BRICS RS-96-19 Hatcliff & Danvy: Thunks and the λ-Calculus

BRICS

Basic Research in Computer Science

Thunks and the λ-Calculus

John Hatcliff
Olivier Danvy

BRICS Report Series **RS-96-19**

ISSN 0909-0878 **June 1996**

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

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

Thunks and the λ-calculus *

John Hatcliff †
DIKU
Copenhagen University ‡
(hatcliff@diku.dk)

Olivier Danvy
BRICS §
Aarhus University ¶
(danvy@brics.dk)

May 1996

Abstract

Thirty-five years ago, thunks were used to simulate call-by-name under call-by-value in Algol 60. Twenty years ago, Plotkin presented continuation-based simulations of call-by-name under call-by-value and vice versa in the λ-calculus. We connect all three of these classical simulations by factorizing the continuation-based call-by-name simulation $\mathcal{C}_{\mathrm{n}}$ with a thunk-based call-by-name simulation $\mathcal{T}$ followed by the continuation-based call-by-value simulation $\mathcal{C}_{\mathrm{v}}$ extended to thunks.

$$\begin{tikzcd} \Lambda \arrow[r, "\mathcal{T}"] \arrow[dr, "\mathcal{C}_{\mathrm{n}}"'] & \Lambda_{\mathrm{thunks}} \arrow[d, "\mathcal{C}_{\mathrm{v}}"] \\ & \Lambda_{\mathrm{CPS}} \end{tikzcd}$$

We show that $\mathcal{T}$ actually satisfies all of Plotkin's correctness criteria for $\mathcal{C}_{\mathrm{n}}$ (*i.e.*, his **Indifference**, **Simulation**, and **Translation** theorems). Furthermore, most of the correctness theorems for $\mathcal{C}_{\mathrm{n}}$ can now be seen as simple corollaries of the corresponding theorems for $\mathcal{C}_{\mathrm{v}}$ and $\mathcal{T}$.

---

*To appear in the Journal of Functional Programming.

†Supported by the Danish Research Academy and by the DART project (Design, Analysis and Reasoning about Tools) of the Danish Research Councils.

‡Computer Science Department, Universitetsparken 1, 2100 Copenhagen Ø, Denmark.

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

¶Computer Science Department, Ny Munkegade, B. 540, 8000 Aarhus C, Denmark.

1 Introduction

In his seminal paper, "Call-by-name, call-by-value and the λ-calculus" [27], Plotkin presents simulations of call-by-name by call-by-value (and vice-versa). Both of Plotkin's simulations rely on *continuations*. Since Algol 60, however, programming wisdom has it that *thunks* can be used to obtain a simpler simulation of call-by-name by call-by-value. We show that composing a thunk-based call-by-name simulation $\mathcal{T}$ with Plotkin's continuation-based call-by-value simulation $\mathcal{C}_{\mathrm{v}}$ actually yields Plotkin's continuation-based call-by-name simulation $\mathcal{C}_{\mathrm{n}}$ (Sections 2 and 3). Revisiting Plotkin's correctness theorems (Section 4), we provide a correction to his **Translation** property for $\mathcal{C}_{\mathrm{n}}$, and show that the thunk-based simulation $\mathcal{T}$ satisfies all of Plotkin's properties for $\mathcal{C}_{\mathrm{n}}$. The factorization of $\mathcal{C}_{\mathrm{n}}$ by $\mathcal{C}_{\mathrm{v}}$ and $\mathcal{T}$ makes it possible to derive correctness properties for $\mathcal{C}_{\mathrm{n}}$ from the corresponding results for $\mathcal{C}_{\mathrm{v}}$ and $\mathcal{T}$. This factorization has also found several other applications already (Section 5). The extended version of this paper [15] gives a more detailed development as well as all proofs.

2 Continuation-based and Thunk-based Simulations

We consider Λ, the untyped λ-calculus parameterized by a set of basic constants b [27, p. 127].

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

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.

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

Figure 1 displays Plotkin's call-by-name CPS transformation $\mathcal{C}_{\mathrm{n}}$ (which simulates call-by-name under call-by-value). (Side note: the term "CPS" stands for "Continuation-Passing Style". It was coined in Steele's MS thesis [35].) Figure 2 displays Plotkin's call-by-value CPS transformation $\mathcal{C}_{\mathrm{v}}$ (which simulates call-by-value under call-by-name). Figure 3 displays the standard thunk-based simulation of call-by-name using call-by-value evaluation of the language Λ_{τ}. Λ_{τ} extends Λ as follows.

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

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

Figure 1: Call-by-name CPS transformation

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

Figure 2: Call-by-value CPS transformation

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

Figure 3: Call-by-name thunk transformation

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

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

The operator *force* triggers the evaluation of such a suspended expression. This is formalized by the following notion of reduction.

Definition 1 (τ-reduction)

$$\mathit{force}\,(\mathit{delay}\ e) \longrightarrow_\tau e$$

We also consider the conventional notions of reduction β, β_{v}, η, and η_{v} [3, 27, 32].

Definition 2 (β,β_{v}, η, η_{v}-reduction)

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

For a notion of reduction r, $\longrightarrow_r$ also denotes construct compatible one-step reduction, $\longrightarrow\!\!\!\!\!\rightarrow_r$ denotes the reflexive, transitive closure of $\longrightarrow_r$, and $=_r$ denotes the smallest equivalence relation generated by $\longrightarrow_r$ [3]. We will also write $\lambda r \vdash e_1 = e_2$ when $e_1 =_r e_2$ (similarly for the other relations).

Figure 4 extends $\mathcal{C}_{\mathrm{v}}$ (see Figure 2) to obtain $\mathcal{C}_{\mathrm{v}}^+$ which CPS-transforms thunks. $\mathcal{C}_{\mathrm{v}}^+$ faithfully implements τ-reduction in terms of β_{v} (and thus β) reduction. We write β_i below to signify that the property holds indifferently for β_{v} and β.

Property 1 *For all $e \in \Lambda_\tau$, $\lambda\beta_i \vdash \mathcal{C}_{\mathrm{v}}^+\langle\!\![\mathit{force}\,(\mathit{delay}\ e)]\!\!\rangle = \mathcal{C}_{\mathrm{v}}^+\langle\!\![e]\!\!\rangle$.*

Proof:

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

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

3 Connecting the Continuation-based and Thunk-based Simulations

$\mathcal{C}_{\mathrm{n}}$ can be factored into two conceptually distinct steps: (1) the suspension of argument evaluation (captured in $\mathcal{T}$); and (2) the sequentialization of function application to give the usual tail-calls of CPS terms (captured in $\mathcal{C}_{\mathrm{v}}^{+}$).

Theorem 1 *For all $e \in \Lambda$, $\lambda\beta_i \vdash (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\![e]\!\!\rangle = \mathcal{C}_{\mathrm{n}}\langle\!\![e]\!\!\rangle$.*

Proof: by induction over the structure of e.

case $e \equiv b$:

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

case $e \equiv x$:

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

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

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

case $e \equiv e_0\, e_1$:

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

∎

This theorem implies that the diagram in the abstract commutes up to β_i-equivalence, *i.e.*, indifferently up to $\beta_{\rm v}$- and β-equivalence. Note that $\mathcal{C}_{\rm v}^+ \circ \mathcal{T}$ and $\mathcal{C}_{\rm n}$ only differ by administrative reductions. In fact, if we consider optimizing versions of $\mathcal{C}_{\rm n}$ and $\mathcal{C}_{\rm v}$ that remove administrative redexes, then the diagram commutes up to identity (*i.e.*, up to α-equivalence).

Figures 5 and 6 present two such optimizing transformations $\mathcal{C}_{{\rm n}.opt}$ and $\mathcal{C}_{{\rm v}.opt}$. The output of $\mathcal{C}_{{\rm n}.opt}$ is $\beta_{\rm v}\eta_{\rm v}$ equivalent to the output of $\mathcal{C}_{\rm n}$, and similarly for $\mathcal{C}_{{\rm v}.opt}$ and $\mathcal{C}_{\rm v}$, as shown by Danvy and Filinski [7, pp. 387 and 367]. Both are applied to the identity continuation. In Figures 5 and 6, they are presented in a two-level language *à la* Nielson and Nielson [22]. 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. It is simple to transcribe $\mathcal{C}_{{\rm n}.opt}$ and $\mathcal{C}_{{\rm v}.opt}$ into functional programs.

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

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

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

Theorem 2 *For all* $e \in \Lambda$, $(\mathcal{C}^+_{{\rm v}.opt} \circ \mathcal{T})\langle\!\![e]\!\!\rangle \overline{@} (\overline{\lambda} a.a) \equiv \mathcal{C}_{{\rm n}.opt}\langle\!\![e]\!\!\rangle \overline{@} (\overline{\lambda} a.a)$.

Proof: A simple structural induction similar to the one required in the proof of Theorem 1. We show only the case for identifiers (the others are similar). The overlined constructs are computed at translation time, and thus simplifying overlined constructs using β-conversion yields equivalent specifications.

case $e \equiv x$:

$$
\begin{array}{rcl}
(\mathcal{C}^+_{{\rm v}.opt} \circ \mathcal{T})\langle\!\![x]\!\!\rangle & = & \overline{\lambda} k.(\overline{\lambda} k.k \overline{@} x) \overline{@} (\overline{\lambda} y_0.y_0 \underline{@} (\underline{\lambda} y_1.k \overline{@} y_1)) \\
& = & \overline{\lambda} k.(\overline{\lambda} y_0.y_0 \underline{@} (\underline{\lambda} y_1.k \overline{@} y_1)) \overline{@} x \\
& = & \overline{\lambda} k.x \underline{@} (\underline{\lambda} y_1.k \overline{@} y_1) \\
& = & \mathcal{C}_{{\rm n}.opt}\langle\!\![x]\!\!\rangle
\end{array}
$$

∎

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

Figure 4: Call-by-value CPS transformation (extended to thunks)

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

Figure 5: Optimizing call-by-name CPS transformation

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

Figure 6: Optimizing call-by-value CPS transformation

Call-by-name:

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

$$\frac{e_0 \longmapsto_{\mathrm{n}} e_0'}{e_0\, e_1 \longmapsto_{\mathrm{n}} e_0'\, e_1}$$

Call-by-value:

$$(\lambda x.e)\, v \longmapsto_{\mathrm{v}} e[x := v]$$

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

Figure 7: Single-step evaluation rules

4 Revisiting Plotkin's Correctness Properties

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

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

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 7.

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

For a language l, $\mathit{Programs}[l]$ denotes the closed terms in l. For meta-language expressions E_1, E_2, 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. We will also write $E_1 \simeq_r E_2$ when E_1 and E_2 are both undefined, or else are both defined and denote r-convertible terms for the convertibility relation generated by some notion of reduction r.

Plotkin expressed the correctness of his simulations $\mathcal{C}_{\rm n}$ and $\mathcal{C}_{\rm v}$ *via* three properties: **Indifference**, **Simulation**, and **Translation**. **Indifference** states that call-by-name and call-by-value evaluation coincide on terms in the image of the CPS transformation. **Simulation** states that the desired evaluation strategy is properly simulated. **Translation** states how the transformation relates program calculi for each evaluation strategy (*e.g.*, $\lambda\beta$, $\lambda\beta_{\rm v}$). Let us restate these properties for Plotkin's original presentation of $\mathcal{C}_{\rm n}$ (hereby noted $\mathcal{P}_{\rm n}$) [27, p. 153], that only differs from Figure 1 at the line for identifiers.

$$\mathcal{P}_{\rm n}\langle\!\langle x\rangle\!\rangle \quad = \quad x$$

Theorem 3 (Plotkin 1975) *For all $e \in Programs[\Lambda]$,*

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

where I denotes the identity function and is used as the initial continuation.

Plotkin also claimed the following **Translation** property.

Claim 1 (Plotkin 1975) *For all $e_1, e_2 \in \Lambda$,*

$$\begin{array}{rlll}
\textbf{Translation:} & \lambda\beta \vdash e_1 = e_2 & \textit{iff} & \lambda\beta_{\rm v} \vdash \mathcal{P}_{\rm n}\langle\!\langle e_1\rangle\!\rangle = \mathcal{P}_{\rm n}\langle\!\langle e_2\rangle\!\rangle \\
& & \textit{iff} & \lambda\beta \vdash \mathcal{P}_{\rm n}\langle\!\langle e_1\rangle\!\rangle = \mathcal{P}_{\rm n}\langle\!\langle e_2\rangle\!\rangle \\
& & \textit{iff} & \lambda\beta_{\rm v} \vdash \mathcal{P}_{\rm n}\langle\!\langle e_1\rangle\!\rangle\, I = \mathcal{P}_{\rm n}\langle\!\langle e_2\rangle\!\rangle\, I \\
& & \textit{iff} & \lambda\beta \vdash \mathcal{P}_{\rm n}\langle\!\langle e_1\rangle\!\rangle\, I = \mathcal{P}_{\rm n}\langle\!\langle e_2\rangle\!\rangle\, I
\end{array}$$

The **Translation** property purports to show that β-equivalence classes are preserved and reflected by $\mathcal{P}_{\rm n}$. The property, however, does not hold because

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

The proof breaks down at the statement "It is straightforward to show that $\lambda\beta \vdash e_1 = e_2$ implies $\lambda\beta_{\rm v} \vdash \mathcal{P}_{\rm n}\langle\!\langle e_1\rangle\!\rangle = \mathcal{P}_{\rm n}\langle\!\langle e_2\rangle\!\rangle$ ..." [27, p. 158]. In some cases, $\eta_{\rm v}$ is needed to establish the equivalence of the CPS-images of two β-convertible terms. For example, $\lambda x.(\lambda z.z)\, x \longrightarrow_\beta \lambda x.x$ but

$$\begin{array}{rcll}
\mathcal{P}_{\rm n}\langle\!\langle \lambda x.(\lambda z.z)\, x\rangle\!\rangle & = & \lambda k.k\,(\lambda x.\lambda k.(\lambda k.k\,(\lambda z.z))\,(\lambda y.y\,x\,k)) & (1) \\
& \longrightarrow_{\beta_{\rm v}} & \lambda k.k\,(\lambda x.\lambda k.(\lambda y.y\,x\,k)\,(\lambda z.z)) & (2)
\end{array}$$

$$\begin{aligned}
&\longrightarrow_{\beta_v} && \lambda k.k\,(\lambda x.\lambda k.(\lambda z.z)\,x\,k) && (3)\\
&\longrightarrow_{\beta_v} && \lambda k.k\,(\lambda x.\lambda k.x\,k) && (4)\\
&\longrightarrow_{\eta_v} && \lambda k.k\,(\lambda x.x) \qquad \ldots\eta_v \textit{ is needed for this step} && (5)\\
&= && \mathcal{P}_n\langle\!\langle \lambda x.x\rangle\!\rangle. && (6)
\end{aligned}$$

Since the two distinct terms at lines (4) and (5) are β_i-normal, confluence of β_i implies $\lambda\beta_i \not\vdash \mathcal{P}_n\langle\!\langle e_1\rangle\!\rangle = \mathcal{P}_n\langle\!\langle e_2\rangle\!\rangle$.

In practice, though, η_v reductions such as those required in the example above are unproblematic if they are embedded in proper CPS contexts (*e.g.*, contexts in the language of terms in the image of $\mathcal{P}_n$ closed under β_i reductions). 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. In this case, the η_v reduction can be expressed by a β_v reduction. If the term, however, is not embedded in a CPS context (*e.g.*, $[\cdot]\,(\lambda y.y\,b)$), the η_v reduction is unsound, *i.e.*, it fails to preserve *operational equivalence* as defined by Plotkin [27, pp. 144,147]. Such reductions are unsound due to "improper" uses of basic constants. For example, $\lambda x.b\,x \longrightarrow_{\eta_v} b$ but $\lambda x.b\,x \not\approx_v b$ (take $C = [\cdot]$) where $\approx_v$ is the call-by-value operational equivalence relation defined by Plotkin [15, p. 9]. Note, finally, that a simple typing discipline eliminates improper uses of basic constants, and consequently give soundness for η_v.

The simplest solution for recovering the **Translation** property is to change the translation of identifiers from $\mathcal{P}_n\langle\!\langle x\rangle\!\rangle = x$ to $\lambda k.x\,k$ — obtaining the translation $\mathcal{C}_n$ given in Figure 1.[1]

For the example above, the modified translation gives

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

The following theorem gives the correctness properties for $\mathcal{C}_n$.

Theorem 4 *For all $e \in Programs[\Lambda]$ and $e_1, e_2 \in \Lambda$,*

1. **Indifference:** $eval_v(\mathcal{C}_n\langle\!\langle e\rangle\!\rangle\, I) \simeq eval_n(\mathcal{C}_n\langle\!\langle e\rangle\!\rangle\, I)$
2. **Simulation:** $\mathcal{C}_n\langle eval_n(e)\rangle \simeq_{\beta_i} eval_v(\mathcal{C}_n\langle\!\langle e\rangle\!\rangle\, I)$
3. **Translation:**

$$\begin{aligned}
\lambda\beta \vdash e_1 = e_2 \quad &\textit{iff} \quad \lambda\beta_v \vdash \mathcal{C}_n\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}_n\langle\!\langle e_2\rangle\!\rangle\\
&\textit{iff} \quad \lambda\beta \vdash \mathcal{C}_n\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}_n\langle\!\langle e_2\rangle\!\rangle\\
&\textit{iff} \quad \lambda\beta_v \vdash \mathcal{C}_n\langle\!\langle e_1\rangle\!\rangle\, I = \mathcal{C}_n\langle\!\langle e_2\rangle\!\rangle\, I\\
&\textit{iff} \quad \lambda\beta \vdash \mathcal{C}_n\langle\!\langle e_1\rangle\!\rangle\, I = \mathcal{C}_n\langle\!\langle e_2\rangle\!\rangle\, I
\end{aligned}$$

[1] In the context of Parigot's $\lambda\mu$-calculus [25], de Groote independently noted the problem with Plotkin's **Translation** theorem and proposed a similar correction [10].

The **Indifference** and **Translation** properties remain the same. The **Simulation** property, however, holds up to β_i-equivalence while Plotkin's **Simulation** for $\mathcal{P}_\mathrm{n}$ holds up to α-equivalence. For example,

$$\mathcal{C}_\mathrm{n}\langle eval_\mathrm{n}((\lambda z.\lambda y.z)\,b)\rangle \;=\; \lambda y.\lambda k.k\,b$$

whereas

$$eval_\mathrm{v}(\mathcal{C}_\mathrm{n}\langle\!\langle(\lambda z.\lambda y.z)\,b\rangle\!\rangle\,I) \;=\; \lambda y.\lambda k.(\lambda k.k\,b)\,k.$$

In fact, proofs of **Indifference**, **Simulation**, and most of the **Translation** can be derived from the correctness properties of $\mathcal{C}_\mathrm{v}^{+}$ and $\mathcal{T}$ (see Section 5). All that remains of **Translation** is to show that $\lambda\beta \;\vdash\; \mathcal{C}_\mathrm{n}\langle\!\langle e_1\rangle\!\rangle\,I \;=\; \mathcal{C}_\mathrm{n}\langle\!\langle e_2\rangle\!\rangle\,I$ implies $\lambda\beta \;\vdash\; e_1 \;=\; e_2$ and this follows in a straightforward manner from Plotkin's original proof for $\mathcal{P}_\mathrm{n}$ [15, p. 31]. The following theorem gives the **Indifference**, **Simulation**, and **Translation** properties for $\mathcal{C}_\mathrm{v}$.

Theorem 5 (Plotkin 1975) *For all $e \;\in\; Programs[\Lambda]$ and $e_1, e_2 \;\in\; \Lambda$,*

1. **Indifference:** $eval_\mathrm{n}(\mathcal{C}_\mathrm{v}\langle\!\langle e\rangle\!\rangle\,I) \simeq\; eval_\mathrm{v}(\mathcal{C}_\mathrm{v}\langle\!\langle e\rangle\!\rangle\,I)$
2. **Simulation:** $\mathcal{C}_\mathrm{v}\langle eval_\mathrm{v}(e)\rangle \simeq\; eval_\mathrm{n}(\mathcal{C}_\mathrm{v}\langle\!\langle e\rangle\!\rangle\,I)$
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 **Translation** property states that β_v-convertible terms are also convertible in the image of $\mathcal{C}_\mathrm{v}$. In contrast to the theory $\lambda\beta$ appearing in the **Translation** property for $\mathcal{C}_\mathrm{n}$ (Theorem 4), the theory $\lambda\beta_\mathrm{v}$ is *incomplete* in the sense that it cannot prove the equivalence of some terms whose CPS images are provably equivalent using $\lambda\beta$ or $\lambda\beta_\mathrm{v}$ [32]. The properties of $\mathcal{C}_\mathrm{v}$ as stated in Theorem 5 can be extended to the transformation $\mathcal{C}_\mathrm{v}^{+}$ defined on the language T — the set of terms in the image of $\mathcal{T}$ closed under $\beta_i\tau$ reduction. It is straightforward to show that the following grammar generates exactly the set of terms T [15, pp. 32,33].

$$t \quad ::= \quad b \;\;|\;\; force\; x \;\;|\;\; force\,(delay\; t) \;\;|\;\; \lambda x.t \;\;|\;\; t_0\,(delay\; t_1)$$

Theorem 6 *For all $t \in Programs[T]$ and $t_1, t_2 \in T$,*

1. **Indifference:** $eval_{\mathrm{n}}(\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t\rangle\!\rangle\, I) \simeq eval_{\mathrm{v}}(\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t\rangle\!\rangle\, I)$
2. **Simulation:** $\mathcal{C}_{\mathrm{v}}^{+}\langle eval_{\mathrm{v}}(t)\rangle \simeq eval_{\mathrm{n}}(\mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t\rangle\!\rangle\, I)$
3. **Translation:**
   If $\lambda\beta_{\mathrm{v}}\tau \vdash t_1 = t_2$ *then* $\lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t_1\rangle\!\rangle = \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t_2\rangle\!\rangle$
   Also $\lambda\beta_{\mathrm{v}} \vdash \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t_1\rangle\!\rangle = \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t_2\rangle\!\rangle$ *iff* $\lambda\beta \vdash \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t_1\rangle\!\rangle = \mathcal{C}_{\mathrm{v}}^{+}\langle\!\langle t_2\rangle\!\rangle$

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 $\mathcal{C}_{\mathrm{v}}$ [27, pp. 148–152]. **Translation** follows from the **Translation** component of Theorem 5 and Property 1 [15, p. 39]. ∎

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 4) in terms of $\mathcal{T}$. The last two assertions of the **Translation** component of Theorem 4 do not appear here since the identity function as the initial continuation only plays a rôle in CPS evaluation.

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

1. **Indifference:** $eval_{\mathrm{v}}(\mathcal{T}\langle\!\langle e\rangle\!\rangle) \simeq eval_{\mathrm{n}}(\mathcal{T}\langle\!\langle e\rangle\!\rangle)$
2. **Simulation:** $\mathcal{T}\langle\!\langle eval_{\mathrm{n}}(e)\rangle\!\rangle \simeq_{\tau} eval_{\mathrm{v}}(\mathcal{T}\langle\!\langle e\rangle\!\rangle)$
3. **Translation:** $\lambda\beta \vdash e_1 = e_2$ *iff* $\lambda\beta_{\mathrm{v}}\tau \vdash \mathcal{T}\langle\!\langle e_1\rangle\!\rangle = \mathcal{T}\langle\!\langle e_2\rangle\!\rangle$
   iff $\lambda\beta\tau \vdash \mathcal{T}\langle\!\langle e_1\rangle\!\rangle = \mathcal{T}\langle\!\langle e_2\rangle\!\rangle$

Proof: The proof of **Indifference** is trivial: one can intuitively see from the grammar for T (which includes the set of terms in the image of $\mathcal{T}$ closed under evaluation steps) that call-by-name and call-by-value evaluation will coincide since all function arguments are values.

The proof of **Simulation** is somewhat involved. It begins by inductively defining a relation $\overset{\mathcal{T}}{\sim} \subseteq \Lambda \times \Lambda_{\tau}$ such that $e \overset{\mathcal{T}}{\sim} t$ holds exactly when $\lambda\tau \vdash \mathcal{T}\langle\!\langle e\rangle\!\rangle = t$. The crucial step is then to show that for all $e \in Programs[\Lambda]$ and $t \in Programs[\Lambda_{\tau}]$ such that $e \overset{\mathcal{T}}{\sim} t$, $e \longmapsto_{\mathrm{n}} e'$ implies that there exists a t' such that $t \longmapsto_{\mathrm{v}}^{+} t'$ and $e' \overset{\mathcal{T}}{\sim} t'$ [15, Sect. 2.3.2].

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

Figure 8: Thunk introduction implemented in Λ

Translation is established by first defining a translation $\mathcal{T}^{-1} : \Lambda_\tau \rightarrow \Lambda$ that simply removes *delay* and *force* constructs. One then shows that $\mathcal{T}$ and $\mathcal{T}^{-1}$ establish an equational correspondence [32] (or more precisely a *reflection* [33]) between theories $\lambda\beta$ and $\lambda\beta_{\mathrm{v}}\tau$ (and $\lambda\beta$ and $\lambda\beta\tau$). **Translation** follows as a corollary of this stronger result [15, Sect. 2.3.3]. ∎

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 Section 3. Elsewhere [14], 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].

Figure 8 presents the transformation $\mathcal{T}_{\mathcal{L}}$ that implements thunks directly in Λ using what Plotkin described as the "protecting by a λ" technique [27, p. 147]. An expression is delayed by wrapping it in an abstraction with a dummy parameter. A thunk is forced by applying it to a dummy argument.

The following theorem recasts the correctness theorem for $\mathcal{C}_{\mathrm{n}}$ (Theorem 4) in terms of $\mathcal{T}_{\mathcal{L}}$.

Theorem 8 *For all $e \in \mathit{Programs}[\Lambda]$ and $e_1, e_2 \in \Lambda$,*

1. **Indifference:** $eval_{\mathrm{v}}(\mathcal{T}_{\mathcal{L}}\langle\!\langle e \rangle\!\rangle) \simeq eval_{\mathrm{n}}(\mathcal{T}_{\mathcal{L}}\langle\!\langle e \rangle\!\rangle)$
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: Follows the same pattern as the proof of Theorem 7 [15, Sect. 2.4]. ∎

5 Applications

5.1 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 1. Here we show only the results where evaluation is undefined or results in a basic constant b. See [15, p. 31] for a derivation of $\mathcal{C}_{\mathrm{n}}$ **Simulation** for arbitrary results.

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

$$\begin{array}{lll}
& eval_{\mathrm{v}}(\mathcal{C}_{\mathrm{n}}\langle\!\!\langle e \rangle\!\!\rangle\,(\lambda y.y)) \;=\; b & \\
\Leftrightarrow & eval_{\mathrm{v}}((\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\!\langle e \rangle\!\!\rangle\,(\lambda y.y)) \;=\; b & \textit{...Theorem 1 and the soundness of } \beta_{\mathrm{v}} \\
\Leftrightarrow & eval_{\mathrm{n}}((\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\!\langle e \rangle\!\!\rangle\,(\lambda y.y)) \;=\; b & \textit{...Theorem 6 } \textbf{(Indifference)} \\
\Leftrightarrow & eval_{\mathrm{n}}(\mathcal{C}_{\mathrm{n}}\langle\!\!\langle e \rangle\!\!\rangle\,(\lambda y.y)) \;=\; b & \textit{...Theorem 1 and the soundness of } \beta
\end{array}$$

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

$$\begin{array}{lll}
& eval_{\mathrm{n}}(e) \;=\; b & \\
\Leftrightarrow & eval_{\mathrm{v}}(\mathcal{T}\langle\!\!\langle e \rangle\!\!\rangle) \;=\; b & \textit{...Theorem 7 } \textbf{(Simulation)} \\
\Leftrightarrow & eval_{\mathrm{n}}((\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\!\langle e \rangle\!\!\rangle\,(\lambda y.y)) \;=\; b & \textit{...Theorem 6 } \textbf{(Simulation)} \\
\Leftrightarrow & eval_{\mathrm{v}}((\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\!\langle e \rangle\!\!\rangle\,(\lambda y.y)) \;=\; b & \textit{...Theorem 6 } \textbf{(Indifference)} \\
\Leftrightarrow & eval_{\mathrm{v}}(\mathcal{C}_{\mathrm{n}}\langle\!\!\langle e \rangle\!\!\rangle\,(\lambda y.y)) \;=\; b & \textit{...Theorem 1 and the soundness of } \beta_{\mathrm{v}}
\end{array}$$

For **Translation**, it is not possible to establish Theorem 4 (**Translation** for $\mathcal{C}_{\mathrm{n}}$) in the manner above since Theorem 6 (**Translation** for $\mathcal{C}_{\mathrm{v}}^{+}$) is weaker in comparison. However, the following weaker version can be derived. Let $e_1, e_2 \in \Lambda$. Then

$$\begin{array}{lll}
& \lambda\beta \vdash e_1 \;=\; e_2 & \\
\Leftrightarrow & \lambda\beta_{\mathrm{v}}\tau \vdash \mathcal{T}\langle\!\!\langle e_1 \rangle\!\!\rangle \;=\; \mathcal{T}\langle\!\!\langle e_2 \rangle\!\!\rangle & \textit{...Theorem 7 } \textbf{(Translation)} \\
\Rightarrow & \lambda\beta_i \vdash (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\!\langle e_1 \rangle\!\!\rangle \;=\; (\mathcal{C}_{\mathrm{v}}^{+} \circ \mathcal{T})\langle\!\!\langle e_2 \rangle\!\!\rangle & \textit{...Theorem 6 } \textbf{(Translation)} \\
\Leftrightarrow & \lambda\beta_i \vdash \mathcal{C}_{\mathrm{n}}\langle\!\!\langle e_1 \rangle\!\!\rangle \;=\; \mathcal{C}_{\mathrm{n}}\langle\!\!\langle e_2 \rangle\!\!\rangle & \textit{...Theorem 1} \\
\Rightarrow & \lambda\beta_i \vdash \mathcal{C}_{\mathrm{n}}\langle\!\!\langle e_1 \rangle\!\!\rangle\, I \;=\; \mathcal{C}_{\mathrm{n}}\langle\!\!\langle e_2 \rangle\!\!\rangle\, I & \textit{...compatibility of } =_{\beta_i}
\end{array}$$

5.2 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 [4, 21, 24]. In an earlier work [9], we gave a transformation $\mathcal{T}_s$ that optimizes thunk introduction based on strictness information. 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 staged approach can be contrasted with Burn and Le Métayer's monolithic strategy [5].

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

Amtoft [1] and Steckler and Wand [34] have proven the correctness of transformations which optimize the introduction of thunks based on strictness information.

5.3 Deriving a call-by-need CPS transformation

Okasaki, Lee, and Tarditi [24] 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 with memo-thunks [4]. $\mathcal{C}_{need}$ 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 it with the memo-thunk introduction.

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.

5.4 Alternative CPS transformations

Thunks can be used to factor a variety of call-by-name CPS transformations. In addition to those discussed here, one can factor a variant of Reynolds's CPS transformation directed by strictness information [14, 30], as well as a call-by-name analogue of Fischer's call-by-value CPS transformation [11, 32].

Obtaining the desired call-by-name CPS transformation *via* $\mathcal{C}_v^+$ and $\mathcal{T}$ depends on the representation of thunks. For example, if one works with $\mathcal{T}_\mathcal{L}$ (see Figure 8) instead of $\mathcal{T}$, $\mathcal{C}_v \circ \mathcal{T}_\mathcal{L}$ still gives a valid CPS simulation of call-by-name by call-by-value. However, β_i equivalence with $\mathcal{C}_n$ is not obtained (*i.e.*, $\lambda\beta_i \not\vdash \mathcal{C}_n\langle\!\langle e\rangle\!\rangle = (\mathcal{C}_v \circ \mathcal{T}_\mathcal{L})\langle\!\langle e\rangle\!\rangle$), as shown by the following derivations.

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

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

The representation of thunks given by $\mathcal{T}_\mathcal{L}$ is too concrete in the sense that the delaying and forcing of computation is achieved using specific instances of the more general abstraction and application constructs. When composed with $\mathcal{T}_\mathcal{L}$, $\mathcal{C}_v$ 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*.

5.5 The factorization holds for types

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 [12, 13, 18, 20]. The thunk transformation $\mathcal{T}$ also preserves well-typedness of terms, and the relationship between $\mathcal{C}_v^+ \circ \mathcal{T}$ and $\mathcal{C}_n$ is reflected in transformations on types [15, Sect. 4].

6 Related Work

Ingerman [16], 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 [29]. 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 [4] study thunks as the basis of an 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 [31] 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 for thunks. 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). 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 formulate thunks in a categorical setting [2, 6]. Two combinators *freeze* and *unfreeze*, which are analogous to *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.

In his original paper [27, p. 147], 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." [27, p. 147]. By "protecting by a λ", Plotkin refers to a representation of thunks as λ-abstractions with a dummy parameter, as in Figure 8. In a set of unpublished notes, however, he later showed that the "protecting by a λ" technique is sufficient for a complete simulation [28].

An earlier version of Section 3 appeared in the proceedings of WSA'92 [8]. Most of these proofs have been checked in Elf [26] by Niss and the first author [23]. Elsewhere [14], we also consider an optimizing version of $\mathcal{T}$ that does not introduce thunks for identifiers occurring as function arguments:

$$\mathcal{T}_{opt}\langle\!\langle e\, x \rangle\!\rangle \quad = \quad \mathcal{T}_{opt}\langle\!\langle e \rangle\!\rangle\, x$$

$\mathcal{T}_{opt}$ generates a language T_{opt} which is more refined than T (referred to in Theorem 6).

Finally, Lawall and Danvy investigate staging the call-by-value CPS transformation into conceptually different passes elsewhere [17].

7 Conclusion

We have connected the traditional thunk-based simulation $\mathcal{T}$ of call-by-name under call-by-value and Plotkin's continuation-based simulations $\mathcal{C}_{\mathrm{n}}$ and $\mathcal{C}_{\mathrm{v}}$ of call-by-name and call-by-value. Almost all of the technical properties Plotkin established for $\mathcal{C}_{\mathrm{n}}$ follow from the properties of $\mathcal{T}$ and $\mathcal{C}_{\mathrm{v}}^{+}$ (the extension of $\mathcal{C}_{\mathrm{v}}$ to thunks). When reasoning about $\mathcal{C}_{\mathrm{n}}$ and $\mathcal{C}_{\mathrm{v}}$, it is thus often sufficient to reason about $\mathcal{C}_{\mathrm{v}}^{+}$ and the simpler simulation $\mathcal{T}$. We have also given several applications involving deriving optimized continuation-based simulations for call-by-name and call-by-need languages and performing CPS transformation after static program analysis.

Acknowledgements

Andrzej Filinski, Sergey Kotov, Julia Lawall, Henning Niss, and David Schmidt gave helpful comments on earlier drafts of this paper. Thanks are also due to Dave Sands for several useful discussions. Special thanks to Gordon Plotkin for enlightening conversations at the LDPL'95 workshop and for subsequently mailing us his unpublished course notes. Finally, we are grateful to the reviewers for their lucid comments and their exhortation to be more concise, and to our editors, for their encouragement and direction.

The commuting diagram was drawn with Kristoffer Rose's XY-pic package.

References

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

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

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

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

[5] Geoffrey Burn and Daniel Le Métayer. Proving the correctness of compiler optimisations based on a global program analysis. *Journal of Functional Programming*, 6(1), 1996.

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

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

[8] Olivier Danvy and John Hatcliff. Thunks (continued). In *Proceedings of the Second International Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 3–11, Bordeaux, France, September 1992. IRISA, Rennes, France.

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

[10] Philippe de Groote. A CPS-translation of the $\lambda\mu$-calculus. In Sophie Tison, editor, *19th Colloquium on Trees in Algebra and Programming (CAAP'94)*, number 787 in Lecture Notes in Computer Science, pages 47–58, Edinburgh, Scotland, April 1994.

[11] Michael J. Fischer. Lambda-calculus schemata. In Talcott [36], pages 259–288. An earlier version appeared in an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.

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

[13] Bob Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Talcott [36].

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

[15] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. Technical Report 95/3, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, February 1995.

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

[17] Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 124–136, Charleston, South Carolina, January 1993. ACM Press.

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

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

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

[21] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1981.

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

[23] Henning Niss and John Hatcliff. Encoding operational semantics in logical frameworks: A critical review of LF/Elf. In Bengt Nördstrom, editor, *Proceedings of the 1995 Workshop on Programming Language Theory*, Göteborg, Sweden, November 1995.

[24] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. In Carolyn L. Talcott, editor, *Special issue on continuations (Part II)*, LISP and Symbolic Computation, Vol. 7, No. 1, pages 57–81. Kluwer Academic Publishers, January 1994.

[25] Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, number 624 in Lecture Notes in Artificial Intelligence, pages 190–201, St. Petersburg, Russia, July 1992.

[26] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

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

[28] Gordon D. Plotkin. Course notes on operational semantics. Unpublished manuscript, 1978.

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

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

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

[32] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Talcott [36], pages 289–360.

[33] Amr Sabry and Philip Wadler. Compiling with reflections. In R. Kent Dybvig, editor, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pennsylvania, May 1996. ACM Press.

[34] Paul Steckler and Mitchell Wand. Selective thunkification. In Baudouin Le Charlier, editor, *Static Analysis*, number 864 in Lecture Notes in Computer Science, pages 162–178, Namur, Belgium, September 1994.

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

[36] Carolyn L. Talcott, editor. *Special issue on continuations (Part I)*, LISP and Symbolic Computation, Vol. 6, Nos. 3/4. Kluwer Academic Publishers, December 1993.

Recent Publications in the BRICS Report Series

RS-96-19 **John Hatcliff and Olivier Danvy.** ***Thunks and the λ-Calculus*****. June 1996. 22 pp. To appear in *Journal of Functional Programming*.**

RS-96-18 **Thomas Troels Hildebrandt and Vladimiro Sassone.** ***Comparing Transition Systems with Independence and Asynchronous Transition Systems*****. June 1996. 14 pp. To appear in Montanari and Sassone, editors, *Concurrency Theory: 7th International Conference*, CONCUR '96 Proceedings, LNCS 1119, 1996.**

RS-96-17 **Olivier Danvy, Karoline Malmkjær, and Jens Palsberg.** ***Eta-Expansion Does The Trick (Revised Version)*****. May 1996. 29 pp. To appear in *ACM Transactions on Programming Languages and Systems (TOPLAS)*.**

RS-96-16 **Lisbeth Fajstrup and Martin Raußen.** ***Detecting Deadlocks in Concurrent Systems*****. May 1996. 10 pp.**

RS-96-15 **Olivier Danvy.** ***Pragmatic Aspects of Type-Directed Partial Evaluation*****. May 1996. 27 pp.**

RS-96-14 **Olivier Danvy and Karoline Malmkjær.** ***On the Idempotence of the CPS Transformation*****. May 1996. 15 pp.**

RS-96-13 **Olivier Danvy and René Vestergaard.** ***Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation*****. May 1996. 28 pp. To appear in *8th International Symposium on Programming Languages, Implementations, Logics, and Programs*, PLILP '96 Proceedings, LNCS, 1996.**

RS-96-12 **Lars Arge, Darren E. Vengroff, and Jeffrey S. Vitter.** ***External-Memory Algorithms for Processing Line Segments in Geographic Information Systems*****. May 1996. 34 pp. A shorter version of this paper appears in Spirakis, editor, *Algorithms - ESA '95: Third Annual European Symposium Proceedings*, LNCS 979, 1995, pages 295–310.**