval}_{\mathrm{n}}(\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t \rangle\!\rangle\ I)$

3. **Translation:**

$$
\begin{array}{l}
\mathit{If}\ \lambda\beta_{\mathrm{v}}\tau \vdash t_1 = t_2\ \ \mathit{then}\ \ \lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t_1 \rangle\!\rangle = \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t_2 \rangle\!\rangle \\
\mathit{Also}\ \ \lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t_1 \rangle\!\rangle = \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t_2 \rangle\!\rangle\ \ \mathit{iff}\ \ \lambda\beta \vdash \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t_1 \rangle\!\rangle = \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t_2 \rangle\!\rangle
\end{array}
$$

[19] One might expect Theorem 8 to hold for the more general Λ_τ instead of simply $\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$. However, **Simulation** fails for Λ_τ because some stuck Λ_τ programs do not stick when translated to CPS. For example, $\mathit{eval}_{\mathrm{v}}(\mathit{force}\ (\lambda x.x))$ sticks but $\mathit{eval}_{\mathrm{n}}(\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle \mathit{force}\ (\lambda x.x) \rangle\!\rangle\ (\lambda y.y)) = \lambda k.k\ (\lambda y.y)$. This mismatch on sticking is due to "improper" uses of *delay* and *force*. The proof of Theorem 8 goes through since the syntax of $\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$ only allows "proper" uses of *delay* and *force*. Furthermore, an analogue of Theorem 8 *does hold* for a typed version of Λ_τ (see [10, 15]) since well-typedness eliminates the possibility of stuck terms.

Proof: For **Indifference** and **Simulation** it is only necessary to extend Plotkin's colon-translation proof technique and definition of *stuck terms* to account for *delay* and *force*. The proofs then proceed along the same lines as Plotkin's original proofs for $\mathcal{C}_{\mathrm{v}}$ [23, pp. 148–152] (see Appendix A.4). **Translation** follows from the **Translation** component of Theorem 4 and Property 6. ∎

3.2 The connection between the thunk-based and continuation-based simulations

We now show the connection between the continuation-based simulations $\mathcal{C}_{\mathrm{n}}$ and $\mathcal{C}_{\mathrm{v}}^{+}$ and the thunk-based simulation $\mathcal{T}$. $\mathcal{C}_{\mathrm{n}}$ can be factored into two conceptually distinct steps:

- the suspension of argument evaluation (captured in $\mathcal{T}$);
- the sequentialization of function application to give the usual tail-calls of CPS terms (captured in $\mathcal{C}_{\mathrm{v}}^{+}$).

Theorem 9 *For all $e \in \Lambda$,*

$$\lambda\beta_i \vdash (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\langle e \rangle\!\rangle \ = \ \mathcal{C}_{\mathrm{n}}\langle\!\langle e \rangle\!\rangle$$

Proof: by induction over the structure of e:

case $e \ \equiv \ b$:

$$\begin{array}{rcl} (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\langle b \rangle\!\rangle & = & \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle b \rangle\!\rangle \\ & = & \lambda k.k\, b \\ & = & \mathcal{C}_{\mathrm{n}}\langle\!\langle b \rangle\!\rangle \end{array}$$

case $e \ \equiv \ x$:

$$\begin{array}{rcl} (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\langle x \rangle\!\rangle & = & \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle \mathit{force}\ x \rangle\!\rangle \\ & = & \lambda k.(\lambda k.k\, x)\, (\lambda y.y\, k) \\ & \longrightarrow_{\beta_i} & \lambda k.(\lambda y.y\, k)\, x \\ & \longrightarrow_{\beta_i} & \lambda k.x\, k \\ & = & \mathcal{C}_{\mathrm{n}}\langle\!\langle x \rangle\!\rangle \end{array}$$

case $e \ \equiv \ \lambda x.e'$:

$$\begin{array}{rcll} (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\langle \lambda x.e' \rangle\!\rangle & = & \lambda k.k\, (\lambda x.(\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\langle e' \rangle\!\rangle) & \\ & =_{\beta_i} & \lambda k.k\, (\lambda x.\mathcal{C}_{\mathrm{n}}\langle\!\langle e' \rangle\!\rangle) & \quad \textit{...by the ind. hyp.} \\ & = & \mathcal{C}_{\mathrm{n}}\langle\!\langle \lambda x.e' \rangle\!\rangle & \end{array}$$

$$
\begin{array}{rcl}
\mathcal{C}_{\mathrm{n}.opt}\langle\!|\cdot|\!\rangle & : & \Lambda \to (\Lambda \to \Lambda) \to \Lambda \\
\mathcal{C}_{\mathrm{n}.opt}\langle\!|v|\!\rangle & = & \overline{\lambda} k.k \overline{@} \mathcal{C}_{\mathrm{n}.opt}\langle v \rangle \\
\mathcal{C}_{\mathrm{n}.opt}\langle\!|x|\!\rangle & = & \overline{\lambda} k.x \underline{@} (\underline{\lambda} y.k \overline{@} y) \\
\mathcal{C}_{\mathrm{n}.opt}\langle\!|e_0\, e_1|\!\rangle & = & \overline{\lambda} k.\mathcal{C}_{\mathrm{n}.opt}\langle\!|e_0|\!\rangle \overline{@} (\overline{\lambda} y_0.y_0 \underline{@} (\underline{\lambda} k.\mathcal{C}_{\mathrm{n}.opt}\langle\!|e_1|\!\rangle \overline{@} (\overline{\lambda} y_1.k \underline{@} y_1)) \\
 & & \qquad\qquad\qquad\qquad \underline{@} (\underline{\lambda} y_2.k \overline{@} y_2)) \\
 & & \\
\mathcal{C}_{\mathrm{n}.opt}\langle\cdot\rangle & : & \mathit{Values}_{\mathrm{n}}[\Lambda] \to \Lambda \\
\mathcal{C}_{\mathrm{n}.opt}\langle b \rangle & = & b \\
\mathcal{C}_{\mathrm{n}.opt}\langle \lambda x.e \rangle & = & \underline{\lambda} x.\underline{\lambda} k.\mathcal{C}_{\mathrm{n}.opt}\langle\!|e|\!\rangle \overline{@} (\overline{\lambda} y.k \underline{@} y)
\end{array}
$$

Figure 10: Optimizing call-by-name CPS transformation

case $e \equiv e_0\, e_1$:

$$
\begin{array}{rcl}
(\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!|e_0\, e_1|\!\rangle & = & \mathcal{C}_{\mathrm{v}}^{+}\langle\!|\mathcal{T}\langle\!|e_0|\!\rangle\ (\mathit{delay}\ \mathcal{T}\langle\!|e_1|\!\rangle)|\!\rangle \\
 & = & \lambda k.(\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!|e_0|\!\rangle\ (\lambda y_0.(\lambda k.k\ (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!|e_1|\!\rangle)\ (\lambda y_1.y_0\, y_1\, k)) \\
 & \longrightarrow_{\beta_i} & \lambda k.(\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!|e_0|\!\rangle\ (\lambda y_0.(\lambda y_1.y_0\, y_1\, k)\ (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!|e_1|\!\rangle) \\
 & \longrightarrow_{\beta_i} & \lambda k.(\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!|e_0|\!\rangle\ (\lambda y_0.y_0\ (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!|e_1|\!\rangle\ k) \\
 & =_{\beta_i} & \lambda k.\mathcal{C}_{\mathrm{n}}\langle\!|e_0|\!\rangle\ (\lambda y_0.y_0\, \mathcal{C}_{\mathrm{n}}\langle\!|e_1|\!\rangle\ k) \quad \ldots \mathit{by\ the\ ind.\ hyp.} \\
 & = & \mathcal{C}_{\mathrm{n}}\langle\!|e_0\, e_1|\!\rangle
\end{array}
$$

∎

Note that $\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T}$ and $\mathcal{C}_{\mathrm{n}}$ only differ by administrative reductions. In fact, if we consider versions of $\mathcal{C}_{\mathrm{n}}$ and $\mathcal{C}_{\mathrm{v}}$ which optimize by removing administrative reductions, then the correspondence holds up to identity (*i.e.*, up to α-equivalence).

Figures 10 and 11 present the optimizing transformations $\mathcal{C}_{\mathrm{n}.opt}$ and $\mathcal{C}_{\mathrm{v}.opt}$ given by Danvy and Filinski [8, pp. 387 and 367].[20] The transformations are presented in a two-level language *à la* Nielson and Nielson [21]. Operationally, the overlined λ's and @'s correspond to functional abstractions and

[20]The output of $\mathcal{C}_{\mathrm{n}.opt}$ is $\beta_{\mathrm{v}}\eta_{\mathrm{v}}$ equivalent to the output of $\mathcal{C}_{\mathrm{n}}$ (similarly for $\mathcal{C}_{\mathrm{v}.opt}$ and $\mathcal{C}_{\mathrm{v}}$). A proof of **Indifference** and **Simulation** for $\mathcal{C}_{\mathrm{v}.opt}$ is given in [8]. This proof extends to $\mathcal{C}_{\mathrm{v}.opt}^{+}$ in a straightforward manner.

$$
\begin{array}{rcl}
\mathcal{C}_{\mathrm{v}.opt}\llbracket\cdot\rrbracket & : & \Lambda \to (\Lambda \to \Lambda) \to \Lambda \\
\mathcal{C}_{\mathrm{v}.opt}\llbracket v\rrbracket & = & \overline{\lambda} k.k \overline{@} \mathcal{C}_{\mathrm{v}.opt}\langle v\rangle \\
\mathcal{C}_{\mathrm{v}.opt}\llbracket e_0\, e_1\rrbracket & = & \overline{\lambda} k.\mathcal{C}_{\mathrm{v}.opt}\llbracket e_0\rrbracket \overline{@} (\overline{\lambda} y_0.\mathcal{C}_{\mathrm{v}.opt}\llbracket e_1\rrbracket \overline{@} (\overline{\lambda} y_1.y_0 \underline{@} y_1 \underline{@} (\underline{\lambda} y_2.k \overline{@} y_2))) \\
 & & \\
\mathcal{C}_{\mathrm{v}.opt}\langle\cdot\rangle & : & \mathit{Values}_{\mathrm{v}}[\Lambda] \to \Lambda \\
\mathcal{C}_{\mathrm{v}.opt}\langle b\rangle & = & b \\
\mathcal{C}_{\mathrm{v}.opt}\langle x\rangle & = & x \\
\mathcal{C}_{\mathrm{v}.opt}\langle \lambda x.e\rangle & = & \underline{\lambda} x.\underline{\lambda} k.\mathcal{C}_{\mathrm{v}.opt}\llbracket e\rrbracket \overline{@} (\overline{\lambda} y.k \underline{@} y)
\end{array}
$$

Figure 11: Optimizing call-by-value CPS transformation

applications in the program implementing the translation, while the underlined λ's and @'s represent abstract-syntax constructors. The figures can be transliterated into functional programs.

The optimizing transformation $\mathcal{C}^{+}_{\mathrm{v}.opt}$ is obtained from $\mathcal{C}_{\mathrm{v}.opt}$ by adding the following definitions.

$$
\begin{array}{rcl}
\mathcal{C}^{+}_{\mathrm{v}.opt}\llbracket \mathit{force}\ e\rrbracket & = & \overline{\lambda} k.\mathcal{C}^{+}_{\mathrm{v}.opt}\llbracket e\rrbracket \overline{@} (\overline{\lambda} y_0.y_0 \underline{@} (\underline{\lambda} y_1.k \overline{@} y_1)) \\
\mathcal{C}^{+}_{\mathrm{v}.opt}\langle \mathit{delay}\ e\rangle & = & \underline{\lambda} k.\mathcal{C}^{+}_{\mathrm{v}.opt}\llbracket e\rrbracket \overline{@} (\overline{\lambda} y.k \underline{@} y)
\end{array}
$$

Taking an operational view of these two-level specifications, the following theorem states that, for all $e \in \Lambda$, the result of applying $\mathcal{C}^{+}_{\mathrm{v}.opt}$ to $\mathcal{T}\llbracket e\rrbracket$ (with an initial continuation $\overline{\lambda} a.a$) is α-equivalent to the result of applying $\mathcal{C}_{\mathrm{n}.opt}$ to e (with an initial continuation $\overline{\lambda} a.a$).

Theorem 10 *For all $e \in \Lambda$,*

$$(\mathcal{C}^{+}_{\mathrm{v}.opt} \circ \mathcal{T})\llbracket e\rrbracket \;=\; \mathcal{C}_{\mathrm{n}.opt}\llbracket e\rrbracket$$

Proof: A simple structural induction similar to the one required in the proof of Theorem 9. We show only the case for identifiers (the others are similar). The overlined constructs are computed at translation time, and thus simplifying overlined constructs using β-conversion yields equivalent specifications.

case $e \equiv x$:

$$\begin{array}{rcl}(\mathcal{C}^+_{\mathrm{v}.opt} \circ \mathcal{T})\langle\!\langle x \rangle\!\rangle & = & \overline{\lambda} k.(\overline{\lambda} k.k \overline{@} x) \overline{@} (\overline{\lambda} y.y \underline{@} (\underline{\lambda} y.k \overline{@} y)) \\ & = & \overline{\lambda} k.(\overline{\lambda} y.y \underline{@} (\underline{\lambda} y.k \overline{@} y)) \overline{@} x \\ & = & \overline{\lambda} k.x \underline{@} (\underline{\lambda} y.k \overline{@} y) \\ & = & \mathcal{C}_{\mathrm{n}.opt}\langle\!\langle x \rangle\!\rangle\end{array}$$

∎

3.3 Applications

3.3.1 Deriving correctness properties of $\mathcal{C}_{\mathrm{n}}$

When working with CPS, one often needs to establish technical properties for both a call-by-name and a call-by-value CPS transformation. This requires two sets of proofs that both involve CPS. By appealing to the factoring property, however, often only one set of proofs over call-by-value CPS terms is necessary. The second set of proofs deals with thunked terms which have a simpler structure. For instance, **Indifference** and **Simulation** for $\mathcal{C}_{\mathrm{n}}$ follow from **Indifference** and **Simulation** for $\mathcal{C}^+_{\mathrm{v}}$ and $\mathcal{T}$ and Theorem 9.[21]

For **Indifference**, let $e, b \in \Lambda$ where b is a basic constant. Then

$$\begin{array}{lll} & eval_{\mathrm{v}}(\mathcal{C}_{\mathrm{n}}\langle\!\langle e \rangle\!\rangle\, (\lambda y.y)) = b & \\ \Leftrightarrow & eval_{\mathrm{v}}((\mathcal{C}^+_{\mathrm{v}} \circ \mathcal{T})\langle\!\langle e \rangle\!\rangle\, (\lambda y.y)) = b & \textit{... Theorem 9 and soundness of } \beta_{\mathrm{v}} \\ \Leftrightarrow & eval_{\mathrm{n}}((\mathcal{C}^+_{\mathrm{v}} \circ \mathcal{T})\langle\!\langle e \rangle\!\rangle\, (\lambda y.y)) = b & \textit{... Theorem 8 } \textbf{(Indifference)} \\ \Leftrightarrow & eval_{\mathrm{n}}(\mathcal{C}_{\mathrm{n}}\langle\!\langle e \rangle\!\rangle\, (\lambda y.y)) = b & \textit{... Theorem 9 and soundness of } \beta \end{array}$$

For **Simulation**, let $e, b \in \Lambda$ where b is a basic constant. Then

$$\begin{array}{lll} & eval_{\mathrm{n}}(e) = b & \\ \Leftrightarrow & eval_{\mathrm{v}}(\mathcal{T}\langle\!\langle e \rangle\!\rangle) = b & \textit{... Theorem 5 } \textbf{(Simulation)} \\ \Leftrightarrow & eval_{\mathrm{n}}((\mathcal{C}^+_{\mathrm{v}} \circ \mathcal{T})\langle\!\langle e \rangle\!\rangle\, (\lambda y.y)) = b & \textit{... Theorem 8 } \textbf{(Simulation)} \\ \Leftrightarrow & eval_{\mathrm{v}}((\mathcal{C}^+_{\mathrm{v}} \circ \mathcal{T})\langle\!\langle e \rangle\!\rangle\, (\lambda y.y)) = b & \textit{... Theorem 8 } \textbf{(Indifference)} \\ \Leftrightarrow & eval_{\mathrm{v}}(\mathcal{C}_{\mathrm{n}}\langle\!\langle e \rangle\!\rangle\, (\lambda y.y)) = b & \textit{... Theorem 9 and soundness of } \beta_{\mathrm{v}} \end{array}$$

For **Translation**, it is not possible to establish Theorem 3 (**Translation** for $\mathcal{C}_{\mathrm{n}}$) in the manner above since Theorem 8 (**Translation** for $\mathcal{C}^+_{\mathrm{v}}$) is weaker in comparison. However, the following weaker version can be derived (the

[21] Here we show only the results where evaluation is undefined or results in a basic constant b. Appendix A.1.2 gives the derivation of $\mathcal{C}_{\mathrm{n}}$ **Simulation** for arbitrary results.

full version is proved in Appendix A.1.3). Let $e_1, e_2 \in \Lambda$. Then

$$
\begin{array}{lll}
 & \lambda\beta \vdash e_1 = e_2 & \\
\Leftrightarrow & \lambda\beta_{\mathrm{v}}\tau \vdash \mathcal{T}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle & \ldots \textit{Theorem 5}\ \textbf{(Translation)} \\
\Rightarrow & \lambda\beta_i \vdash (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\langle e_1 \rangle\!\rangle = (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\langle e_2 \rangle\!\rangle & \ldots \textit{Theorem 8}\ \textbf{(Translation)} \\
\Leftrightarrow & \lambda\beta_i \vdash \mathcal{C}_{\mathrm{n}}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{C}_{\mathrm{n}}\langle\!\langle e_2 \rangle\!\rangle & \ldots \textit{Theorem 9} \\
\Rightarrow & \lambda\beta_i \vdash \mathcal{C}_{\mathrm{n}}\langle\!\langle e_1 \rangle\!\rangle\, I = \mathcal{C}_{\mathrm{n}}\langle\!\langle e_2 \rangle\!\rangle\, I & \ldots \textit{compatibility of} =_{\beta_i}
\end{array}
$$

3.3.2 Deriving a CPS transformation directed by strictness information

Strictness information indicates arguments that may be safely evaluated eagerly (*i.e.*, without being delayed) — in effect, reducing the number of thunks needed in a program and the overhead associated with creating and evaluating suspensions [5, 10, 22]. In recent work [10], we gave a transformation $\mathcal{T}_s$ that optimizes thunk introduction based on strictness information.[22] We then used the factorization of $\mathcal{C}_{\mathrm{n}}$ by $\mathcal{C}_{\mathrm{v}}^{+}$ and $\mathcal{T}$ to derive an optimized CPS transformation $\mathcal{C}_s$ for strictness-analyzed call-by-name terms. This situation is summarized by the following diagram.

$$
\begin{tikzcd}
\Lambda \arrow[r, "\mathcal{T}_s", "\text{+ strictness info}"'] \arrow[dr, "\mathcal{C}_s"'] & \text{(thunked \& unthunked)} \arrow[d, "\mathcal{C}_{\mathrm{v}}^{+}"] \\
 & \text{CPS terms}
\end{tikzcd}
$$

The resulting transformation $\mathcal{C}_s$ yields both call-by-name-like and call-by-value-like continuation-passing terms. Due to the factorization, the proof of correctness for the optimized transformation follows as a corollary of the correctness of the strictness analysis, and the correctness of $\mathcal{T}$ and $\mathcal{C}_{\mathrm{v}}^{+}$.

3.3.3 Deriving a call-by-need CPS transformation

Okasaki, Lee, and Tarditi [22] have also applied the factorization to obtain a "call-by-need CPS transformation" $\mathcal{C}_{need}$. The lazy evaluation strategy

[22] Amtoft [1] and Stecker and Wand [32] have proven the correctness of transformations which optimize the introduction of thunks based on strictness information.

characterizing call-by-need is captured by memoizing the thunks [5]. $\mathcal{C}_{need}$ is obtained by extending $\mathcal{C}_{\mathrm{v}}^{+}$ to transform memo-thunks to CPS terms with store operations (which are used to implement the memoization) and composing with the memo-thunk introduction as follows.

$$\begin{tikzcd}
\Lambda \arrow[r, "\mathcal{T}"] \arrow[dr, "\mathcal{C}_{need}"'] & \text{memo-thunks} \arrow[d, "\mathcal{C}_{\mathrm{v}}^{+}"] \\
& \begin{array}{c}\text{CPS terms}\\ \scriptstyle + \text{ store operations}\end{array}
\end{tikzcd}$$

Okasaki *et al.* optimize $\mathcal{C}_{need}$ by using strictness information along the lines discussed above. They also use sharing information to detect where memo-thunks can be replaced by ordinary thunks. In both cases, optimizations are achieved by working with simpler thunked terms as opposed to working directly with CPS terms.

3.4 Assessment

Thunks can be used to factor a variety of call-by-name CPS transformations. In addition to those discussed here, we have factored a variant of Reynolds's CPS transformation directed by strictness information [15, 26], as well as a call-by-name analogue of Fischer's call-by-value CPS transformation [12, 31].

Obtaining the desired call-by-name CPS transformation *via* $\mathcal{C}_{\mathrm{v}}^{+}$ and $\mathcal{T}$ depends on the representation of thunks. For example, if one works with $\mathcal{T}_{\mathcal{L}}$ instead of $\mathcal{T}$, $\mathcal{C}_{\mathrm{v}} \circ \mathcal{T}_{\mathcal{L}}$ still gives a valid CPS simulation of call-by-name by call-by-value. However, the following derivations show that β_i equivalence with $\mathcal{C}_{\mathrm{n}}$ is not obtained (*i.e.*, $\lambda\beta_i \not\vdash \mathcal{C}_{\mathrm{n}}\langle\!\langle e\rangle\!\rangle = (\mathcal{C}_{\mathrm{v}} \circ \mathcal{T}_{\mathcal{L}})\langle\!\langle e\rangle\!\rangle$).

$$\begin{array}{rcl}
(\mathcal{C}_{\mathrm{v}} \circ \mathcal{T}_{\mathcal{L}})\langle\!\langle x\rangle\!\rangle & = & \mathcal{C}_{\mathrm{v}}\langle\!\langle x\, b\rangle\!\rangle \\
& = & \lambda k.(x\, b)\, k
\end{array}$$

$$\begin{array}{rcl}
(\mathcal{C}_{\mathrm{v}} \circ \mathcal{T}_{\mathcal{L}})\langle\!\langle e_0\, e_1\rangle\!\rangle & = & \mathcal{C}_{\mathrm{v}}\langle\!\langle \mathcal{T}_{\mathcal{L}}\langle\!\langle e_0\rangle\!\rangle\, (\lambda z.\mathcal{T}_{\mathcal{L}}\langle\!\langle e_1\rangle\!\rangle)\rangle\!\rangle \\
& = & \lambda k.(\mathcal{C}_{\mathrm{v}} \circ \mathcal{T}_{\mathcal{L}})\langle\!\langle e_0\rangle\!\rangle\, (\lambda y.(y\, (\lambda z.(\mathcal{C}_{\mathrm{v}} \circ \mathcal{T}_{\mathcal{L}})\langle\!\langle e_1\rangle\!\rangle))\, k)
\end{array}$$

The representation of thunks given by $\mathcal{T}_{\mathcal{L}}$ is too concrete in the sense that the delaying and forcing of computation is achieved using specific instances

$$\Gamma \vdash b : \iota \qquad \Gamma \vdash x : \Gamma(x)$$

$$\frac{\Gamma, x{:}\sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \lambda x.e : \sigma_1 \rightarrow \sigma_2} \qquad \frac{\Gamma \vdash e_0 : \sigma_1 \rightarrow \sigma_2 \quad \Gamma \vdash e_1 : \sigma_1}{\Gamma \vdash e_0\, e_1 : \sigma_2}$$

$$\begin{array}{lcl} \sigma & \in & \mathit{Types}[\Lambda_\tau] \\ \sigma & ::= & \iota \mid \sigma_1 \rightarrow \sigma_2 \end{array} \qquad \begin{array}{lcl} \Gamma & \in & \mathit{Asm}[\Lambda_\tau] \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}\sigma \end{array}$$

Figure 12: Typing rules for Λ

of the more general abstraction and application constructs. When composed with $\mathcal{T}_\mathcal{L}$, $\mathcal{C}_\mathrm{v}$ treats the specific instances of thunks in their full generality, and the resulting CPS terms contain a level of inessential encoding of *delay* and *force*.

4 Thunks in a Typed Setting

Plotkin's continuation-passing transformations were originally stated in terms of untyped λ-calculi. These transformations have been shown to preserve well-typedness of terms [13, 14, 18, 20]. In this section, we introduce typing rules for the suspension operators of Λ_τ and show that the thunk transformation $\mathcal{T}$ also preserves well-typedness of terms. In addition, we show how the relationship between $\mathcal{C}_\mathrm{v}^+ \circ \mathcal{T}$ and $\mathcal{C}_\mathrm{n}$ is reflected in transformations on types.

4.1 Thunk introduction for a typed language

Figure 12 presents type assignment rules for the language Λ [4]. Γ is a set $\{x_1 : \sigma_1, ..., x_n : \sigma_n\}$ of type assumptions for identifiers. We assume that the identifiers of Γ are pairwise distinct. $\Gamma, x{:}\sigma$ abbreviates $\Gamma \cup \{x : \sigma\}$.

Figure 13 presents type assignment rules for the language Λ_τ. A type constructor $\widetilde{\cdot}$ is added to type suspension constructs *delay* and *force*. $\widetilde{\sigma}$ types a suspension (*i.e.*, a thunk) that will yield a value of type σ when forced.[23]

[23]Note that we use the same meta-variables (Γ for type assumptions, σ for types, and e for terms) for both Λ and Λ_τ. Ambiguity is avoided by subscripting the typing judgement symbol $\vdash_\tau$ for the language Λ_τ.

$$\Gamma \vdash_\tau b : \iota \qquad \Gamma \vdash_\tau x : \Gamma(x)$$

$$\frac{\Gamma, x{:}\sigma_1 \vdash_\tau e : \sigma_2}{\Gamma \vdash_\tau \lambda x.e : \sigma_1 \to \sigma_2} \qquad \frac{\Gamma \vdash_\tau e_0 : \sigma_1 \to \sigma_2 \quad \Gamma \vdash_\tau e_1 : \sigma_1}{\Gamma \vdash_\tau e_0\, e_1 : \sigma_2}$$

$$\frac{\Gamma \vdash_\tau e : \sigma}{\Gamma \vdash_\tau \mathit{delay}\ e : \widetilde{\sigma}} \qquad \frac{\Gamma \vdash_\tau e : \widetilde{\sigma}}{\Gamma \vdash_\tau \mathit{force}\ e : \sigma}$$

$$\begin{array}{lcl} \sigma & \in & \mathit{Types}[\Lambda_\tau] \\ \sigma & ::= & \iota \mid \sigma_1 \to \sigma_2 \mid \widetilde{\sigma} \end{array} \qquad \begin{array}{lcl} \Gamma & \in & \mathit{Asm}[\Lambda_\tau] \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}\sigma \end{array}$$

Figure 13: Typing rules for Λ_τ

$$\begin{array}{rcl} \mathcal{T}\langle\!|\cdot|\!\rangle & : & \mathit{Types}[\Lambda] \to \mathit{Types}[\Lambda_\tau] \\ \mathcal{T}\langle\!|\iota|\!\rangle & = & \iota \\ \mathcal{T}\langle\!|\sigma_1 \to \sigma_2|\!\rangle & = & \widetilde{\mathcal{T}\langle\!|\sigma_1|\!\rangle} \to \mathcal{T}\langle\!|\sigma_2|\!\rangle \end{array} \qquad \begin{array}{rcl} \mathcal{T}\langle\!|\cdot|\!\rangle & : & \mathit{Asm}[\Lambda] \to \mathit{Asm}[\Lambda_\tau] \\ \mathcal{T}\langle\!|\Gamma, x{:}\sigma|\!\rangle & = & \mathcal{T}\langle\!|\Gamma|\!\rangle, x{:}\widetilde{\mathcal{T}\langle\!|\sigma|\!\rangle} \end{array}$$

Figure 14: Transformation on types for $\mathcal{T}$

Figure 14 presents the type transformation for $\mathcal{T}$. The definition of $\mathcal{T}$ on function types and on type assumptions reflects the fact that all function arguments are thunks in the image of $\mathcal{T}$.

The following property states that $\mathcal{T}$ preserves well-typedness of terms.

Property 7 *If* $\Gamma \vdash e : \sigma$ *then* $\mathcal{T}\langle\!|\Gamma|\!\rangle \vdash_\tau \mathcal{T}\langle\!|e|\!\rangle : \mathcal{T}\langle\!|\sigma|\!\rangle$.

Proof: by induction over the derivation of $\Gamma \vdash e : \sigma$. ∎

4.2 CPS transformations for a typed language

Figures 15 and 16 present the type transformations for $\mathcal{C}_\mathrm{n}$ and $\mathcal{C}_\mathrm{v}$ (where *Ans* is a distinguished type of final answers [18]). The definition of $\mathcal{C}_\mathrm{n}$ on

$$
\begin{aligned}
\mathcal{C}_{\mathrm{n}}\langle\!|\cdot|\!\rangle &: \mathit{Types}[\Lambda] \to \mathit{Types}[\Lambda] \\
\mathcal{C}_{\mathrm{n}}\langle\!|\sigma|\!\rangle &= (\mathcal{C}_{\mathrm{n}}\langle\sigma\rangle \to \mathit{Ans}) \to \mathit{Ans} \\
\mathcal{C}_{\mathrm{n}}\langle\iota\rangle &= \iota \\
\mathcal{C}_{\mathrm{n}}\langle\sigma_1 \to \sigma_2\rangle &= \mathcal{C}_{\mathrm{n}}\langle\!|\sigma_1|\!\rangle \to \mathcal{C}_{\mathrm{n}}\langle\!|\sigma_2|\!\rangle
\end{aligned}
\qquad
\begin{aligned}
\mathcal{C}_{\mathrm{n}}\langle\!|\cdot|\!\rangle &: \mathit{Asm}[\Lambda] \to \mathit{Asm}[\Lambda] \\
\mathcal{C}_{\mathrm{n}}\langle\!|\Gamma, x:\sigma|\!\rangle &= \mathcal{C}_{\mathrm{n}}\langle\!|\Gamma|\!\rangle, x:\mathcal{C}_{\mathrm{n}}\langle\!|\sigma|\!\rangle
\end{aligned}
$$

Figure 15: Transformation on types for $\mathcal{C}_{\mathrm{n}}$

$$
\begin{aligned}
\mathcal{C}_{\mathrm{v}}\langle\!|\cdot|\!\rangle &: \mathit{Types}[\Lambda] \to \mathit{Types}[\Lambda] \\
\mathcal{C}_{\mathrm{v}}\langle\!|\sigma|\!\rangle &= (\mathcal{C}_{\mathrm{v}}\langle\sigma\rangle \to \mathit{Ans}) \to \mathit{Ans} \\
\mathcal{C}_{\mathrm{v}}\langle\iota\rangle &= \iota \\
\mathcal{C}_{\mathrm{v}}\langle\sigma_1 \to \sigma_2\rangle &= \mathcal{C}_{\mathrm{v}}\langle\sigma_1\rangle \to \mathcal{C}_{\mathrm{v}}\langle\!|\sigma_2|\!\rangle
\end{aligned}
\qquad
\begin{aligned}
\mathcal{C}_{\mathrm{v}}\langle\!|\cdot|\!\rangle &: \mathit{Asm}[\Lambda] \to \mathit{Asm}[\Lambda] \\
\mathcal{C}_{\mathrm{v}}\langle\!|\Gamma, x:\sigma|\!\rangle &= \mathcal{C}_{\mathrm{v}}\langle\!|\Gamma|\!\rangle, x:\mathcal{C}_{\mathrm{v}}\langle\sigma\rangle
\end{aligned}
$$

Figure 16: Transformation on types for $\mathcal{C}_{\mathrm{v}}$

function types and on type assumptions reflects the fact that source functions are translated to functions whose arguments are expressions needing a continuation. The definition of $\mathcal{C}_{\mathrm{v}}$ on function types and on type assumptions reflects the fact that source functions are translated to functions whose arguments are values.

The following property states that $\mathcal{C}_{\mathrm{n}}$ and $\mathcal{C}_{\mathrm{v}}$ preserve well-typedness of terms.

Property 8

- *If* $\Gamma \vdash e : \sigma$ *then* $\mathcal{C}_{\mathrm{n}}\langle\!|\Gamma|\!\rangle \vdash \mathcal{C}_{\mathrm{n}}\langle\!|e|\!\rangle : \mathcal{C}_{\mathrm{n}}\langle\!|\sigma|\!\rangle$.
- *If* $\Gamma \vdash e : \sigma$ *then* $\mathcal{C}_{\mathrm{v}}\langle\!|\Gamma|\!\rangle \vdash \mathcal{C}_{\mathrm{v}}\langle\!|e|\!\rangle : \mathcal{C}_{\mathrm{v}}\langle\!|\sigma|\!\rangle$.

Proof: by induction over the derivation of $\Gamma \vdash e : \sigma$ (see [13, 14, 18, 20] for further details). ∎

4.3 Connecting the thunk-based and the continuation-based simulations

The following definition extends $\mathcal{C}_\mathrm{v}$ to the types of $\Lambda_\mathcal{T}$.

$$\mathcal{C}^+_\mathrm{v}\langle\widetilde{\sigma}\rangle \;=\; \mathcal{C}^+_\mathrm{v}\langle\!\langle\sigma\rangle\!\rangle$$

This reflects the fact that suspensions are translated to terms expecting a continuation (see Figure 9). It is simple to show that the well-typedness property for $\mathcal{C}_\mathrm{v}$ (Property 8) extends to $\mathcal{C}^+_\mathrm{v}$.

The factoring of $\mathcal{C}_\mathrm{n}$ by $\mathcal{T}$ and $\mathcal{C}^+_\mathrm{v}$ (Theorem 9) is reflected in the transformations on types as follows.

Property 9

1. $\mathcal{C}^+_\mathrm{v}\langle\!\langle\mathcal{T}\langle\!\langle\sigma\rangle\!\rangle\rangle\!\rangle \;=\; \mathcal{C}_\mathrm{n}\langle\!\langle\sigma\rangle\!\rangle$ *types*
2. $\mathcal{C}^+_\mathrm{v}\langle\mathcal{T}\langle\!\langle\sigma\rangle\!\rangle\rangle \;=\; \mathcal{C}_\mathrm{n}\langle\sigma\rangle$ *value types*
3. $\mathcal{C}^+_\mathrm{v}\langle\!\langle\mathcal{T}\langle\!\langle\Gamma\rangle\!\rangle\rangle\!\rangle \;=\; \mathcal{C}_\mathrm{n}\langle\!\langle\Gamma\rangle\!\rangle$ *type assumptions*

Proof: The proof of components 1 and 2 proceeds by induction over the structure of σ. The case of function types for values is as follows.

$$\begin{array}{rcll}
\mathcal{C}_\mathrm{v}\langle\mathcal{T}\langle\!\langle\sigma_1\to\sigma_2\rangle\!\rangle\rangle & = & \mathcal{C}^+_\mathrm{v}\langle\widetilde{\mathcal{T}\langle\!\langle\sigma_1\rangle\!\rangle}\to\mathcal{T}\langle\!\langle\sigma_2\rangle\!\rangle\rangle & \\
& = & \mathcal{C}^+_\mathrm{v}\langle\widetilde{\mathcal{T}\langle\!\langle\sigma_1\rangle\!\rangle}\rangle\to\mathcal{C}^+_\mathrm{v}\langle\!\langle\mathcal{T}\langle\!\langle\sigma_2\rangle\!\rangle\rangle\!\rangle & \\
& = & \mathcal{C}^+_\mathrm{v}\langle\!\langle\mathcal{T}\langle\!\langle\sigma_1\rangle\!\rangle\rangle\!\rangle\to\mathcal{C}^+_\mathrm{v}\langle\!\langle\mathcal{T}\langle\!\langle\sigma_2\rangle\!\rangle\rangle\!\rangle & \\
& = & \mathcal{C}_\mathrm{n}\langle\!\langle\sigma_1\rangle\!\rangle\to\mathcal{C}_\mathrm{n}\langle\!\langle\sigma_2\rangle\!\rangle & \ldots \textit{by ind. hyp.} \\
& = & \mathcal{C}_\mathrm{n}\langle\sigma_1\to\sigma_2\rangle &
\end{array}$$

∎

4.4 Assessment

$\mathcal{C}_\mathrm{n}$ and $\mathcal{C}_\mathrm{v}$ are alike in that they both introduce continuation-passing terms. This is reflected by the similarity in the definitions $\mathcal{C}_\mathrm{n}\langle\!\langle\sigma\rangle\!\rangle = (\mathcal{C}_\mathrm{n}\langle\sigma\rangle\to Ans)\to Ans$ and $\mathcal{C}_\mathrm{v}\langle\!\langle\sigma\rangle\!\rangle = (\mathcal{C}_\mathrm{v}\langle\sigma\rangle\to Ans)\to Ans$. $\mathcal{C}_\mathrm{n}$ and $\mathcal{C}_\mathrm{v}$ differ in how arguments are treated. This is reflected by the difference in the definitions $\mathcal{C}_\mathrm{n}\langle\sigma_1\to\sigma_2\rangle = \mathcal{C}_\mathrm{n}\langle\!\langle\sigma_1\rangle\!\rangle\to\mathcal{C}_\mathrm{n}\langle\!\langle\sigma_2\rangle\!\rangle$ and $\mathcal{C}_\mathrm{v}\langle\sigma_1\to\sigma_2\rangle = \mathcal{C}_\mathrm{v}\langle\sigma_1\rangle\to\mathcal{C}_\mathrm{v}\langle\!\langle\sigma_2\rangle\!\rangle$. The only effect of $\mathcal{T}$ is to change how arguments are treated. This is reflected by the fact that the only effect of $\mathcal{T}$ on types is the introduction of suspension types for arguments, *i.e.*, $\mathcal{T}\langle\!\langle\sigma_1\to\sigma_2\rangle\!\rangle = \widetilde{\mathcal{T}\langle\!\langle\sigma_1\rangle\!\rangle}\to\mathcal{T}\langle\!\langle\sigma_2\rangle\!\rangle$. Thus, the action by $\mathcal{T}$ is exactly what is needed to move from $\mathcal{C}^+_\mathrm{v}$ to $\mathcal{C}_\mathrm{n}$.

5 Related Work

Ingerman [17], in his work on the implementation of Algol 60, gave a general technique for generating machine code implementing procedure parameter passing. The term *thunk* was coined to refer to the compiled representation of a delayed expression as it gets pushed on the control stack [25]. Since then, the term *thunk* has been applied to other higher-level representations of delayed expressions and we have followed this practice.

Bloss, Hudak, and Young [5] study thunks as the basis of implementation of lazy evaluation. Optimizations associated with lazy evaluation (*e.g.*, overwriting a forced expression with its resulting value) are encapsulated in the thunk. They give several representations with differing effects on space and time overhead.

Riecke [28] has used thunks to obtain fully-abstract translations between versions of PCF with differing evaluation strategies. In effect, he establishes a fully-abstract version of the **Simulation** property of Theorem 7.[24] The thunk translation required for full abstraction is much more complicated than our transformation $\mathcal{T}$ and consequently it cannot be used to factor $\mathcal{C}_{\mathrm{n}}$. In addition, since Riecke's translation is based on typed-indexed retractions, it does not seem possible to use it (and the corresponding results) in an untyped setting as we require here.

Asperti and Curien give an interesting formulation of thunks in a categorical setting [2, 7]. Two combinators *freeze* and *unfreeze*, which are analogous to our *delay* and *force* but have slightly different equational properties, are used to implement lazy evaluation in the Categorical Abstract Machine. In addition, *freeze* and *unfreeze* can be elegantly characterized using a comonad.

6 Conclusion

The technique of thunks has been widely applied in both theory and practice. Our aim has been to clarify the properties of thunks with respect to Plotkin's classic study of evaluation strategies and continuation-passing styles [23].

We have shown that all of the correctness properties of the continuation-based simulation $\mathcal{C}_{\mathrm{n}}$ can be obtained via a simpler thunk-based transforma-

[24]The **Indifference** property is also immediate for Riecke since all function arguments are values in the image of his translation (and this property is maintained under reductions).

tion $\mathcal{T}$. As a consequence, simulating call-by-name operational behavior and equational reasoning in a call-by-value setting are simpler than with $\mathcal{C}_{\mathrm{n}}$.

Furthermore, we have shown that the thunk transformation $\mathcal{T}$ establishes a previously unrecognized connection between the simulations $\mathcal{C}_{\mathrm{n}}$ and $\mathcal{C}_{\mathrm{v}}$ — $\mathcal{C}_{\mathrm{n}}$ can be obtained by composing $\mathcal{C}_{\mathrm{v}}^{+}$ with $\mathcal{T}$. The benefit is that almost all the technical properties of $\mathcal{C}_{\mathrm{n}}$ follow from the formal properties of $\mathcal{C}_{\mathrm{v}}^{+}$ and $\mathcal{T}$. $\mathcal{T}$ can also be used to factor a call-by-name version of Fischer's call-by-value CPS transformation $\mathcal{F}$ as used by Sabry and Felleisen [31], and also to factor a variant of Reynolds's CPS transformation directed by strictness information [15]. These factorings prove useful in several applications dealing with the implementation of call-by-name and lazy languages [10, 22].

For simplicity, we have presented both the simulation and the factorization results for thunks using simple Λ terms. However, the results scale up to more realistic languages with *e.g.*, primitive operators, products and co-products, and recursive functions [15]. In a preliminary version of Section 3.2 [9], we presented the factorization of $\mathcal{C}_{\mathrm{n}}$ *via* $\mathcal{C}_{\mathrm{v}}^{+}$ and $\mathcal{T}$, for the untyped λ-calculus with n-ary functions (*à la* Scheme [6]).

This work is part of a broader investigation of the structure of continuation-passing styles. Elsewhere [15, 16] we have shown how structural relationships between many different continuation-passing styles can be exploited to simplify transformations, correctness proofs, and reasoning about CPS programs. This investigation aims to clarify intuition and to aid in understanding the often complicated structure of CPS programs.

Acknowledgements

Andrzej Filinski, Julia Lawall, Sergey Kotov and David Schmidt gave helpful comments on earlier drafts of this paper. Thanks are also due to Dave Sands for several useful discussions. Finally, we are grateful to the reviewers.

The diagrams of Section 3 were drawn with Kristoffer Rose's XY-pic package.

A Proofs

A.1 Correctness of $\mathcal{C}_{\mathrm{n}}$

A.1.1 Indifference

One may appeal to the factoring of $\mathcal{C}_{\mathrm{n}}$ to prove **Indifference** for $\mathcal{C}_{\mathrm{n}}$ up to β-equivalence. The proof is similar to the proof of **Simulation** in the following section. To prove **Indifference** up to α-equivalence (as stated in Theorem 3), consider the following grammar.[25]

$$\begin{array}{lcl} u & \in & \Lambda_{cps} \\ u & ::= & w \mid u\,w \end{array} \qquad\qquad \begin{array}{lcl} w & \in & \mathit{Values}[\Lambda_{cps}] \\ w & ::= & b \mid x \mid \lambda x.u \end{array}$$

An induction on the structure of $e \in \Lambda$ shows that $\mathcal{C}_{\rm n}\langle\!\langle e\rangle\!\rangle \in \mathit{Values}[\Lambda_{cps}]$. It then follows that $\mathcal{C}_{\rm n}\langle\!\langle e\rangle\!\rangle\,(\lambda x.x) \in \Lambda_{cps}$. Now **Indifference** for $\mathcal{C}_{\rm n}$ follows from the fact that for all $u \in \mathit{Programs}[\Lambda_{cps}], u \longmapsto_{\rm n} u'$ *iff* $u \longmapsto_{\rm v} u'$ (and moreover $u' \in \mathit{Programs}[\Lambda_{cps}]$).

A.1.2 Simulation

Simulation for $\mathcal{C}_{\rm n}$ can be derived by appealing to the factoring of $\mathcal{C}_{\rm n}$. Let $e \in \Lambda$. Then

$$\begin{array}{rcl}
\mathcal{C}_{\rm n}\langle \mathit{eval}_{\rm n}(e)\rangle & \simeq_{\beta_i} & \mathcal{C}_{\rm v}^{+}\langle \mathcal{T}\langle\!\langle \mathit{eval}_{\rm n}(e)\rangle\!\rangle\rangle \\
& & \textit{...factoring of } \mathcal{C}_{\rm n} \textit{ (Theorem 9)} \\
& \simeq_{\beta_i} & \mathcal{C}_{\rm v}^{+}\langle \mathit{eval}_{\rm v}(\mathcal{T}\langle\!\langle e\rangle\!\rangle)\rangle \\
& & \textit{...}\mathcal{T}\ \textbf{Simulation}\ \textit{(Theorem 5)} \\
& & \quad \textit{and } \mathcal{C}_{\rm v}^{+}\ \textbf{Translation}\ \textit{(Theorem 8)} \\
& \simeq & \mathit{eval}_{\rm n}((\mathcal{C}_{\rm v}^{+} \circ \mathcal{T})\langle\!\langle e\rangle\!\rangle\,(\lambda x.x)) \\
& & \textit{...}\mathcal{C}_{\rm v}^{+}\ \textbf{Simulation}\ \textit{(Theorem 8)} \\
& \simeq & \mathit{eval}_{\rm v}((\mathcal{C}_{\rm v}^{+} \circ \mathcal{T})\langle\!\langle e\rangle\!\rangle\,(\lambda x.x)) \\
& & \textit{...}\mathcal{C}_{\rm v}^{+}\ \textbf{Indifference}\ \textit{(Theorem 8)} \\
& \simeq_{\beta_i} & \mathit{eval}_{\rm v}(\mathcal{C}_{\rm n}\langle\!\langle e\rangle\!\rangle\,(\lambda x.x)) \\
& & \textit{...factoring of } \mathcal{C}_{\rm n} \textit{ (Theorem 9)} \\
& & \quad \textit{and soundness of } \beta_{\rm v} \textit{ (Theorem 1)}
\end{array}$$

A.1.3 Translation

Section 3.3.1 shows that the following portion of **Translation** for $\mathcal{C}_{\rm n}$ can be derived from the correctness properties of $\mathcal{C}_{\rm v}^{+}$ and $\mathcal{T}$. Let $e_1, e_2 \in \Lambda$.

$$\begin{array}{l}
\textit{If } \lambda\beta \vdash e_1 = e_2 \textit{ then } \lambda\beta_{\rm v} \vdash \mathcal{C}_{\rm n}\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}_{\rm n}\langle\!\langle e_2\rangle\!\rangle \\
\textit{Also } \lambda\beta_{\rm v} \vdash \mathcal{C}_{\rm n}\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}_{\rm n}\langle\!\langle e_2\rangle\!\rangle \textit{ iff } \lambda\beta \vdash \mathcal{C}_{\rm n}\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}_{\rm n}\langle\!\langle e_2\rangle\!\rangle \\
\textit{iff } \lambda\beta_{\rm v} \vdash \mathcal{C}_{\rm n}\langle\!\langle e_1\rangle\!\rangle\, I = \mathcal{C}_{\rm n}\langle\!\langle e_2\rangle\!\rangle\, I \textit{ iff } \lambda\beta \vdash \mathcal{C}_{\rm n}\langle\!\langle e_1\rangle\!\rangle\, I = \mathcal{C}_{\rm n}\langle\!\langle e_2\rangle\!\rangle\, I
\end{array}$$

It only remains to show the converse of the first implication. For this, it is sufficient to show $\lambda\beta \vdash \mathcal{C}_{\rm n}\langle\!\langle e_1\rangle\!\rangle\, I = \mathcal{C}_{\rm n}\langle\!\langle e_2\rangle\!\rangle\, I$ implies $\lambda\beta \vdash e_1 = e_2$ which

[25] Suggested by Kristian Nielsen and Morten Heine Sørensen.

follows immediately from Plotkin's original proof for $\mathcal{P}_{\mathrm{n}}$. The relevant section of the proof begins at the second paragraph under "Proof of Theorem 6" [23, p. 158]. One only needs to show that for all $e \in \Lambda$, $e \overset{\circ}{\sim} \mathcal{C}_{\mathrm{n}}\langle\!\langle e \rangle\!\rangle\, I$ (where $\overset{\circ}{\sim}$ is defined by Plotkin) and this follows by a straightforward induction over the structure of e.

A.2 Correctness of $\mathcal{T}$

A.2.1 Preliminaries

The definition of stuck terms is extended from Λ (see Section 1.4.4) to Λ_τ as follows.

$$
\begin{array}{lll}
s & \in & \mathit{Stuck}_{\mathrm{n}}[\Lambda_\tau] \\
s & ::= & \ldots \mid (\mathit{delay}\ e_0)\, e_1 \mid \mathit{force}\ b \mid \mathit{force}\ \lambda x.e \mid \mathit{force}\ s \\
 & & \ldots \mathit{where}\ e, e_0, e_1 \in \Lambda_\tau
\end{array}
$$

$$
\begin{array}{lll}
s & \in & \mathit{Stuck}_{\mathrm{v}}[\Lambda_\tau] \\
s & ::= & \ldots \mid (\mathit{delay}\ e_0)\, e_1 \mid \mathit{force}\ b \mid \mathit{force}\ \lambda x.e \mid \mathit{force}\ s \\
 & & \ldots \mathit{where}\ e, e_0, e_1 \in \Lambda_\tau
\end{array}
$$

A simple induction over the structure of $e \in \mathit{Programs}[\Lambda_\tau]$ shows that either $e \in \mathit{Values}_{\mathrm{n}}[\Lambda_\tau]$, or $e \in \mathit{Stuck}_{\mathrm{n}}[\Lambda_\tau]$, or $e \longmapsto_{\mathrm{n}} e'$ (similarly for call-by-value).

The following properties shows how $\mathcal{T}$ and $\mathcal{T}^{-1}$ interact with substitution.

Property 10 *For all* $e_1, e_2 \in \Lambda$,

$$\mathcal{T}\langle\!\langle e_1 \rangle\!\rangle[x := \mathit{delay}\ \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle] \longrightarrow\!\!\!\!\!\rightarrow_\tau \mathcal{T}\langle\!\langle e_1[x := e_2] \rangle\!\rangle.$$

Proof: by induction over the structure of e_1. The interesting case is...

case $e_1 \equiv x$:

$$
\begin{array}{rcl}
\mathcal{T}\langle\!\langle x \rangle\!\rangle[x := \mathit{delay}\ \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle] & = & (\mathit{force}\ x)[x := \mathit{delay}\ \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle] \\
 & \equiv & \mathit{force}\ (\mathit{delay}\ \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle) \\
 & \longrightarrow_\tau & \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle \\
 & \equiv & \mathcal{T}\langle\!\langle x[x := e_2] \rangle\!\rangle
\end{array}
$$

The rest of the cases follow trivially or by the induction hypothesis and compatibility of reductions. ∎

Property 11 *For all* $t_1, t_2 \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$,

$$\mathcal{T}^{-1}\langle\!\langle t_1 \rangle\!\rangle[x := \mathcal{T}^{-1}\langle\!\langle t_2 \rangle\!\rangle] \equiv \mathcal{T}^{-1}\langle\!\langle t_1[x := \mathit{delay}\ t_2] \rangle\!\rangle.$$

Proof: a simple induction over the structure of t_1. ∎

A.2.2 The language $\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$

This section shows that the language $\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$ (see Section 2.2.2) corresponds to the set of terms T reachable from the image of $\mathcal{T}$ *via* $\beta\tau$ reduction.

$$T \stackrel{\text{def}}{=} \{t \in \Lambda_\tau \mid \exists e \in \Lambda\,.\,\lambda\beta\tau \vdash \mathcal{T}\langle\!\langle e \rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow t\}$$

First, we show that $\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$ is closed under relevant substitutions (Property 12) and under $\beta\tau$ reduction (Property 13).

Property 12 *For all* $t_1, t_2 \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$, $t_1[x := \mathit{delay}\ t_2] \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$.

Proof: by induction over the structure of t_1. The interesting case is...

case $t_1 \equiv \mathit{force}\ x$: $(\mathit{force}\ x)[x := \mathit{delay}\ t_2] = \mathit{force}\ (\mathit{delay}\ t_2) \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$

The other cases are either trivial or follow from the induction hypothesis. ∎

Property 13 *For all* $t \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$, $\lambda\beta\tau \vdash t \longrightarrow t'$ *implies* $t' \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$.

Proof: by induction over the structure of t. It is sufficient to show the following.

case $t \equiv \mathit{force}\ (\mathit{delay}\ t') \longrightarrow_\tau t' \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$
case $t \equiv (\lambda x.t_0)\ (\mathit{delay}\ t_1) \longrightarrow_\beta t_0[x := \mathit{delay}\ t_1] \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$
...since $\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$ is closed under substitution (Property 12)

∎

To show that $\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^* = T$, it is shown that $T \subseteq \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$ and $\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^* \subseteq T$.

Property 14 $T \subseteq \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$.

Proof: Let $t \in T$. From the definition of T there exists an $e \in \Lambda$ such that $\lambda\beta\tau \vdash \mathcal{T}\langle\!\langle e \rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow t$ in n steps. Now we show $t \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$ by induction on n.

case $n = 0$: a simple induction over the structure of e shows $\mathcal{T}\langle\!|e|\!\rangle \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$.
case $n = i + 1$: then $\lambda\beta\tau \vdash \mathcal{T}\langle\!|e|\!\rangle \twoheadrightarrow t \longrightarrow u$. By ind. hyp. $t \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$ and therefore $u \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$ since $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$ is closed under reductions (Property 13).

∎

Property 15 $\mathcal{T}\langle\!|\Lambda|\!\rangle^* \subseteq T$.

Proof: It is required to show that $t \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$ implies $t \in T$, *i.e.*, there exists an $e \in \Lambda$ such that $\lambda\beta\tau \vdash \mathcal{T}\langle\!|e|\!\rangle \twoheadrightarrow t$. The proof is by induction over the structure of t. The interesting case is...

case $t \equiv \mathit{force}\ (\mathit{delay}\ t_0)$:
Since $t_0 \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$, by the induction hypothesis there exists an $e_0 \in \Lambda$ such that $\lambda\beta\tau \vdash \mathcal{T}\langle\!|e_0|\!\rangle \twoheadrightarrow t_0$. So take $e \equiv (\lambda x.x)\, e_0$.

∎

The stuck terms of $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$ are defined as follows.

$$
\begin{array}{lcllcl}
s & \in & \mathit{Stuck}_{\mathrm{n}}[\mathcal{T}\langle\!|\Lambda|\!\rangle^*] & s & \in & \mathit{Stuck}_{\mathrm{v}}[\mathcal{T}\langle\!|\Lambda|\!\rangle^*] \\
s & ::= & b\,(\mathit{delay}\ t) \mid s\,(\mathit{delay}\ t) & s & ::= & b\,(\mathit{delay}\ t) \mid s\,(\mathit{delay}\ t)
\end{array}
$$

...where $t \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$.

A simple induction over the structure of $t \in \mathit{Programs}[\mathcal{T}\langle\!|\Lambda|\!\rangle^*]$ shows that either $t \in \mathit{Values}_{\mathrm{n}}[\Lambda_\tau]$, or $t \in \mathit{Stuck}_{\mathrm{n}}[\mathcal{T}\langle\!|\Lambda|\!\rangle^*]$, or $t \longmapsto_{\mathrm{n}} t'$ (similarly for call-by-value). Note that improper uses of *delay* and *force* (*e.g.*, $(\mathit{delay}\ e)\, b$, $\mathit{force}\, \lambda x.e$) which were present in the stuck terms of Λ_τ (see Appendix A.2.1) do not occur in $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$. Intuitively, since $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$ simulates call-by-name evaluation, the stuck terms of $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$ parallel $\mathit{Stuck}_{\mathrm{n}}[\Lambda]$ (see Section 1.4.4).

A.2.3 Simulation

The proof for $\mathcal{T}$ **Simulation** proceeds as outlined in Section 2.3.2. The following property shows that all $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$ terms related by $\overset{\mathcal{T}}{\sim}$ to a certain Λ term are τ equivalent.

Property 16 *For all* $e \in \Lambda$ *and for all* $t \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$ *such that* $e \overset{\mathcal{T}}{\sim} t$, $\lambda\tau \vdash \mathcal{T}\langle\!|e|\!\rangle = t$.

Proof: by induction over the derivation of $e \overset{\mathcal{T}}{\sim} t$. ∎

The following property shows how the relation $\stackrel{\mathcal{T}}{\sim}$ interacts with substitution.

Property 17 *For all* $e_0, e_1 \in \Lambda$ *and* $t_0, t_1 \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$,

$$e_0 \stackrel{\mathcal{T}}{\sim} t_0 \;\wedge\; e_1 \stackrel{\mathcal{T}}{\sim} t_1 \quad\Rightarrow\quad e_0[x := e_1] \stackrel{\mathcal{T}}{\sim} t_0[x := \mathit{delay}\, t_1]$$

The following property expresses that evaluation of thunked terms may involve moving past initial τ reductions.

Property 18 *For all* $e \in \mathit{Programs}[\Lambda]$ *and* $t \in \mathit{Programs}[\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*]$ *such that* $e \stackrel{\mathcal{T}}{\sim} t$,

$$\begin{array}{lllll} e \equiv & b & \stackrel{\mathcal{T}}{\sim} t \Rightarrow t \longmapsto_{\mathrm{v}}^{*} b & & \\ e \equiv & \lambda x.e_0 & \stackrel{\mathcal{T}}{\sim} t \Rightarrow t \longmapsto_{\mathrm{v}}^{*} \lambda x.t_0 & \mathit{where}\ e_0 \stackrel{\mathcal{T}}{\sim} t_0 & \\ e \equiv & e_0\, e_1 & \stackrel{\mathcal{T}}{\sim} t \Rightarrow t \longmapsto_{\mathrm{v}}^{*} t_0\, (\mathit{delay}\, t_1) & \mathit{where}\ e_0 \stackrel{\mathcal{T}}{\sim} t_0 & \mathit{and}\ e_1 \stackrel{\mathcal{T}}{\sim} t_1 \end{array}$$

Proof: by induction over the derivation of $e \stackrel{\mathcal{T}}{\sim} t$. We show only the cases necessary for the second component (the others are similar).

case $\stackrel{\mathcal{T}}{\sim}.3$: $\lambda x.e_0 \stackrel{\mathcal{T}}{\sim} \lambda x.t_0$ because $e_0 \stackrel{\mathcal{T}}{\sim} t_0$: immediate
case $\stackrel{\mathcal{T}}{\sim}.5$: $\lambda x.e_0 \stackrel{\mathcal{T}}{\sim} \mathit{force}\,(\mathit{delay}\, t')$ because $\lambda x.e_0 \stackrel{\mathcal{T}}{\sim} t'$:

$$\begin{array}{llll} \mathit{force}\,(\mathit{delay}\, t') & \longmapsto_{\mathrm{v}} t' & & \\ & \longmapsto_{\mathrm{v}}^{*} \lambda x.t_0 & \text{where } e_0 \stackrel{\mathcal{T}}{\sim} t_0 & \ldots\mathit{by\ ind.\ hyp.} \end{array}$$

∎

The following property (corresponding to Property 1 of Section 2.3.2) states that each $\longmapsto_{\mathrm{n}}$ step on a Λ program implies corresponding $\longmapsto_{\mathrm{v}}$ steps on a thunked program.

Property 19 ($\mathcal{T}$ — one step simulation)
For all $e_0, e_1 \in \mathit{Programs}[\Lambda]$ *and* $t_0 \in \mathit{Programs}[\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*]$ *such that* $e_0 \stackrel{\mathcal{T}}{\sim} t_0$,

$$e_0 \longmapsto_{\mathrm{n}} e_1 \quad\Rightarrow\quad \exists t_1 \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^* \,.\; t_0 \longmapsto_{\mathrm{v}}^{+} t_1 \;\wedge\; e_1 \stackrel{\mathcal{T}}{\sim} t_1$$

Proof: by induction over the derivation of $e_0 \longmapsto_{\mathrm{n}} e_1$. The proof uses Property 18 extensively.

case $(\lambda x.e_a)\, e_b \longmapsto_{\mathrm{n}} e_a[x := e_b]$:

$$\begin{array}{lll} t_0 & \longmapsto_{\mathrm{v}}^{*} t'_a\,(\mathit{delay}\, t_b) & \text{where } \lambda x.e_a \stackrel{\mathcal{T}}{\sim} t'_a \text{ and } e_b \stackrel{\mathcal{T}}{\sim} t_b \\ & \longmapsto_{\mathrm{v}}^{*} (\lambda x.t_a)\,(\mathit{delay}\, t_b) & \text{where } e_a \stackrel{\mathcal{T}}{\sim} t_a \\ & \longmapsto_{\mathrm{v}} t_a[x := \mathit{delay}\, t_b] & \end{array}$$

and $e_a[x := e_b] \stackrel{\mathcal{T}}{\sim} t_a[x := \mathit{delay}\, t_b]$ by Property 17.

case $e_a\, e_b \longmapsto_{\mathrm{n}} e'_a\, e_b$ because $e_a \longmapsto_{\mathrm{n}} e'_a$:

$$\begin{array}{lllll} t_0 & \longmapsto_{\mathrm{v}}^{*} & t_a\,(\mathit{delay}\; t_b) & \text{where } e_a \overset{\mathcal{T}}{\sim} t_a \text{ and } e_b \overset{\mathcal{T}}{\sim} t_b & \\ & \longmapsto_{\mathrm{v}}^{+} & t'_a\,(\mathit{delay}\; t_b) & \text{where } e'_a \overset{\mathcal{T}}{\sim} t'_a & \textit{...by ind. hyp.} \end{array}$$

and $e'_a\, e_b \overset{\mathcal{T}}{\sim} t'_a\,(\mathit{delay}\; t_b)$ by rule $\overset{\mathcal{T}}{\sim}$.5.

∎

The following property states that if a Λ program is stuck under $\mathit{eval}_{\mathrm{n}}$, then all $\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$ programs related to it by $\overset{\mathcal{T}}{\sim}$ will reach a stuck program under $\mathit{eval}_{\mathrm{v}}$.

Property 20 ($\mathcal{T}$ — coincidence of stuck terms)
For all programs $s \in \mathit{Stuck}_{\mathrm{n}}[\Lambda]$ and $t \in \mathit{Programs}[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*]$,

$$s \overset{\mathcal{T}}{\sim} t \;\Rightarrow\; \exists t' \in \mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^* \,.\, t \longmapsto_{\mathrm{v}}^{*} t' \;\wedge\; t' \in \mathit{Stuck}_{\mathrm{n}}[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*] = \mathit{Stuck}_{\mathrm{v}}[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*]$$

Proof: First, note that

$$\mathit{Stuck}_{\mathrm{n}}[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*] = \mathit{Stuck}_{\mathrm{v}}[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*]$$

(see Appendix A.2.1). The proof then proceeds by induction over the structure of $s \in \mathit{Stuck}_{\mathrm{n}}[\Lambda]$ (see Section 1.4.4) appealing to Property 18 at each step.

case $s \equiv b\, e_1$:

$$\begin{array}{llll} t & \longmapsto_{\mathrm{v}}^{*} & t_b\,(\mathit{delay}\; t_1) & \text{where } b \overset{\mathcal{T}}{\sim} t_b \text{ and } e_1 \overset{\mathcal{T}}{\sim} t_1 \\ & \longmapsto_{\mathrm{v}}^{*} & b\,(\mathit{delay}\; t_1) & \\ & \in & \mathit{Stuck}_{\mathrm{n}}[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*] & \end{array}$$

case $s \equiv s_0\, e_1$:

$$\begin{array}{lllll} t & \longmapsto_{\mathrm{v}}^{*} & t_0\,(\mathit{delay}\; t_1) & \text{where } s_0 \overset{\mathcal{T}}{\sim} t_0 \text{ and } e_1 \overset{\mathcal{T}}{\sim} t_1 & \\ & \longmapsto_{\mathrm{v}}^{*} & t_s\,(\mathit{delay}\; t_1) & \text{where } t_s \in \mathit{Stuck}_{\mathrm{n}}[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*] & \textit{...by ind. hyp.} \\ & \in & \mathit{Stuck}_{\mathrm{n}}[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*] & & \end{array}$$

∎

Lemma 1 ($\mathcal{T}$ — simulation) *For all $e \in \mathit{Programs}[\Lambda]$,*

$$\mathcal{T}\langle\!\langle \mathit{eval}_{\mathrm{n}}(e)\rangle\!\rangle \simeq_{\mathcal{T}} \mathit{eval}_{\mathrm{v}}(\mathcal{T}\langle\!\langle e\rangle\!\rangle)$$

Proof:

case $eval_n(e)$ is defined, *i.e.*, $e \longmapsto_n^* v$:

Since $e \overset{\mathcal{T}}{\sim} \mathcal{T}\langle\!\langle e \rangle\!\rangle$, an induction over the number of steps in $e \longmapsto_n^* v$ (applying Property 19) gives $\mathcal{T}\langle\!\langle e \rangle\!\rangle \longmapsto_v^* t$ where $v \overset{\mathcal{T}}{\sim} t$. Now by Property 18, one can see that for all closed values $v \in Values_n[\Lambda]$, $t \longmapsto_v^* w$ where $w \in Values_v[\Lambda_\tau]$ and $v \overset{\mathcal{T}}{\sim} w$. Finally, $\lambda\tau \vdash \mathcal{T}\langle\!\langle v \rangle\!\rangle = w$ by Property 16.

case $eval_n(e)$ is undefined: we have two cases:

case e heads an infinite sequence $e = e_1 \longmapsto_n e_2 \longmapsto_n ...$:
Since $e \overset{\mathcal{T}}{\sim} \mathcal{T}\langle\!\langle e \rangle\!\rangle$, applying Property 19 repeatedly gives an infinite sequence $\mathcal{T}\langle\!\langle e \rangle\!\rangle = t_1 \longmapsto_v^+ t_2 \longmapsto_v^+ ...$ and so $eval_v(\mathcal{T}\langle\!\langle e \rangle\!\rangle)$ is undefined as well.
case $e \longmapsto_n^* s \in Stuck_n[\Lambda]$:
Since $e \overset{\mathcal{T}}{\sim} \mathcal{T}\langle\!\langle e \rangle\!\rangle$, applying Property 19 repeatedly gives $\mathcal{T}\langle\!\langle e \rangle\!\rangle \longmapsto_v^* t$ where $s \overset{\mathcal{T}}{\sim} t$. By Property 20, $t \longmapsto_v^* t_s \in Stuck_v[\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*]$ and so $eval_v(\mathcal{T}\langle\!\langle e \rangle\!\rangle)$ is undefined as well.

∎

A.2.4 Translation

This section establishes the equational correspondence (Theorem 6) between the language Λ under theory $\lambda\beta$ and language $\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$ under theory $\lambda\beta_i\tau$. This is sufficient for establishing the **Translation** property for $\mathcal{T}$ (Theorem 5).

Components 1 and 2 of Theorem 6 follow from Properties 2 and 3 of Section 2.3.3. Both of these properties follow from simple structural inductions and the proofs are omitted.

The following property (corresponding to Property 4 of Section 2.3.3) states that each reduction on source terms corresponds to one or more reductions on thunked terms.

Property 21 *For all* $e_1, e_2 \in \Lambda$*,*

$$\lambda\beta \vdash e_1 \longrightarrow e_2 \Rightarrow \lambda\beta_i\tau \vdash \mathcal{T}\langle\!\langle e_1 \rangle\!\rangle \twoheadrightarrow \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle$$

Proof: It is sufficient to show the following:

case $\lambda\beta \vdash (\lambda x.e_1)\, e_2 \longrightarrow e_1[x := e_2]$:

$$\begin{array}{lll} \mathcal{T}\langle\!\langle (\lambda x.e_1)\, e_2 \rangle\!\rangle & = & (\lambda x.\mathcal{T}\langle\!\langle e_1 \rangle\!\rangle)\,(\mathit{delay}\; \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle) \\ & \longrightarrow_{\beta_i} & \mathcal{T}\langle\!\langle e_1 \rangle\!\rangle[x := \mathit{delay}\; \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle] \\ & \twoheadrightarrow_{\tau} & \mathcal{T}\langle\!\langle e_1[x := e_2] \rangle\!\rangle \quad \textit{...by Property 10} \end{array}$$

$\blacksquare$

The following property (corresponding to Property 5 of Section 2.3.3) shows that each reduction on thunked terms corresponds to zero or one reduction on source terms.

Property 22 *For all* $t_1, t_2 \in \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$,

$$\lambda\beta_i\tau \vdash t_1 \longrightarrow t_2 \;\Rightarrow\; \lambda\beta \vdash \mathcal{T}^{-1}\langle\!\langle t_1 \rangle\!\rangle \twoheadrightarrow \mathcal{T}^{-1}\langle\!\langle t_2 \rangle\!\rangle$$

Proof: It is sufficient to show the following:

case $\lambda\beta_i\tau \vdash \mathit{force}\,(\mathit{delay}\,t) \longrightarrow t$:

$$\begin{array}{lcl} \mathcal{T}^{-1}\langle\!\langle \mathit{force}\,(\mathit{delay}\,t) \rangle\!\rangle & = & \mathcal{T}^{-1}\langle\!\langle \mathit{delay}\,t \rangle\!\rangle \\ & = & \mathcal{T}^{-1}\langle\!\langle t \rangle\!\rangle \quad \ldots\textit{by definition of } \mathcal{T}^{-1} \end{array}$$

case $\lambda\beta_i\tau \vdash (\lambda x.t_1)\,(\mathit{delay}\,t_2) \twoheadrightarrow t_1[x := \mathit{delay}\,t_2]$:

$$\begin{array}{lcl} \mathcal{T}^{-1}\langle\!\langle (\lambda x.t_1)\,(\mathit{delay}\,t_2) \rangle\!\rangle & = & (\lambda x.\mathcal{T}^{-1}\langle\!\langle t_1 \rangle\!\rangle)\,\mathcal{T}^{-1}\langle\!\langle t_2 \rangle\!\rangle \\ & \longrightarrow_\beta & \mathcal{T}^{-1}\langle\!\langle t_1 \rangle\!\rangle[x := \mathcal{T}^{-1}\langle\!\langle t_2 \rangle\!\rangle] \\ & \equiv & \mathcal{T}^{-1}\langle\!\langle t_1[x := \mathit{delay}\,t_2] \rangle\!\rangle \\ & & \ldots\textit{by Property 11.} \end{array}$$

$\blacksquare$

Components 3 and 4 of the equational correspondence (Theorem 6) are now proved as follows:

$$\begin{array}{rl} (1) \quad \lambda\beta \vdash e_1 \twoheadrightarrow e_2 \;\Rightarrow & \lambda\beta_i\tau \vdash \mathcal{T}\langle\!\langle e_1 \rangle\!\rangle \twoheadrightarrow \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle \\ & \ldots\textit{by ind. on \# of reductions} \\ & \quad \textit{and Prop. 21.} \\ (2) \quad \lambda\beta \vdash e_1 = e_2 \;\Rightarrow & \lambda\beta_i\tau \vdash \mathcal{T}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle \\ & \ldots\textit{by Church-Rosser and } (1). \\ (3) \; \lambda\beta_i\tau \vdash t_1 \twoheadrightarrow t_2 \;\Rightarrow & \lambda\beta \vdash \mathcal{T}^{-1}\langle\!\langle t_1 \rangle\!\rangle \twoheadrightarrow \mathcal{T}^{-1}\langle\!\langle t_2 \rangle\!\rangle \\ & \ldots\textit{by ind. on \# of reductions} \\ & \quad \textit{and Prop. 22.} \\ (4) \quad \lambda\beta_i\tau \vdash t_1 = t_2 \;\Rightarrow & \lambda\beta \vdash \mathcal{T}^{-1}\langle\!\langle t_1 \rangle\!\rangle = \mathcal{T}^{-1}\langle\!\langle t_2 \rangle\!\rangle \\ & \ldots\textit{by Church-Rosser and } (3). \end{array}$$

(5) $\lambda\beta_i\tau \vdash \mathcal{T}\langle\!\langle e_1\rangle\!\rangle = \mathcal{T}\langle\!\langle t_2\rangle\!\rangle \Rightarrow \lambda\beta \vdash (\mathcal{T}^{-1}\circ\mathcal{T})\langle\!\langle e_1\rangle\!\rangle = (\mathcal{T}^{-1}\circ\mathcal{T})\langle\!\langle e_2\rangle\!\rangle$
...*by* (4).

(6) $\Rightarrow \lambda\beta \vdash e_1 = e_2$
...*by Prop. 2.*

(7) $\lambda\beta \vdash \mathcal{T}^{-1}\langle\!\langle t_1\rangle\!\rangle = \mathcal{T}^{-1}\langle\!\langle t_2\rangle\!\rangle \Rightarrow \lambda\beta_i\tau \vdash (\mathcal{T}\circ\mathcal{T}^{-1})\langle\!\langle t_1\rangle\!\rangle = (\mathcal{T}\circ\mathcal{T}^{-1})\langle\!\langle t_2\rangle\!\rangle$
...*by* (2).

(8) $\Rightarrow \lambda\beta_i\tau \vdash t_1 = t_2$
...*by Prop. 3.*

A.3 Correctness of $\mathcal{T}_{\mathcal{L}}$

A.3.1 The language $\mathcal{T}_{\mathcal{L}}\langle\!\langle\Lambda\rangle\!\rangle^*$

The language of terms $\mathcal{T}_{\mathcal{L}}\langle\!\langle\Lambda\rangle\!\rangle^*$ in the image of $\mathcal{T}_{\mathcal{L}}$ closed under β_i reduction is as follows ($z \notin FV(t)$ in the third clause and $z \notin FV(t_1)$ in the fifth clause).

$$\begin{array}{rcl} t & \in & Terms[\mathcal{T}_{\mathcal{L}}\langle\!\langle\Lambda\rangle\!\rangle^*] \\ t & ::= & b \ \mid \ x\,b \ \mid \ (\lambda z.t)\,b \ \mid \ \lambda x.t \ \mid \ t_0\,(\lambda z.t_1) \end{array}$$

The proofs of correctness for the grammar are similar to the proofs of correctness for $\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$ given in Section A.2.2.

A.3.2 Indifference and Simulation

To prove **Indifference** and **Simulation** for $\mathcal{T}_{\mathcal{L}}$, we take advantage of the fact that $\mathcal{T}_{\mathcal{L}} = \mathcal{L}\circ\mathcal{T}$. The following three properties (which are straightforward to prove) are sufficient for establishing **Indifference** and **Simulation** properties for $\mathcal{L}$ (see Lemma 2 below).

Property 23 ($\mathcal{L}$ — commutation with substitution)
For all $e \in \Lambda_\tau$, $\mathcal{L}\langle\!\langle e_0[x := e_1]\rangle\!\rangle \equiv \mathcal{L}\langle\!\langle e_0\rangle\!\rangle[x := \mathcal{L}\langle\!\langle e_1\rangle\!\rangle]$

Property 24 ($\mathcal{L}$ — one step indifference and simulation)
For all $e_0, e_1 \in Programs[\Lambda_\tau]$,

$$e_0 \longmapsto_{\mathrm{v}} e_1 \;\Rightarrow\; \mathcal{L}\langle\!\langle e_0\rangle\!\rangle \longmapsto_i \mathcal{L}\langle\!\langle e_1\rangle\!\rangle$$

Property 25 ($\mathcal{L}$ — coincidence of stuck terms)
For all $s \in Stuck_{\mathrm{v}}[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*]$, $\mathcal{L}\langle\!\langle s\rangle\!\rangle \in Stuck_{\mathrm{n}}[\Lambda] \subset Stuck_{\mathrm{v}}[\Lambda]$.

$$
\begin{array}{rcl}
\mathcal{L}^{-1} & : & \mathit{Terms}[\mathcal{T}_{\mathcal{L}}\langle\!|\Lambda|\!\rangle^*] \to \mathit{Terms}[\mathcal{T}\langle\!|\Lambda|\!\rangle^*] \\
\mathcal{L}^{-1}\langle\!|b|\!\rangle & = & b \\
\mathcal{L}^{-1}\langle\!|x\,b|\!\rangle & = & \mathit{force}\ x \\
\mathcal{L}^{-1}\langle\!|(\lambda z.t)\,b|\!\rangle & = & \mathit{force}\ (\mathit{delay}\ \mathcal{L}^{-1}\langle\!|t|\!\rangle) \\
\mathcal{L}^{-1}\langle\!|\lambda x.t|\!\rangle & = & \lambda x.\mathcal{L}^{-1}\langle\!|t|\!\rangle \\
\mathcal{L}^{-1}\langle\!|t_0\,(\lambda z.t_1)|\!\rangle & = & \mathcal{L}^{-1}\langle\!|t_0|\!\rangle\ (\mathit{delay}\ \mathcal{L}^{-1}\langle\!|t_1|\!\rangle)
\end{array}
$$

Figure 17: Mapping Λ thunks to abstract Λ_τ thunks

Lemma 2 ($\mathcal{L}$ — Indifference and Simulation)
For all $t \in \mathit{Programs}[\mathcal{T}_{\mathcal{L}}\langle\!|\Lambda|\!\rangle^*]$,

$$\mathcal{L}\langle\!|\mathit{eval}_{\mathrm{v}}(t)|\!\rangle \simeq \mathit{eval}_i(\mathcal{L}\langle\!|t|\!\rangle)$$

Proof: The proof follows from Properties 23, 24, and 25 and has the same structure as the proof for $\mathcal{T}$ **Simulation** (Lemma 1, Appendix A.2.3). ∎

Now, **Simulation** for $\mathcal{T}_{\mathcal{L}}$ is proved as follows (the proof of **Indifference** is similar).

$$
\begin{array}{rcl}
\mathcal{T}_{\mathcal{L}}\langle\!|\mathit{eval}_{\mathrm{n}}(e)|\!\rangle & \simeq & (\mathcal{L}\circ\mathcal{T})\langle\!|\mathit{eval}_{\mathrm{n}}(e)|\!\rangle \\
& \simeq_{\beta_i} & \mathcal{L}\langle\!|\mathit{eval}_{\mathrm{v}}(\mathcal{T}\langle\!|e|\!\rangle)|\!\rangle \\
& & \ldots\mathcal{T}\ \textbf{Simulation}\ \textit{(Theorem 5) and Theorem 11} \\
& \simeq & \mathit{eval}_{\mathrm{v}}((\mathcal{L}\circ\mathcal{T})\langle\!|e|\!\rangle) \\
& & \ldots\mathcal{L}\ \textbf{Simulation}\ \textit{(Lemma 2)} \\
& \simeq & \mathit{eval}_{\mathrm{v}}(\mathcal{T}_{\mathcal{L}}\langle\!|e|\!\rangle)
\end{array}
$$

A.3.3 Translation

For **Translation** for $\mathcal{T}_{\mathcal{L}}$, we again take advantage of the fact that $\mathcal{T}_{\mathcal{L}} = \mathcal{L}\circ\mathcal{T}$ and show that $\mathcal{L}$ and its inverse $\mathcal{L}^{-1}$ (given in Figure 17) establish an equational correspondence between $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$ and $\mathcal{T}_{\mathcal{L}}\langle\!|\Lambda|\!\rangle^*$.

Theorem 11 (Equational Correspondence for $\mathcal{T}_{\mathcal{L}}$)
For all $e, e_1, e_2 \in \mathit{Terms}[\mathcal{T}\langle\!|\Lambda|\!\rangle^*]$ and $t, t_1, t_2 \in \mathit{Terms}[\mathcal{T}_{\mathcal{L}}\langle\!|\Lambda|\!\rangle^*]$,

1. $\lambda\beta_i\tau \vdash e = (\mathcal{L}^{-1} \circ \mathcal{L})\langle\!\langle e \rangle\!\rangle$
2. $\lambda\beta_i \vdash t = (\mathcal{L} \circ \mathcal{L}^{-1})\langle\!\langle t \rangle\!\rangle$
3. $\lambda\beta_i\tau \vdash e_1 = e_2$ *iff* $\lambda\beta_i \vdash \mathcal{L}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{L}\langle\!\langle e_2 \rangle\!\rangle$
4. $\lambda\beta_i \vdash t_1 = t_2$ *iff* $\lambda\beta_i\tau \vdash \mathcal{L}^{-1}\langle\!\langle t_1 \rangle\!\rangle = \mathcal{L}^{-1}\langle\!\langle t_2 \rangle\!\rangle$

The proofs for the equational correspondence mirror those of the equational correspondence for $\mathcal{T}$ (Theorem 6 — see Sections 2.3.3 and A.2.4) and are easy to establish. Now, given the equational correspondences established by $\mathcal{T}$ (Theorem 6) and $\mathcal{L}$ (Theorem 11), **Translation** for $\mathcal{T}_{\mathcal{L}}$ is proved as follows.

$$\begin{array}{lll} & \lambda\beta \vdash e_1 = e_2 & \\ \Leftrightarrow & \lambda\beta_i\tau \vdash \mathcal{T}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle & \textit{...Theorem 6 (Component 3)} \\ \Leftrightarrow & \lambda\beta_i \vdash (\mathcal{L} \circ \mathcal{T})\langle\!\langle e_1 \rangle\!\rangle = (\mathcal{L} \circ \mathcal{T})\langle\!\langle e_2 \rangle\!\rangle & \textit{...Theorem 11 (Component 3)} \\ \Leftrightarrow & \lambda\beta_i \vdash \mathcal{T}_{\mathcal{L}}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{T}_{\mathcal{L}}\langle\!\langle e_2 \rangle\!\rangle & \textit{...definition of } \mathcal{T}_{\mathcal{L}} \end{array}$$

A.4 Correctness of $\mathcal{C}_{\mathrm{v}}^{+}$

A.4.1 Indifference and Simulation

Following Plotkin's proofs for $\mathcal{C}_{\mathrm{v}}$ [23, pp. 149–152], **Indifference** and **Simulation** for $\mathcal{C}_{\mathrm{v}}^{+}$ on $\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$ are proved simultaneously. The proofs only involve minor extensions to Plotkin's original proofs and we summarize only the differences.[26]

First, we show that $\mathcal{C}_{\mathrm{v}}^{+}$ commutes with substitution.

Property 26 ($\mathcal{C}_{\mathrm{v}}^{+}$ — commutation with substitution)
For all $e \in \Lambda_\tau$ and $v \in Values_{\mathrm{v}}[\Lambda_\tau]$,

$$\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e[x := v] \rangle\!\rangle \equiv \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle[x := \mathcal{C}_{\mathrm{v}}^{+}\langle v \rangle]$$

Proof: by induction of the structure of e. To extend Plotkin's proof, we need only consider the cases where $e \equiv$ *delay* e' and $e \equiv$ *force* e'. These follow immediately from the inductive hypothesis. ∎

Next, we extend Plotkin's colon translation for $\mathcal{C}_{\mathrm{v}}$ [23, p. 150] to $\mathcal{C}_{\mathrm{v}}^{+}$, *i.e.*, to handle *delay* (which is included in the cases for values v) and *force*.

[26] Note that Properties 26, 27, and 28 are stronger than necessary since they are stated for $\Lambda_\tau \supset \mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*$.

Definition 4 ($\mathcal{C}_{\mathrm{v}}^{+}$ — colon translation)
For all closed $v, v_0, v_1 \in Values_{\mathrm{v}}[\Lambda_\tau]$, closed non-values $e, e_0, e_1 \in \Lambda_\tau$, closed $e_1' \in \Lambda_\tau$, and for all closed $\kappa \in Values_{\mathrm{n}}[\Lambda]$,

$$\begin{array}{rcl}
v \;:\; \kappa & = & \kappa\, \mathcal{C}_{\mathrm{v}}^{+}\langle v\rangle \\
e_0\, e_1' \;:\; \kappa & = & e_0 \;:\; (\lambda y_0.\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e_1'\rangle\!\rangle\,(\lambda y_1.y_0\, y_1\, \kappa)) \\
v_0\, e_1 \;:\; \kappa & = & e_1 \;:\; (\lambda y_1.\mathcal{C}_{\mathrm{v}}^{+}\langle v_0\rangle\, y_1\, \kappa) \\
v_0\, v_1 \;:\; \kappa & = & \mathcal{C}_{\mathrm{v}}^{+}\langle v_0\rangle\, \mathcal{C}_{\mathrm{v}}^{+}\langle v_1\rangle\, \kappa \\
\mathit{force}\; e \;:\; \kappa & = & e \;:\; (\lambda y.y\, \kappa) \\
\mathit{force}\; v \;:\; \kappa & = & \mathcal{C}_{\mathrm{v}}^{+}\langle v\rangle\, \kappa
\end{array}$$

Property 27 ($\mathcal{C}_{\mathrm{v}}^{+}$ — correctness of colon translation)
For all $e \in \Lambda_\tau$ and closed $\kappa \in Values_{\mathrm{n}}[\Lambda]$,

$$\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e\rangle\!\rangle\, \kappa \longmapsto_i^{+} e \;:\; \kappa$$

Proof: We only show the cases for *delay* and *force*. The remaining cases are identical to Plotkin's proof for $\mathcal{C}_{\mathrm{v}}$ [23, p. 150].

case $e \equiv \mathit{delay}\; e_0$:

$$\begin{array}{rcl}
\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle \mathit{delay}\; e_0\rangle\!\rangle\, \kappa & = & (\lambda k.k\, \mathcal{C}_{\mathrm{v}}^{+}\langle \mathit{delay}\; e_0\rangle)\, \kappa \\
& \longmapsto_i & \kappa\, \mathcal{C}_{\mathrm{v}}^{+}\langle \mathit{delay}\; e_0\rangle \\
& = & \mathit{delay}\; e_0 \;:\; \kappa
\end{array}$$

case $e \equiv \mathit{force}\; e_0$:

$$\begin{array}{rcl}
\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle \mathit{force}\; e_0\rangle\!\rangle\, \kappa & = & (\lambda k.\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e_0\rangle\!\rangle\,(\lambda y.y\, k))\, \kappa \\
& \longmapsto_i & \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e_0\rangle\!\rangle\,(\lambda y.y\, \kappa) \\
& \longmapsto_i^{+} & e_0 \;:\; (\lambda y.y\, \kappa) \\
& & \ldots \mathit{by\ ind.\ hyp.\ and\ call\ this\ term}\ z
\end{array}$$

If $e_0 \notin Values_{\mathrm{v}}[\Lambda]$, then

$$z = \mathit{force}\; e_0 \;:\; \kappa$$

If $e_0 \in Values_{\mathrm{v}}[\Lambda]$, then

$$\begin{array}{rcl}
z & = & (\lambda y.y\, \kappa)\, \mathcal{C}_{\mathrm{v}}^{+}\langle e_0\rangle \\
& \longmapsto_i & \mathcal{C}_{\mathrm{v}}^{+}\langle e_0\rangle\, \kappa \\
& = & \mathit{force}\; e_0 \;:\; \kappa
\end{array}$$

∎

The following property states that one $\longmapsto_{\mathrm{v}}$ step in Λ_τ implies one or more $\longmapsto_i$ steps on CPS programs.

Property 28 ($\mathcal{C}_{\mathrm{v}}^{+}$ — one step simulation)
For all $e_0, e_1 \in$ Programs$[\Lambda_\tau]$ and closed $\kappa \in$ Values$_{\mathrm{n}}[\Lambda]$,

$$e_0 \longmapsto_{\mathrm{v}} e_1 \;\Rightarrow\; e_0 : \kappa \longmapsto_i^{+} e_1 : \kappa$$

Proof: by induction over the derivation of $e_0 \longmapsto_{\mathrm{v}} e_1$. We only show the cases for *delay* and *force*. The remaining cases are identical to Plotkin's proofs for $\mathcal{C}_{\mathrm{v}}$ [23, p. 151].

case $\mathit{force}\,(\mathit{delay}\; e) \longmapsto_{\mathrm{v}} e$:

$$\begin{array}{rcl} \mathit{force}\,(\mathit{delay}\; e) : \kappa & = & \mathcal{C}_{\mathrm{v}}^{+}\langle \mathit{delay}\; e\rangle\,\kappa \\ & = & \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e\rangle\!\rangle\,\kappa \\ & \longmapsto_i^{+} & e : \kappa \quad \textit{...by Property 27} \end{array}$$

case $\mathit{force}\; e \longmapsto_{\mathrm{v}} \mathit{force}\; e'$ because $e \longmapsto_{\mathrm{v}} e'$:

$$\begin{array}{rcl} \mathit{force}\; e : \kappa & = & e : (\lambda y.y\,\kappa) \\ & \longmapsto_i^{+} & e' : (\lambda y.y\,\kappa) \quad \textit{...by ind. hyp. and call this term } z \end{array}$$

If $e' \notin \mathit{Values}_{\mathrm{v}}[\Lambda_\tau]$, then

$$z \;=\; \mathit{force}\; e' : \kappa$$

If $e' \in \mathit{Values}_{\mathrm{v}}[\Lambda_\tau]$, then

$$\begin{array}{rcl} z & = & (\lambda y.y\,\kappa)\,\mathcal{C}_{\mathrm{v}}^{+}\langle e'\rangle \\ & \longmapsto_i & \mathcal{C}_{\mathrm{v}}^{+}\langle e'\rangle\,\kappa \\ & = & \mathit{force}\; e' : \kappa \end{array}$$

∎

The following property states that if a $\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$ program is stuck under *eval*$_{\mathrm{v}}$, then its CPS image will reach a stuck program under *eval*$_i$.[27]

Property 29 ($\mathcal{C}_{\mathrm{v}}^{+}$ — coincidence of stuck terms)
For all $s \in \mathit{Stuck}_{\mathrm{v}}[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*]$ and all closed $\kappa \in \mathit{Values}_{\mathrm{n}}[\Lambda]$, $s : \kappa \in \mathit{Stuck}_{\mathrm{n}}[\Lambda] \subset \mathit{Stuck}_{\mathrm{v}}[\Lambda]$.

Proof: First note that $\mathit{Stuck}_{\mathrm{n}}[\Lambda] \subset \mathit{Stuck}_{\mathrm{v}}[\Lambda]$. The proof then proceeds by induction over the definition of $s \in \mathit{Stuck}_{\mathrm{v}}[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*]$ (see Appendix A.2.2).

case $s \equiv b\,(\mathit{delay}\; t)$:

$b\,(\mathit{delay}\; t) : \kappa = b\,\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t\rangle\!\rangle\,\kappa$ and since $b\,\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t\rangle\!\rangle \in \mathit{Stuck}_{\mathrm{n}}[\Lambda]$ then $b\,\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t\rangle\!\rangle\,\kappa \in \mathit{Stuck}_{\mathrm{n}}[\Lambda]$.

[27] Note that this doesn't hold for Λ_τ (see footnote 19).

case $s \equiv s_0\,(\mathit{delay}\ t)$:

$s_0\,(\mathit{delay}\ t)\ :\ \kappa\ =\ s_0\ :\ (\lambda y_0.\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle \mathit{delay}\ t_1 \rangle\!\rangle\,(\lambda y_1.y_0\,t_1\,\kappa)) \in \mathit{Stuck}_{\mathrm{n}}[\Lambda]$ by ind. hyp.

∎

Given Properties 27, 28 and 29, the proof of **Indifference** and **Simulation** for $\mathcal{C}_{\mathrm{v}}^{+}$ on $\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^{*}$ (Theorem 8) follow exactly Plotkin's proof for $\mathcal{C}_{\mathrm{v}}$ [23, p. 152]. The proof is similar in structure to the proof of **Simulation** for $\mathcal{T}$ (see Appendix A.2.3).

References

[1] Torben Amtoft. Minimal thunkification. In Patrick Cousot, Moreno Falaschi, Gilberto Filè, and Antoine Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis WSA'93*, volume 724 of *Lecture Notes in Computer Science*, pages 218–229, Padova, Italy, September 1993.

[2] Andrea Asperti. A categorical understanding of environment machines. *Journal of Functional Programming*, 2(1):23–59, January 1992.

[3] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North-Holland, 1984.

[4] Henk Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabby, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 2*, chapter 2, pages 118–309. Oxford University Press, Oxford, 1992.

[5] Adrienne Bloss, Paul Hudak, and Jonathan Young. Code optimization for lazy evaluation. *LISP and Symbolic Computation*, 1:147–164, 1988.

[6] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[7] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*, volume 1 of *Research Notes in Theoretical Computer Science.* Pitman, 1986.

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

[9] Olivier Danvy and John Hatcliff. Thunks (continued). In *Proceedings of the Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 3–11, Bordeaux, France, September 1992. IRISA, Rennes, France. Extended version available as Technical Report CIS-92-28, Kansas State University.

[10] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

[11] Francis Dupont. *Langage fonctionnels et parallélisme. Une réalisation pour le système CAML.* PhD thesis, École Polytechnique, Palaiseau, France, July 1990.

[12] Michael J. Fischer. Lambda-calculus schemata. In Talcott [34]. An earlier version appeared in the *Proceedings of the ACM Conference on Proving Assertions about Programs*, SIGPLAN Notices, Vol. 7, No. 1, January 1972.

[13] Timothy G. Griffin. A formulae-as-types notion of control. In Frances E. Allen, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

[14] Bob Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Talcott [34].

[15] John Hatcliff. *The Structure of Continuation-Passing Styles.* PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, USA, July 1994.

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

[17] P. Z. Ingerman. Thunks, a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961.

[18] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, June 1985.

[19] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[20] Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, 1990.

[21] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[22] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. In Talcott [34].

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

[24] Gordon D. Plotkin. Personal communication, New Orleans, Louisiana, March 1993.

[25] Eric Raymond (editor). *The New Hacker's Dictionary*. The MIT Press, 1992.

[26] John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *2nd Colloquium on Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages 141–156, Saarbrücken, West Germany, July 1974.

[27] John C. Reynolds. The discoveries of continuations. In Talcott [34], pages 233–247.

[28] Jon G. Riecke. Fully abstract translations between functional languages. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 245–254, Orlando, Florida, January 1991. ACM Press.

[29] Jon G. Riecke. *The Logic and Expressibility of Simply-typed Call-by-value and Lazy Languages*. PhD thesis, MIT, Cambridge, Massachusetts, August 1991.

[30] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. Technical Report 92-180, Department of Computer Science, Rice University, Houston, Texas, 1992.

[31] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Talcott [34], pages 289–360.

[32] Paul Steckler and Mitch Wand. Selective thunkification. In *Proceedings of the First International Static Analysis Symposium*, pages 162–178, Namur, Belgium, September 1994.

[33] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974.

[34] Carolyn L. Talcott, editor. *Special issue on continuations*, LISP and Symbolic Computation, Vol. 6, Nos. 3/4. Kluwer Academic Publishers, 1993.

Recent BRICS Report Series Publications

RS-97-7 **John Hatcliff and Olivier Danvy.** ***Thunks and the λ-Calculus (Extended Version)*****. March 1997. 55 pp. Extended version of article to appear in the *Journal of Functional Programming*.**

RS-97-6 **Olivier Danvy and Ulrik P. Schultz.** ***Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure*****. March 1997. 53 pp. Extended version of an article to appear in the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97), Amsterdam, The Netherlands, June 1997.**

RS-97-5 **Kousha Etessami, Moshe Y. Vardi, and Thomas Wilke.** ***First-Order Logic with Two Variables and Unary Temporal Logic*****. March 1997. 18 pp. To appear in *Twelfth Annual IEEE Symposium on Logic in Computer Science*, LICS '97 Proceedings.**

RS-97-4 **Richard Blute, Josée Desharnais, Abbas Edalat, and Prakash Panangaden.** ***Bisimulation for Labelled Markov Processes*****. March 1997. 48 pp. To appear in *Twelfth Annual IEEE Symposium on Logic in Computer Science*, LICS '97 Proceedings.**

RS-97-3 **Carsten Butz and Ieke Moerdijk.** ***A Definability Theorem for First Order Logic*****. March 1997. 10 pp.**

RS-97-2 **David A. Schmidt.** ***Abstract Interpretation in the Operational Semantics Hierarchy*****. March 1997. 33 pp.**

RS-97-1 **Olivier Danvy and Mayer Goldberg.** ***Partial Evaluation of the Euclidian Algorithm (Extended Version)*****. January 1997. 16 pp. To appear in the journal *Lisp and Symbolic Computation*.**

RS-96-62 **P. S. Thiagarajan and Igor Walukiewicz.** ***An Expressively Complete Linear Time Temporal Logic for Mazurkiewicz Traces*****. December 1996. i+13 pp. To appear in *Twelfth Annual IEEE Symposium on Logic in Computer Science*, LICS '97 Proceedings.**

RS-96-61 **Sergei Soloviev.** ***Proof of a Conjecture of S. Mac Lane*****. December 1996. 53 pp. Extended abstract appears in Pitt, Rydeheard and Johnstone, editors, *Category Theory and Computer Science: 6th International Conference*, CTCS '95 Proceedings, LNCS 953, 1995, pages 59–80.**