Thunks and the λ-calculus *

John Hatcliff
DIKU
Computer Science Department
Copenhagen University †
(hatcliff@diku.dk)

Olivier Danvy
DAIMI
Computer Science Department
Aarhus University ‡
(danvy@daimi.aau.dk)

February 21, 1995

Abstract

In his paper, *Call-by-name, call-by-value and the λ-calculus*, Plotkin 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 $\mathcal{C}_{\rm n}$ and $\mathcal{C}_{\rm v}$, by *deriving* $\mathcal{C}_{\rm n}$ as the composition of our thunk-based simulation $\mathcal{T}$ and of $\mathcal{C}_{\rm v}^+$ — an extension of $\mathcal{C}_{\rm v}$ handling thunks. Almost all of the correctness properties of $\mathcal{C}_{\rm n}$ follow from the properties of $\mathcal{T}$ and $\mathcal{C}_{\rm v}^+$. This simplifies reasoning about call-by-name continuation-passing style.

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

*Revised version: DIKU Rapport Nr. 95/3. This paper supersedes our earlier Technical Report CIS-93-15, Kansas State University.

†Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark.

This work is supported by the Danish Research Academy and by the DART project (Design, Analysis and Reasoning about Tools) of the Danish Research Councils.

‡Ny Munkegade, DK-8000 Aarhus C, Denmark.

This work is supported by the DART project (Design, Analysis and Reasoning about Tools) of the Danish Research Councils.

Contents

List of Figures

1 Introduction and Background

1.1 Motivation

In his seminal paper, *Call-by-name, call-by-value and the λ-calculus* [23], Plotkin 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 $\mathcal{C}_{\rm n}$.[3]

Given this, one may question what rôle continuations actually play in $\mathcal{C}_{\rm n}$ since they are unnecessary for achieving a simulation. We show that the continuation-passing structure of $\mathcal{C}_{\rm n}$ can actually be obtained by extending Plotkin's call-by-value continuation-based simulation $\mathcal{C}_{\rm v}$ to process the abstract representation of thunks and composing this extended transformation $\mathcal{C}_{\rm v}^{+}$ with $\mathcal{T}$, *i.e.*,[4]

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

This establishes a previously unrecognized connection between $\mathcal{C}_{\rm n}$ and $\mathcal{C}_{\rm v}$ 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 $\mathcal{C}_{\rm n}$ follow from the properties of $\mathcal{C}_{\rm v}^{+}$ and $\mathcal{T}$. So as a byproduct, when reasoning about $\mathcal{C}_{\rm n}$ and $\mathcal{C}_{\rm v}$, it is

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

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

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:

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

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:

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

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

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.

$$\begin{aligned} (\lambda x_1 . (\lambda x_2 . \mathit{force}\ x_1)\, (\mathit{delay}\ \Omega))\, (\mathit{delay}\ b) &\longmapsto_{\mathrm{v}} (\lambda x_2 . \mathit{force}\ (\mathit{delay}\ b))\, (\mathit{delay}\ \Omega) \\ &\longmapsto_{\mathrm{v}} \mathit{force}\ (\mathit{delay}\ b) \\ &\longmapsto_{\mathrm{v}} b \end{aligned} \tag{2}$$

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].

$$(\lambda k . (\lambda k . k\, (\lambda x_1 . \lambda k . (\lambda k . k\, (\lambda x_2 . \lambda k . x_1\, k))\, (\lambda y_1 . y_1\, \mathcal{C}_{\mathrm{n}}\langle\!\![\Omega]\!\!\rangle\, k)))\, (\lambda y_2 . y_2\, (\lambda k . k\, b)\, k))\, (\lambda y_3 . y_3) \tag{3}$$

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 $\lambda k \ldots$ and $\lambda y_i \ldots$ of line (3)) [23, p. 149],

$$(\lambda x_1 \,.\, \lambda k \,.\, (\lambda x_2 \,.\, \lambda k \,.\, x_1\, k)\, \mathcal{C}_{\mathrm{n}}\langle\!\![\![\Omega]\!]\!\!\rangle\, k)\,(\lambda k \,.\, k\, b)\,(\lambda y_3 \,.\, y_3) \tag{4}$$

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

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

[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. 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

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.

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

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

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 $\lambda A \vdash$

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:

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

Figure 2: Single-step evaluation rules

$e_1 = e_2$. 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.

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

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 $eval(e)$ 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

[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).

evaluation.

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

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]

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

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

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

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: $\mathcal{P}_{\mathrm{n}}\langle\!\!\{\cdot\}\!\!\rangle$ 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:**

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

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 $\mathcal{P}_\mathrm{n}$. However, the property does not hold because[11]

$$\lambda\beta \vdash e_1 = e_2 \;\not\Rightarrow\; \lambda\beta_i \vdash \mathcal{P}_\mathrm{n}\langle\!\![e_1]\!\!\rangle = \mathcal{P}_\mathrm{n}\langle\!\![e_2]\!\!\rangle.$$

In some cases, η_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

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

In practice, η_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 η_v reduction is unsound.

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

$$\mathcal{C}_\mathrm{n}\langle\!\![x]\!\!\rangle = \lambda k\,.\,x\,k$$

For the example above, the new translation gives $\lambda\beta_i \vdash \mathcal{C}_\mathrm{n}\langle\!\![\lambda x\,.\,(\lambda z\,.\,z)\,x]\!\!\rangle = \mathcal{C}_\mathrm{n}\langle\!\![\lambda x\,.\,x]\!\!\rangle$. The following theorem gives the correctness properties for $\mathcal{C}_\mathrm{n}$.

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

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

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

[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 $\lambda\beta_\mathrm{v} \vdash \mathcal{P}_\mathrm{n}\langle\!\![e_1]\!\!\rangle = \mathcal{P}_\mathrm{n}\langle\!\![e_2]\!\!\rangle$ ...".

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

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

The **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:** *If* $\lambda\beta_{\mathrm{v}} \vdash e_1 = e_2$ *then* $\lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{v}}\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\langle e_2\rangle\!\rangle$
   Also $\lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{v}}\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\langle e_2\rangle\!\rangle$ *iff* $\lambda\beta \vdash \mathcal{C}_{\mathrm{v}}\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\langle e_2\rangle\!\rangle$

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

[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).

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

Figure 5: Thunk introduction

$\mathcal{C}_{\mathrm{v}}$. 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 $\mathcal{C}_{\mathrm{n}}\langle\!\![e_1]\!\!\rangle \approx_{\mathrm{v}} \mathcal{C}_{\mathrm{n}}\langle\!\![e_2]\!\!\rangle$ (and similarly for $\mathcal{C}_{\mathrm{v}}$).[15]

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.

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

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.

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

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

---

[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 $\lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{v}}\langle\!\![e_1]\!\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\![e_2]\!\!\rangle$ and $\lambda\beta \vdash \mathcal{C}_{\mathrm{v}}\langle\!\![e_1]\!\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\![e_2]\!\!\rangle$ 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* $\lambda\beta\eta \vdash \mathcal{F}\langle\!\![e_1]\!\!\rangle = \mathcal{F}\langle\!\![e_2]\!\!\rangle$. $\mathcal{F}$ 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].

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 $\beta_{\mathrm{v}}\tau$ 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 $\twoheadrightarrow_{\tau}$ commutes with $\twoheadrightarrow_{\beta}$ and $\twoheadrightarrow_{\beta_{\mathrm{v}}}$.

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.

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

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 $\mathcal{C}_{\mathrm{n}}$. Specifically, we prove the following theorem which recasts the correctness theorem for $\mathcal{C}_{\mathrm{n}}$ (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:** $\mathit{eval}_{\mathrm{v}}(\mathcal{T}\langle\!\![e]\!\!\rangle) \simeq \mathit{eval}_{\mathrm{n}}(\mathcal{T}\langle\!\![e]\!\!\rangle)$
2. **Simulation:** $\mathcal{T}\langle\!\![\mathit{eval}_{\mathrm{n}}(e)]\!\!\rangle \simeq_{\tau} \mathit{eval}_{\mathrm{v}}(\mathcal{T}\langle\!\![e]\!\!\rangle)$
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* $\lambda\beta\tau \vdash \mathcal{T}\langle\!\![e_1]\!\!\rangle = \mathcal{T}\langle\!\![e_2]\!\!\rangle$

[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.

$$
\begin{array}{ccccccccc}
e_0 \overset{\mathcal{T}}{\sim} \mathcal{T}\langle\!\langle e_0 \rangle\!\rangle & & e_1 \overset{\mathcal{T}}{\sim} t_1^j & & e_2 \overset{\mathcal{T}}{\sim} t_2^j & & & & e_m \overset{\mathcal{T}}{\sim} t_m^j \\
\mathcal{T}\langle\!\langle e_0 \rangle\!\rangle & \longmapsto_{\beta_v} t_1^1 \longmapsto_\tau^* t_1^{i_1} & & \longmapsto_{\beta_v} t_2^1 \longmapsto_\tau^* t_2^{i_2} & & \longmapsto_{\beta_v} & \cdots\cdots & & \longmapsto_{\beta_v} t_m^1 \longmapsto_\tau^* t_m^{i_m} \\
e_0 & \longmapsto_{\mathrm{n}} & e_1 & \longmapsto_{\mathrm{n}} & e_2 & \longmapsto_{\mathrm{n}} & \cdots\cdots & \longmapsto_{\mathrm{n}} & e_m
\end{array}
$$

Figure 6: Evaluation of thunked terms

2.3.1 Indifference

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

2.3.2 Simulation

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

$$
\begin{array}{llll}
\overset{\mathcal{T}}{\sim}.1 & b \overset{\mathcal{T}}{\sim} b & \overset{\mathcal{T}}{\sim}.4 & \dfrac{e_0 \overset{\mathcal{T}}{\sim} t_0 \qquad e_1 \overset{\mathcal{T}}{\sim} t_1}{e_0\, e_1 \overset{\mathcal{T}}{\sim} t_0\,(delay\; t_1)} \\[2ex]
\overset{\mathcal{T}}{\sim}.2 & x \overset{\mathcal{T}}{\sim} force\; x & \overset{\mathcal{T}}{\sim}.5 & \dfrac{e \overset{\mathcal{T}}{\sim} t}{e \overset{\mathcal{T}}{\sim} force\,(delay\; t)} \\[2ex]
\overset{\mathcal{T}}{\sim}.3 & \dfrac{e \overset{\mathcal{T}}{\sim} t}{\lambda x \,.\, e \overset{\mathcal{T}}{\sim} \lambda x \,.\, t} & &
\end{array}
$$

Simple inductions show that $e \overset{\mathcal{T}}{\sim} \mathcal{T}\langle\!\langle e \rangle\!\rangle$, and that $e \overset{\mathcal{T}}{\sim} t$ implies $\mathcal{T}\langle\!\langle e \rangle\!\rangle$ 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 $t_0 \in Programs[\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*]$ such that $e_0 \overset{\mathcal{T}}{\sim} t_0$,*

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

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).

[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 $\mathcal{T}\langle\!\langle eval_{\mathrm{n}}(e) \rangle\!\rangle$ may be in τ-normal form where $eval_{\mathrm{v}}(\mathcal{T}\langle\!\langle e \rangle\!\rangle)$ may contain τ-redexes inside the body of a resulting abstraction.

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

Figure 7: Thunk elimination

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 $\mathcal{T}\langle\!\lbrack \Lambda \rbrack\!\rangle^*$ under theory $\lambda\beta_i\tau$ (*i.e.*, $\lambda\beta_{\rm v}\tau$ 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 $\mathcal{T}\langle\!\lbrack \Lambda \rbrack\!\rangle^*$. For the reverse direction, the thunk elimination $\mathcal{T}^{-1}$ of Figure 7 establishes a mapping from $\mathcal{T}\langle\!\lbrack \Lambda \rbrack\!\rangle^*$ 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* $t \in \mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$, $\lambda\tau \vdash t = (\mathcal{T} \circ \mathcal{T}^{-1})\langle\!\langle t\rangle\!\rangle$.

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

$$(\mathcal{T} \circ \mathcal{T}^{-1})\langle\!\langle(\lambda x\,.\,\mathit{force}\,(\mathit{delay}\;b))\,(\mathit{delay}\;b)\rangle\!\rangle \;=\; \mathcal{T}\langle\!\langle(\lambda x\,.\,b)\,b\rangle\!\rangle \;=\; (\lambda x\,.\,b)\,(\mathit{delay}\;b).$$

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

Property 4 *For all* $e_1, e_2 \in \Lambda$, $\lambda\beta \vdash e_1 \longrightarrow e_2 \;\Rightarrow\; \lambda\beta_i\tau \vdash \mathcal{T}\langle\!\langle e_1\rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}\langle\!\langle e_2\rangle\!\rangle$.

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

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

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

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

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

Property 5 *For all* $t_1, t_2 \in \mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$, $\lambda\beta_i\tau \vdash t_1 \longrightarrow t_2 \;\Rightarrow\; \lambda\beta \vdash \mathcal{T}^{-1}\langle\!\langle t_1\rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}^{-1}\langle\!\langle t_2\rangle\!\rangle$

Specifically, a τ-reduction in $\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$ 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 $\mathcal{T}^{-1}\langle\!\langle t_1\rangle\!\rangle = b = \mathcal{T}^{-1}\langle\!\langle t_2\rangle\!\rangle$, so no reductions occur.

A β_i-reduction in $\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$ 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

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

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).

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

Figure 8: Thunk introduction implemented in Λ

2.4 Thunks implemented in Λ

Representing thunks *via* abstract suspension operators *delay* and *force* simplifies the technical presentation and enables the connection between $\mathcal{C}_{\mathrm{n}}$ and $\mathcal{C}_{\mathrm{v}}$ 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).

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

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

1. $\mathcal{L}\langle\!\![\mathit{delay}\ e]\!\!\rangle \ = \ \lambda z\,.\,\mathcal{L}\langle\!\![e]\!\!\rangle$ is a value; and

2. τ-reduction is faithfully implemented in both the call-by-name and call-by-value calculi, *i.e.*,
$$\mathcal{L}\langle\!\![\mathit{force}\,(\mathit{delay}\ e)]\!\!\rangle \ = \ (\lambda z\,.\,\mathcal{L}\langle\!\![e]\!\!\rangle)\,b \ \longrightarrow_{\beta_i} \ \mathcal{L}\langle\!\![e]\!\!\rangle.$$

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 $\mathcal{C}_{\mathrm{n}}$ (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:** $\mathit{eval}_{\mathrm{v}}(\mathcal{T}_{\mathcal{L}}\langle\!\![e]\!\!\rangle) \simeq \mathit{eval}_{\mathrm{n}}(\mathcal{T}_{\mathcal{L}}\langle\!\![e]\!\!\rangle)$

2. **Simulation:** $\mathcal{T}_{\mathcal{L}}\langle\!\langle eval_{\mathrm{n}}(e)\rangle\!\rangle \simeq_{\beta_i} eval_{\mathrm{v}}(\mathcal{T}_{\mathcal{L}}\langle\!\langle e\rangle\!\rangle)$

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

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 $t \in Programs[\mathcal{T}\langle\!\langle \Lambda\rangle\!\rangle^*]$, $\mathcal{L}\langle\!\langle eval_{\mathrm{v}}(t)\rangle\!\rangle \simeq eval_i(\mathcal{L}\langle\!\langle t\rangle\!\rangle)$. 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* $e \in Values_{\mathrm{v}}[\Lambda_\tau]$, $\mathcal{C}_{\mathrm{v}}^+\langle\!\langle delay\ e\rangle\!\rangle = \lambda k\,.\,k\,(\mathcal{C}_{\mathrm{v}}^+\langle delay\ e\rangle)$. 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.*, $\mathcal{C}_{\mathrm{v}}^+\langle delay\ e\rangle = \mathcal{C}_{\mathrm{v}}^+\langle\!\langle e\rangle\!\rangle$. 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 $\mathcal{C}_{\mathrm{v}}^+\langle delay\ e\rangle = \mathcal{C}_{\mathrm{v}}^+\langle\!\langle e\rangle\!\rangle$.

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

[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].

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

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

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:

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

The last step follows by a straightforward induction over the structure of e since $\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle e \rangle\!\rangle$ 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:** $\mathcal{C}_{\mathrm{v}}\langle \mathit{eval}_{\mathrm{v}}(e) \rangle \simeq \mathit{eval}_{\mathrm{n}}(\mathcal{C}_{\mathrm{v}}\langle\!\langle e \rangle\!\rangle\, I)$
3. **Translation:** *If* $\lambda\beta_{\mathrm{v}}\tau \vdash e_1 = e_2$ *then* $\lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{v}}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\langle e_2 \rangle\!\rangle$
   Also $\lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{v}}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\langle e_2 \rangle\!\rangle$ *iff* $\lambda\beta \vdash \mathcal{C}_{\mathrm{v}}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{C}_{\mathrm{v}}\langle\!\langle e_2 \rangle\!\rangle$

[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 $e \;\equiv\; \lambda x \,.\, e'$:

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

case $e \;\equiv\; e_0\, e_1$:

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

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

Figure 10: Optimizing call-by-name CPS transformation

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

Figure 11: Optimizing call-by-value CPS transformation

∎

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 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.

[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.

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

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

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

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

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 will be computed at translation time.

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

$$\begin{array}{lll}
& \mathit{eval}_{\mathrm{v}}(\mathcal{C}_{\mathrm{n}}\langle\!\langle e \rangle\!\rangle\,(\lambda y \,.\, y)) \;=\; b & \\
\Leftrightarrow & \mathit{eval}_{\mathrm{v}}((\mathcal{C}^+_{\mathrm{v}} \circ \mathcal{T})\langle\!\langle e \rangle\!\rangle\,(\lambda y \,.\, y)) \;=\; b & \ldots \textit{Theorem 9 and Theorem 1 (soundness of } \beta_{\mathrm{v}}\textit{)} \\
\Leftrightarrow & \mathit{eval}_{\mathrm{n}}((\mathcal{C}^+_{\mathrm{v}} \circ \mathcal{T})\langle\!\langle e \rangle\!\rangle\,(\lambda y \,.\, y)) \;=\; b & \ldots \textit{Theorem 8 } \textbf{(Indifference)} \\
\Leftrightarrow & \mathit{eval}_{\mathrm{n}}(\mathcal{C}_{\mathrm{n}}\langle\!\langle e \rangle\!\rangle\,(\lambda y \,.\, y)) \;=\; b & \ldots \textit{Theorem 9 and Theorem 1 (soundness of } \beta\textit{)}
\end{array}$$

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

$$\begin{array}{lll}
& \mathit{eval}_{\mathrm{n}}(e) \;=\; b & \\
\Leftrightarrow & \mathit{eval}_{\mathrm{v}}(\mathcal{T}\langle\!\langle e \rangle\!\rangle) \;=\; b & \ldots \textit{Theorem 5 } \textbf{(Simulation)} \\
\Leftrightarrow & \mathit{eval}_{\mathrm{n}}((\mathcal{C}^+_{\mathrm{v}} \circ \mathcal{T})\langle\!\langle e \rangle\!\rangle\,(\lambda y \,.\, y)) \;=\; b & \ldots \textit{Theorem 8 } \textbf{(Simulation)} \\
\Leftrightarrow & \mathit{eval}_{\mathrm{v}}((\mathcal{C}^+_{\mathrm{v}} \circ \mathcal{T})\langle\!\langle e \rangle\!\rangle\,(\lambda y \,.\, y)) \;=\; b & \ldots \textit{Theorem 8 } \textbf{(Indifference)} \\
\Leftrightarrow & \mathit{eval}_{\mathrm{v}}(\mathcal{C}_{\mathrm{n}}\langle\!\langle e \rangle\!\rangle\,(\lambda y \,.\, y)) \;=\; b & \ldots \textit{Theorem 9 and Theorem 1 (soundness of } \beta_{\mathrm{v}}\textit{)}
\end{array}$$

[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.

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

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

This can be summarized as follows.

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

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 characterizing call-by-need is captured by

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

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.*, $\lambda\beta_i \not\vdash \mathcal{C}_{\mathrm{n}}\langle\!\![e]\!\!\rangle = (\mathcal{C}_{\mathrm{v}} \circ \mathcal{T}_{\mathcal{L}})\langle\!\![e]\!\!\rangle$).

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

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

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 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*.

$$\Gamma \vdash b : \iota \qquad \Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma, x\!:\!\sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \lambda x \,.\, e : \sigma_1 \rightarrow \sigma_2} \qquad \frac{\Gamma \vdash e_0 : \sigma_1 \rightarrow \sigma_2 \qquad \Gamma \vdash e_1 : \sigma_1}{\Gamma \vdash e_0\, e_1 : \sigma_2}$$

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

Figure 12: Typing rules for Λ

$$\Gamma \vdash_\tau b : \iota \qquad \Gamma \vdash_\tau x : \Gamma(x) \qquad \frac{\Gamma, x\!:\!\sigma_1 \vdash_\tau e : \sigma_2}{\Gamma \vdash_\tau \lambda x \,.\, e : \sigma_1 \rightarrow \sigma_2} \qquad \frac{\Gamma \vdash_\tau e_0 : \sigma_1 \rightarrow \sigma_2 \qquad \Gamma \vdash_\tau e_1 : \sigma_1}{\Gamma \vdash_\tau e_0\, e_1 : \sigma_2}$$

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

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

Figure 13: Typing rules for Λ_τ

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

$$\begin{aligned} \mathcal{T}\langle\!\langle\cdot\rangle\!\rangle &: \mathit{Types}[\Lambda] \rightarrow \mathit{Types}[\Lambda_\tau] \\ \mathcal{T}\langle\!\langle\iota\rangle\!\rangle &= \iota \\ \mathcal{T}\langle\!\langle\sigma_1 \rightarrow \sigma_2\rangle\!\rangle &= \widetilde{\mathcal{T}\langle\!\langle\sigma_1\rangle\!\rangle} \rightarrow \mathcal{T}\langle\!\langle\sigma_2\rangle\!\rangle \end{aligned}$$

$$\begin{aligned} \mathcal{T}\langle\!\langle\cdot\rangle\!\rangle &: \mathit{Assums}[\Lambda] \rightarrow \mathit{Assums}[\Lambda_\tau] \\ \mathcal{T}\langle\!\langle\Gamma, x:\sigma\rangle\!\rangle &= \mathcal{T}\langle\!\langle\Gamma\rangle\!\rangle, x:\widetilde{\mathcal{T}\langle\!\langle\sigma\rangle\!\rangle} \end{aligned}$$

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

$$\begin{aligned} \mathcal{C}_\mathrm{n}\langle\!\langle\cdot\rangle\!\rangle &: \mathit{Types}[\Lambda] \rightarrow \mathit{Types}[\Lambda] \\ \mathcal{C}_\mathrm{n}\langle\!\langle\sigma\rangle\!\rangle &= (\mathcal{C}_\mathrm{n}\langle\sigma\rangle \rightarrow \mathit{Ans}) \rightarrow \mathit{Ans} \\ \mathcal{C}_\mathrm{n}\langle\iota\rangle &= \iota \\ \mathcal{C}_\mathrm{n}\langle\sigma_1 \rightarrow \sigma_2\rangle &= \mathcal{C}_\mathrm{n}\langle\!\langle\sigma_1\rangle\!\rangle \rightarrow \mathcal{C}_\mathrm{n}\langle\!\langle\sigma_2\rangle\!\rangle \end{aligned}$$

$$\begin{aligned} \mathcal{C}_\mathrm{n}\langle\!\langle\cdot\rangle\!\rangle &: \mathit{Assums}[\Lambda] \rightarrow \mathit{Assums}[\Lambda] \\ \mathcal{C}_\mathrm{n}\langle\!\langle\Gamma, x:\sigma\rangle\!\rangle &= \mathcal{C}_\mathrm{n}\langle\!\langle\Gamma\rangle\!\rangle, x:\mathcal{C}_\mathrm{n}\langle\!\langle\sigma\rangle\!\rangle \end{aligned}$$

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

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* $\mathcal{T}\langle\!\langle\Gamma\rangle\!\rangle \vdash_\tau \mathcal{T}\langle\!\langle e\rangle\!\rangle : \mathcal{T}\langle\!\langle\sigma\rangle\!\rangle$.

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

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

and Λ_τ. Ambiguity is avoided by subscripting the typing judgement symbol $\vdash_\tau$ for the language Λ_τ.

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

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

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

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

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

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. $\mathcal{C}_{\mathrm{v}}^{+}\llbracket\mathcal{T}\llbracket\sigma\rrbracket\rrbracket = \mathcal{C}_{\mathrm{n}}\llbracket\sigma\rrbracket$ *types*
2. $\mathcal{C}_{\mathrm{v}}^{+}\langle\mathcal{T}\llbracket\sigma\rrbracket\rangle = \mathcal{C}_{\mathrm{n}}\langle\sigma\rangle$ *value types*
3. $\mathcal{C}_{\mathrm{v}}^{+}\llbracket\mathcal{T}\llbracket\Gamma\rrbracket\rrbracket = \mathcal{C}_{\mathrm{n}}\llbracket\Gamma\rrbracket$ *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.

$$
\begin{aligned}
\mathcal{C}_{\mathrm{v}}\langle\mathcal{T}\llbracket\sigma_1 \rightarrow \sigma_2\rrbracket\rangle &= \mathcal{C}_{\mathrm{v}}^{+}\langle\widetilde{\mathcal{T}\llbracket\sigma_1\rrbracket} \rightarrow \mathcal{T}\llbracket\sigma_2\rrbracket\rangle \\
&= \mathcal{C}_{\mathrm{v}}^{+}\langle\widetilde{\mathcal{T}\llbracket\sigma_1\rrbracket}\rangle \rightarrow \mathcal{C}_{\mathrm{v}}^{+}\llbracket\mathcal{T}\llbracket\sigma_2\rrbracket\rrbracket \\
&= \mathcal{C}_{\mathrm{v}}^{+}\llbracket\mathcal{T}\llbracket\sigma_1\rrbracket\rrbracket \rightarrow \mathcal{C}_{\mathrm{v}}^{+}\llbracket\mathcal{T}\llbracket\sigma_2\rrbracket\rrbracket \\
&= \mathcal{C}_{\mathrm{n}}\llbracket\sigma_1\rrbracket \rightarrow \mathcal{C}_{\mathrm{n}}\llbracket\sigma_2\rrbracket \qquad \textit{...by ind. hyp.} \\
&= \mathcal{C}_{\mathrm{n}}\langle\sigma_1 \rightarrow \sigma_2\rangle
\end{aligned}
$$

∎

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 $\mathcal{C}_{\mathrm{n}}\llbracket\sigma\rrbracket = (\mathcal{C}_{\mathrm{n}}\langle\sigma\rangle \rightarrow \mathit{Ans}) \rightarrow \mathit{Ans}$ and $\mathcal{C}_{\mathrm{v}}\llbracket\sigma\rrbracket = (\mathcal{C}_{\mathrm{v}}\langle\sigma\rangle \rightarrow \mathit{Ans}) \rightarrow \mathit{Ans}$. $\mathcal{C}_{\mathrm{n}}$ and $\mathcal{C}_{\mathrm{v}}$ differ in how arguments are treated. This is reflected by the difference in the definitions $\mathcal{C}_{\mathrm{n}}\langle\sigma_1 \rightarrow \sigma_2\rangle = \mathcal{C}_{\mathrm{n}}\llbracket\sigma_1\rrbracket \rightarrow \mathcal{C}_{\mathrm{n}}\llbracket\sigma_2\rrbracket$ and $\mathcal{C}_{\mathrm{v}}\langle\sigma_1 \rightarrow \sigma_2\rangle = \mathcal{C}_{\mathrm{v}}\langle\sigma_1\rangle \rightarrow \mathcal{C}_{\mathrm{v}}\llbracket\sigma_2\rrbracket$. 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.*, $\mathcal{T}\langle\!\langle\sigma_1 \rightarrow \sigma_2\rangle\!\rangle = \widetilde{\mathcal{T}\langle\!\langle\sigma_1\rangle\!\rangle} \rightarrow \mathcal{T}\langle\!\langle\sigma_2\rangle\!\rangle$. 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 co-monad.

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 transformation $\mathcal{T}$. 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

[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).

$\mathcal{C}_{\mathrm{v}}^{+}$ 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]

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

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

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