CPS-Transformation After Strictness Analysis

OLIVIER DANVY and JOHN HATCLIFF
Kansas State University

Strictness analysis is a common component of compilers for call-by-name functional languages; the continuation-passing-style (CPS-) transformation is a common component of compilers for call-by-value functional languages. To bridge these two implementation techniques, we present a hybrid CPS-transformation $\mathscr{C}_s$ for a language with annotations resulting from strictness analysis. $\mathscr{C}_s$ is derived by symbolically composing two transformations $\mathscr{S}$ and $\mathscr{C}_v$; that is

$$\mathscr{C}_s \stackrel{\text{def}}{=} \mathscr{C}_v \circ \mathscr{S}.$$

—$\mathscr{S}$ transforms a call-by-name program with strictness annotations into a call-by-value program extended with explicit suspension constructs (i.e., *delay* and *force*).
—$\mathscr{C}_v$ is the traditional call-by-value CPS-transformation extended with transformations for the suspension constructs *delay* and *force*.
—$\mathscr{C}_s$ generalizes both the call-by-name and the call-by-value CPS-transformations, in that restricting it to nonstrict constructs gives the call-by-name CPS-transformation and restricting it to strict constructs gives the call-by-value CPS-transformation.

$\mathscr{C}_s$ enables a new strategy for compiling call-by-name programs combining the traditional advantages of CPS (tail-recursive code, evaluation-order independence) and the usual benefits of strictness analysis (elimination of unnecessary suspensions). We also address and solve the problem of recursive binding. Finally, we express $\mathscr{C}_s$ in Nielson and Nielson's two-level λ-calculus, enabling a simple and efficient implementation in a functional programming language.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*syntax*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; D.3.3 [**Programming Languages**]: Language Constructs—*control structures*; *procedures, functions and subroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*functional constructs*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

General Terms: Languages

Additional Key Words and Phrases: Call-by-name, call-by-value, continuation-passing-style transformation, λ-calculus, strictness analysis

This work was partly supported by the NSF under grant CCR-9102625.
Authors' address: Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506.

1. INTRODUCTION

1.1 Implementing Call-by-Name Programs

In a call-by-name (CBN) functional program, parameter passing obeys the copy rule. However, the copy rule is rarely used to implement CBN, for efficiency reasons. Instead, since Algol 60 [11], CBN parameter passing is implemented by passing suspensions using call-by-value (CBV). Suspensions improve over the copy rule, but they need to be created for each procedure call and activated for each identifier lookup. This overhead can be reduced using strictness analysis, which tells one where CBV binding of suspensions can be safely changed to CBV binding of values [17].

1.2 Implementing Call-by-Value Programs

Continuation-passing style (CPS) is commonly used to implement CBV functional programs [2, 22]. CPS offers practical benefits [2, pp. 4–6]. For example, the control flow of the evaluation strategy is explicitly encoded, and the code is tail-recursive.

However, CBN functional programs are not usually implemented with the CBN CPS-transformation. Also, we do not know of any strictness analysis applied after CPS-transformation (a moot point, since CPS terms are evaluation-order independent [19]).

1.3 Toward a Mixed Implementation

A program with strictness annotations specifies both CBN and CBV parameter passing. In a previous paper [6], we showed that the CBN CPS-transformation can be factored into two transformations: (1) the introduction of suspensions, and (2) the CBV CPS-transformation. In the present work, we use strictness information to decide where it is possible to leave out suspensions, and we derive a new CPS-transformation for strictness-annotated programs.

Specifically, we first encode a CBN program with strictness annotations into a CBV program with explicit operations over suspensions. Then we transform this CBV program into CPS, extending the CBV CPS-transformation to deal with suspensions. The main result of this paper is the direct mapping of a program with strictness annotations into a CPS program without strictness annotations. This mapping is obtained by symbolically composing the encoding of strictness properties using suspensions with the extended CBV CPS-transformation and then by simplifying the result of the symbolic composition.

This new CPS-transformation enables a new strategy for compiling CBN programs, combining the traditional advantages of CPS (tail-recursive code, evaluation-order independence) and the usual benefits of strictness analysis (elimination of unnecessary suspensions). This transformation should prove useful in several areas:

—A CPS-based compiler can process the output of any CPS-transformation. Yet, existing CPS-based compilers, for example, Appel's Standard ML

compiler [2] and Steele's Scheme compiler [22], process CBV programs. Our new CPS-transformation enables one to use an existing CPS-based compiler to compile CBN programs, including optimizations enabled by strictness analysis.

—Programs can also be compiled using program-transformation techniques. This approach is used by Kelsey and Hudak [12] and by Fradet and Le Métayer [10]. Both include a CPS-transformation. Fradet and Le Métayer compile both CBN and CBV programs by using the CBN and the CBV CPS-transformations. Recently, Burn and Le Métayer [3] have combined this technique with a global program analysis, which is comparable to our goal here.

1.4 Overview

Section 2 presents the syntax of the source language and the strictness-annotated language. We consider a typed language because it enables us to express the well-formedness of a strictness-annotated term concisely. Section 3 defines two program transformations: encoding strictness annotations in a CBV language with suspension operators, and extending the CBV CPS-transformation to deal with suspensions. Section 4 formally derives the CPS-transformation for programs with strictness annotations. Section 5 addresses recursion. Section 6 presents a "well-staged" CPS-transformation that eliminates administrative overhead. Finally, Section 7 concludes and puts this work into perspective.

2. THE LANGUAGE

Figure 1 presents the syntax of the source language: an applied and typed λ-calculus. Applications are tagged with @, for annotation purposes. Recursive binding (with *fix*) is treated in Section 5. For simplicity, we assume that all identifiers are unique. Figure 2 gives the usual type-checking rules, where ι denotes base types (Boolean, etc.).

Figure 3 presents the syntax of the language with annotations resulting from strictness analysis. We use the diacritical accents ´ and ` to denote strict and nonstrict constructs, respectively. The important points of the figure are as follows:

—Annotations of identifiers indicate that they are declared as formal parameters of strict or of nonstrict λ-abstractions or *let* expressions.

—λ-abstractions may be strict or nonstrict depending on whether or not the functions they represent are strict or nonstrict, respectively.

—Strict (resp., nonstrict) applications denote the application of strict (resp., nonstrict) λ-abstractions.

—Conditional expressions are strict in their test.

—Pairing may be strict or nonstrict in either argument. The constructor *páir* is strict on both arguments. *pàir* is nonstrict on both arguments. *pâir* is strict on its first argument and nonstrict on its second. *păir* is nonstrict on its first argument and strict on its second.

$$
\begin{array}{lcl}
e & \in & \mathit{Exp} \\
e & ::= & c \mid x \mid \lambda x{:}\tau\,.\,e \mid @\,e_0\,e_1 \mid \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid \mathit{let}\ x = e_0\ \mathit{in}\ e_1 \\
 & & \mid \mathit{pair}\ e_1\ e_2 \mid \mathit{fst}\ e \mid \mathit{snd}\ e \mid \mathit{fix}\ e \\
 & & \\
\tau & \in & \mathit{Typ} \\
\tau & ::= & \iota \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2
\end{array}
$$

Fig. 1. Abstract syntax of the source language.

$$
\vdash c : \iota \qquad \{\ldots, x : \tau, \ldots\} \vdash x : \tau \qquad \frac{\pi \vdash e : \tau \to \tau}{\pi \vdash \mathit{fix}\ e : \tau}
$$

$$
\frac{\pi \cup \{x : \tau_1\} \vdash e : \tau_2}{\pi \vdash \lambda x{:}\tau_1\,.\,e : \tau_1 \to \tau_2} \qquad \frac{\pi \vdash e_0 : \tau_1 \to \tau_2 \quad \pi \vdash e_1 : \tau_1}{\pi \vdash @\,e_0\,e_1 : \tau_2}
$$

$$
\frac{\pi \vdash e_1 : \iota \quad \pi \vdash e_2 : \tau \quad \pi \vdash e_3 : \tau}{\pi \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : \tau} \qquad \frac{\pi \vdash e_0 : \tau_0 \quad \pi \cup \{x : \tau_0\} \vdash e_1 : \tau_1}{\pi \vdash \mathit{let}\ x = e_0\ \mathit{in}\ e_1 : \tau_1}
$$

$$
\frac{\pi \vdash e_1 : \tau_1 \quad \pi \vdash e_2 : \tau_2}{\pi \vdash \mathit{pair}\ e_1\ e_2 : \tau_1 \times \tau_2} \qquad \frac{\pi \vdash e : \tau_1 \times \tau_2}{\pi \vdash \mathit{fst}\ e : \tau_1} \qquad \frac{\pi \vdash e : \tau_1 \times \tau_2}{\pi \vdash \mathit{snd}\ e : \tau_2}
$$

Fig. 2. Type-checking rules for the source language.

$$
\begin{array}{lcl}
e & \in & \mathit{AExp} \\
e & ::= & c \mid \acute{x} \mid \grave{x} \mid \acute{\lambda} x{:}\tau\,.\,e \mid \grave{\lambda} x{:}\tau\,.\,e \mid \acute{@}\,e_0\,e_1 \mid \grave{@}\,e_0\,e_1 \\
 & & \mid \acute{\mathit{if}}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid \acute{\mathit{let}}\ x = e_0\ \mathit{in}\ e_1 \mid \grave{\mathit{let}}\ x = e_0\ \mathit{in}\ e_1 \\
 & & \mid \acute{\mathit{pair}}\ e_0\ e_1 \mid \hat{\mathit{pair}}\ e_0\ e_1 \mid \check{\mathit{pair}}\ e_0\ e_1 \mid \grave{\mathit{pair}}\ e_0\ e_1 \\
 & & \mid \acute{\mathit{fst}}\ e \mid \grave{\mathit{fst}}\ e \mid \acute{\mathit{snd}}\ e \mid \grave{\mathit{snd}}\ e \\
 & & \\
\tau & \in & \mathit{ATyp} \\
\tau & ::= & \iota \mid \tau_1 \acute{\to} \tau_2 \mid \tau_1 \grave{\to} \tau_2 \mid \tau_1 \acute{\times} \tau_2 \mid \tau_1 \hat{\times} \tau_2 \mid \tau_1 \check{\times} \tau_2 \mid \tau_1 \grave{\times} \tau_2
\end{array}
$$

Fig. 3. Abstract syntax of the source language with strictness annotations.

—The projections *fst* and *snd* are strict constructs. Their argument must denote a pair. They are annotated as strict ($\acute{\mathit{fst}}$, $\acute{\mathit{snd}}$) or nonstrict ($\acute{\mathit{fst}}$, $\grave{\mathit{snd}}$) depending on whether the corresponding pair constructor was strict or nonstrict on the first or second component.

We assume that the strictness analyzer guarantees the well-formedness of the annotations. This well-formedness is captured in the type-checking rules of Figure 4.

$$\vdash c : \iota \qquad \{..., \acute{x} : \tau, ...\} \vdash \acute{x} : \tau \qquad \{..., \grave{x} : \tau, ...\} \vdash \grave{x} : \tau$$

$$\frac{\pi \cup \{\acute{x} : \tau_1\} \vdash e : \tau_2}{\pi \vdash \acute{\lambda} x : \tau_1 . e : \tau_1 \acute{\rightarrow} \tau_2} \qquad \frac{\pi \cup \{\grave{x} : \tau_1\} \vdash e : \tau_2}{\pi \vdash \grave{\lambda} x : \tau_1 . e : \tau_1 \grave{\rightarrow} \tau_2}$$

$$\frac{\pi \vdash e_1 : \iota \qquad \pi \vdash e_2 : \tau \qquad \pi \vdash e_3 : \tau}{\pi \vdash \acute{if}\ e_1\ then\ e_2\ else\ e_3 : \tau}$$

$$\frac{\pi \vdash e_0 : \tau_1 \acute{\rightarrow} \tau_2 \qquad \pi \vdash e_1 : \tau_1}{\pi \vdash \acute{@}\, e_0\, e_1 : \tau_2} \qquad \frac{\pi \vdash e_0 : \tau_1 \grave{\rightarrow} \tau_2 \qquad \pi \vdash e_1 : \tau_1}{\pi \vdash \grave{@}\, e_0\, e_1 : \tau_2}$$

$$\frac{\pi \vdash e_0 : \tau_0 \qquad \pi \cup \{\acute{x} : \tau_0\} \vdash e_1 : \tau_1}{\pi \vdash \acute{let}\ x = e_0\ in\ e_1 : \tau_1} \qquad \frac{\pi \vdash e_0 : \tau_0 \qquad \pi \cup \{\grave{x} : \tau_0\} \vdash e_1 : \tau_1}{\pi \vdash \grave{let}\ x = e_0\ in\ e_1 : \tau_1}$$

$$\frac{\pi \vdash e_0 : \tau_0 \qquad \pi \vdash e_1 : \tau_1}{\pi \vdash \acute{pair}\ e_0\ e_1 : \tau_0 \acute{\times} \tau_1} \qquad \frac{\pi \vdash e_0 : \tau_0 \qquad \pi \vdash e_1 : \tau_1}{\pi \vdash \hat{pair}\ e_0\ e_1 : \tau_0 \hat{\times} \tau_1}$$

$$\frac{\pi \vdash e_0 : \tau_0 \qquad \pi \vdash e_1 : \tau_1}{\pi \vdash \check{pair}\ e_0\ e_1 : \tau_0 \check{\times} \tau_1} \qquad \frac{\pi \vdash e_0 : \tau_0 \qquad \pi \vdash e_1 : \tau_1}{\pi \vdash \grave{pair}\ e_0\ e_1 : \tau_0 \grave{\times} \tau_1}$$

$$\frac{\pi \vdash e : \tau_0 \acute{\times} \tau_1}{\pi \vdash \acute{fst}\ e : \tau_0} \qquad \frac{\pi \vdash e : \tau_0 \hat{\times} \tau_1}{\pi \vdash \hat{fst}\ e : \tau_0} \qquad \frac{\pi \vdash e : \tau_0 \check{\times} \tau_1}{\pi \vdash \check{fst}\ e : \tau_0} \qquad \frac{\pi \vdash e : \tau_0 \grave{\times} \tau_1}{\pi \vdash \grave{fst}\ e : \tau_0}$$

$$\frac{\pi \vdash e : \tau_0 \acute{\times} \tau_1}{\pi \vdash \acute{snd}\ e : \tau_1} \qquad \frac{\pi \vdash e : \tau_0 \hat{\times} \tau_1}{\pi \vdash \hat{snd}\ e : \tau_1} \qquad \frac{\pi \vdash e : \tau_0 \check{\times} \tau_1}{\pi \vdash \check{snd}\ e : \tau_1} \qquad \frac{\pi \vdash e : \tau_0 \grave{\times} \tau_1}{\pi \vdash \grave{snd}\ e : \tau_1}$$

Fig. 4. Type-checking rules of the source language with strictness annotations.

3. THE PROGRAM TRANSFORMATIONS

3.1 Encoding Strictness Properties with Suspension Constructs

Based on the idea of thunks [11], we translate the language with strictness annotations to a CBV language extended with the suspension constructs *delay* and *force*. To correspond to the addition of suspension operators in the syntax, we add the type constructor $\tilde{}$. Then a type $\tilde{\tau}$ denotes the type of a suspension that would yield a value of type τ. Figures 5 and 6 give the syntax of this language and its type-checking rules, respectively.

$$
\begin{array}{lcl}
e & \in & \mathit{DFExp} \\
e & ::= & c \mid x \mid \lambda x{:}\tau\,.\,e \mid @\,e_0\,e_1 \mid \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid \mathit{let}\ x = e_0\ \mathit{in}\ e_1 \\
 & & \mid \mathit{pair}\ e_1\ e_2 \mid \mathit{fst}\ e \mid \mathit{snd}\ e \mid \mathit{delay}\ e \mid \mathit{force}\ e \\
 & & \\
\tau & \in & \mathit{DFTyp} \\
\tau & ::= & \iota \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \widetilde{\tau}
\end{array}
$$

Fig. 5. Abstract syntax of the CBV language with suspension constructs.

$$
\vdash c : \iota \qquad \{\ldots, x : \sigma, \ldots\} \vdash x : \sigma \qquad \frac{\pi \vdash e : \tau}{\pi \vdash \mathit{delay}\ e : \widetilde{\tau}} \qquad \frac{\pi \vdash e : \widetilde{\tau}}{\pi \vdash \mathit{force}\ e : \tau}
$$

$$
\frac{\pi \cup \{x : \sigma\} \vdash e : \tau}{\pi \vdash \lambda x{:}\sigma\,.\,e : \sigma \to \tau} \qquad \frac{\pi \vdash e_0 : \sigma \to \tau \qquad \pi \vdash e_1 : \sigma}{\pi \vdash @\,e_0\,e_1 : \tau}
$$

$$
\frac{\pi \vdash e_1 : \iota \qquad \pi \vdash e_2 : \tau \qquad \pi \vdash e_3 : \tau}{\pi \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : \tau} \qquad \frac{\pi \vdash e_0 : \sigma \qquad \pi \cup \{x : \sigma\} \vdash e_1 : \tau}{\pi \vdash \mathit{let}\ x = e_0\ \mathit{in}\ e_1 : \tau}
$$

$$
\frac{\pi \vdash e_1 : \sigma_1 \qquad \pi \vdash e_2 : \sigma_2}{\pi \vdash \mathit{pair}\ e_1\ e_2 : \sigma_1 \times \sigma_2} \qquad \frac{\pi \vdash e : \sigma_1 \times \sigma_2}{\pi \vdash \mathit{fst}\ e : \sigma_1} \qquad \frac{\pi \vdash e : \sigma_1 \times \sigma_2}{\pi \vdash \mathit{snd}\ e : \sigma_2}
$$

Fig. 6. Type-checking rules for the CBV language with suspension constructs.

Figure 7 presents the translation S. There is nothing surprising in this translation; all strict constructs are essentially copied, while nonstrict constructs have their arguments suspended by *delay*. Evaluation of expressions yielding suspensions is initiated using *force*.

The following proposition required us to write the formal semantics of the source language with strictness annotations ($\mathscr{P}_{AExp}$) and of the CBV language with suspension constructs ($\mathscr{P}_{DFExp}$):

PROPOSITION 1. *For any program $p \in AExp$ (i.e., closed expression of ground type),*

$$\mathscr{P}_{AExp}[\![p]\!] = \mathscr{P}_{DFExp} \circ \mathscr{S}[\![p]\!].$$

PROOF. Routine [15, 21]. In other words, $\mathscr{S}$ is meaning-preserving. □

3.2 The CBV CPS-Transformation

Figure 8 presents Plotkin's CBV CPS-transformation $\mathscr{C}_v$ extended to our applied λ-calculus [5, 13, 18]. $\mathscr{C}_v$ is a syntax-to-syntax translation yielding terms whose meanings are independent of evaluation order [19]. The result of

$$
\begin{array}{rcl}
\mathcal{S} & : & AExp \to DFExp \\
\mathcal{S}[\![\acute{x}]\!] & = & x \\
\mathcal{S}[\![\acute{\lambda}\, x : \tau \,.\, e]\!] & = & \lambda\, x{:}\mathcal{S}[\![\tau]\!] \,.\, \mathcal{S}[\![e]\!] \\
\mathcal{S}[\![\acute{@}\, e_0\, e_1]\!] & = & @\, \mathcal{S}[\![e_0]\!]\, \mathcal{S}[\![e_1]\!] \\
\mathcal{S}[\![\acute{let}\, x = e_0\ in\ e_1]\!] & = & let\ x = \mathcal{S}[\![e_0]\!] \\
 & & in\ \mathcal{S}[\![e_1]\!]^{\bullet} \\
\mathcal{S}[\![\acute{pair}\, e_0\, e_1]\!] & = & pair\ \mathcal{S}[\![e_0]\!] \\
 & & \quad \mathcal{S}[\![e_1]\!] \\
\mathcal{S}[\![\hat{pair}\, e_0\, e_1]\!] & = & pair\ \mathcal{S}[\![e_0]\!] \\
 & & \quad (delay\ \mathcal{S}[\![e_1]\!]) \\
\mathcal{S}[\![\acute{fst}\, e]\!] & = & fst\ \mathcal{S}[\![e]\!] \\
\mathcal{S}[\![\acute{snd}\, e]\!] & = & snd\ \mathcal{S}[\![e]\!]
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{S}[\![c]\!] & = & c \\
\mathcal{S}[\![\grave{x}]\!] & = & force\ x \\
\mathcal{S}[\![\grave{\lambda}\, x : \tau \,.\, e]\!] & = & \lambda\, x{:}\widetilde{\mathcal{S}[\![\tau]\!]} \,.\, \mathcal{S}[\![e]\!] \\
\mathcal{S}[\![\grave{@}\, e_0\, e_1]\!] & = & @\, \mathcal{S}[\![e_0]\!]\, (delay\ \mathcal{S}[\![e_1]\!]) \\
\mathcal{S}[\![\grave{let}\, x = e_0\ in\ e_1]\!] & = & let\ x = delay\ \mathcal{S}[\![e_0]\!] \\
 & & in\ \mathcal{S}[\![e_1]\!] \\
\mathcal{S}[\![\grave{pair}\, e_0\, e_1]\!] & = & pair\ (delay\ \mathcal{S}[\![e_0]\!]) \\
 & & \quad (delay\ \mathcal{S}[\![e_1]\!]) \\
\mathcal{S}[\![\check{pair}\, e_0\, e_1]\!] & = & pair\ (delay\ \mathcal{S}[\![e_0]\!]) \\
 & & \quad \mathcal{S}[\![e_1]\!] \\
\mathcal{S}[\![\grave{fst}\, e]\!] & = & force\ (fst\ \mathcal{S}[\![e]\!]) \\
\mathcal{S}[\![\grave{snd}\, e]\!] & = & force\ (snd\ \mathcal{S}[\![e]\!])
\end{array}
$$

$$
\mathcal{S}[\![\acute{if}\ e_0\ then\ e_1\ else\ e_2]\!] \;=\; if\ \mathcal{S}[\![e_0]\!]\ then\ \mathcal{S}[\![e_1]\!]\ else\ \mathcal{S}[\![e_2]\!]
$$

$$
\begin{array}{rcl}
\mathcal{S} & : & ATyp \to DFTyp \\
\mathcal{S}[\![\tau_1 \acute{\to} \tau_2]\!] & = & \mathcal{S}[\![\tau_1]\!] \to \mathcal{S}[\![\tau_2]\!] \\
\mathcal{S}[\![\tau_1 \acute{\times} \tau_2]\!] & = & \mathcal{S}[\![\tau_1]\!] \times \mathcal{S}[\![\tau_2]\!] \\
\mathcal{S}[\![\tau_1 \hat{\times} \tau_2]\!] & = & \mathcal{S}[\![\tau_1]\!] \times \widetilde{\mathcal{S}[\![\tau_2]\!]}
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{S}[\![\iota]\!] & = & \iota \\
\mathcal{S}[\![\tau_1 \grave{\to} \tau_2]\!] & = & \widetilde{\mathcal{S}[\![\tau_1]\!]} \to \mathcal{S}[\![\tau_2]\!] \\
\mathcal{S}[\![\tau_1 \grave{\times} \tau_2]\!] & = & \widetilde{\mathcal{S}[\![\tau_1]\!]} \times \widetilde{\mathcal{S}[\![\tau_2]\!]} \\
\mathcal{S}[\![\tau_1 \check{\times} \tau_2]\!] & = & \widetilde{\mathcal{S}[\![\tau_1]\!]} \times \mathcal{S}[\![\tau_2]\!]
\end{array}
$$

Fig. 7. Encoding strictness properties with suspension operators.

$$
\begin{array}{rcl}
\mathcal{C}_{\mathrm{v}} & : & Exp \to Exp \\
\mathcal{C}_{\mathrm{v}}[\![c]\!] & = & \lambda\, \kappa \,.\, @\, \kappa\, c \\
\mathcal{C}_{\mathrm{v}}[\![x]\!] & = & \lambda\, \kappa \,.\, @\, \kappa\, x \\
\mathcal{C}_{\mathrm{v}}[\![\lambda\, x{:}\tau \,.\, e]\!] & = & \lambda\, \kappa \,.\, @\, \kappa\, (\lambda\, x{:}\mathcal{C}_{\mathrm{v}}[\![\tau]\!] \,.\, \mathcal{C}_{\mathrm{v}}[\![e]\!]) \\
\mathcal{C}_{\mathrm{v}}[\![@\, e_0\, e_1]\!] & = & \lambda\, \kappa \,.\, @\, \mathcal{C}_{\mathrm{v}}[\![e_0]\!]\, (\lambda\, v_0 \,.\, @\, \mathcal{C}_{\mathrm{v}}[\![e_1]\!]\, (\lambda\, v_1 \,.\, @\, (@\, v_0\, v_1)\, \kappa)) \\
\mathcal{C}_{\mathrm{v}}[\![if\ e_0\ then\ e_1\ else\ e_2]\!] & = & \lambda\, \kappa \,.\, @\, \mathcal{C}_{\mathrm{v}}[\![e_0]\!]\, (\lambda\, v_0 \,.\, let\ \kappa' = \kappa \\
 & & \qquad in\ if\ v_0\ then\ @\, \mathcal{C}_{\mathrm{v}}[\![e_1]\!]\, \kappa'\ else\ @\, \mathcal{C}_{\mathrm{v}}[\![e_2]\!]\, \kappa') \\
\mathcal{C}_{\mathrm{v}}[\![let\ x = e_0\ in\ e_1]\!] & = & \lambda\, \kappa \,.\, @\, \mathcal{C}_{\mathrm{v}}[\![e_0]\!]\, (\lambda\, v_0 \,.\, let\ x = v_0\ in\ @\, \mathcal{C}_{\mathrm{v}}[\![e_1]\!]\, \kappa) \\
\mathcal{C}_{\mathrm{v}}[\![pair\ e_0\, e_1]\!] & = & \lambda\, \kappa \,.\, @\, \mathcal{C}_{\mathrm{v}}[\![e_0]\!]\, (\lambda\, v_0 \,.\, @\, \mathcal{C}_{\mathrm{v}}[\![e_1]\!]\, (\lambda\, v_1 \,.\, @\, \kappa\, (pair\ v_0\, v_1))) \\
\mathcal{C}_{\mathrm{v}}[\![fst\ e]\!] & = & \lambda\, \kappa \,.\, @\, \mathcal{C}_{\mathrm{v}}[\![e]\!]\, (\lambda\, v \,.\, @\, \kappa\, (fst\ v)) \\
\mathcal{C}_{\mathrm{v}}[\![snd\ e]\!] & = & \lambda\, \kappa \,.\, @\, \mathcal{C}_{\mathrm{v}}[\![e]\!]\, (\lambda\, v \,.\, @\, \kappa\, (snd\ v))
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_{\mathrm{v}} & : & Typ \to Typ \\
\mathcal{C}_{\mathrm{v}}[\![\iota]\!] & = & \iota \\
\mathcal{C}_{\mathrm{v}}[\![\tau_1 \to \tau_2]\!] & = & \mathcal{C}_{\mathrm{v}}[\![\tau_1]\!] \to (\mathcal{C}_{\mathrm{v}}[\![\tau_2]\!] \to Ans) \to Ans \\
\mathcal{C}_{\mathrm{v}}[\![\tau_1 \times \tau_2]\!] & = & \mathcal{C}_{\mathrm{v}}[\![\tau_1]\!] \times \mathcal{C}_{\mathrm{v}}[\![\tau_2]\!]
\end{array}
$$

Fig. 8. CBV CPS-transformation.

transforming a term e into CPS in an empty context (represented by the identity function) is given by

$$@\mathscr{C}_{\mathrm{v}}[\![e]\!](\lambda v.v),$$

where the v's are fresh variables. *Ans* denotes the type of final answers.

Given a CBV semantics $\mathscr{P}_{Exp}$, the following proposition captures the correctness of $\mathscr{C}_{\mathrm{v}}$:

PROPOSITION 2. *For any program $p \in Exp$ (i.e., closed expression of ground type),*

$$\mathscr{P}_{Exp}[\![p]\!] = \mathscr{P}_{Exp}[\![@\mathscr{C}_{\mathrm{v}}[\![p]\!](\lambda v.v)]\!].$$

PROOF. Routine [18]. Plotkin's original proof was concerned with an untyped language. Meyer and Wand pointed out that Plotkin's proposition holds in a typed setting [13]. □

3.3 CPS-Transformation of Suspension Constructs

3.3.1 *Terms.* Let us determine the analogue of *delay* in CPS. Suspensions are traditionally formed by transforming an expression into a parameterless procedure (or into a procedure with a dummy parameter). However, this encoding is not necessary in the CPS world because all expressions are passed a continuation. Therefore, to delay the evaluation of an expression, it is sufficient to deprive it of its continuation and to pass this value to the current continuation. This naturally leads us to the following definition of *delay*:

$$\mathscr{C}_{\mathrm{v}}[\![delay\, e]\!] = \lambda\kappa.@\kappa\mathscr{C}_{\mathrm{v}}[\![e]\!].$$

Since suspending an evaluation is achieved by depriving an expression of its continuation, forcing a CPS suspension is achieved by supplying it with a continuation.

The definition of *force* follows from the definition of *delay*, since we desire $force \circ delay = Id$. In terms of the CPS-transformation, $\mathscr{C}_{\mathrm{v}}[\![force\ (delay\ e)]\!]$ and $\mathscr{C}_{\mathrm{v}}[\![e]\!]$ should be interconvertible. This naturally leads us to the following definition of *force*:

$$\mathscr{C}_{\mathrm{v}}[\![force\, e]\!] = \lambda\kappa.@\mathscr{C}_{\mathrm{v}}[\![e]\!](\lambda v.@v\kappa).$$

The derivation below verifies that the transformation of *force* has the desired property:

$$\begin{aligned}
\mathscr{C}_{\mathrm{v}}[\![force\ (delay\, e)]\!] &= \lambda\kappa.@\mathscr{C}_{\mathrm{v}}[\![delay\, e]\!](\lambda v.@v\kappa) && \text{Definition of } force\\
&= \lambda\kappa.@\big(\lambda\kappa.@\kappa\mathscr{C}_{\mathrm{v}}[\![e]\!]\big)(\lambda v.@v\kappa) && \text{Definition of } delay\\
&\Rightarrow \lambda\kappa.@\mathscr{C}_{\mathrm{v}}[\![e]\!]\kappa && \beta\text{-reduction (twice)}\\
&\Rightarrow \mathscr{C}_{\mathrm{v}}[\![e]\!] && \eta\text{-reduction.}
\end{aligned}$$

The arrow $\Rightarrow$ is used to denote valid CBV reduction steps. For example, the two β-reductions preserve CBV meaning since the arguments are λ-

expressions (i.e., values). Similarly, the η-reduction is valid since $\mathscr{C}_{\mathrm{v}}[\![e]\!]$ yields a λ-expression. The two new clauses above extend the CBV CPS-transformation over terms (cf. Figure 8).

3.3.2 *Types.* The transformation on terms is nicely reflected on types. We only need to accommodate the type-constructor $\tilde{}$. Since a suspension is a CPS term deprived of its continuation, we extend the CBV CPS-transformation over types (cf. Figure 8) with the following clause:

$$\mathscr{C}_{\mathrm{v}}[\![\tilde{\tau}]\!] = \left(\mathscr{C}_{\mathrm{v}}[\![\tau]\!] \to Ans\right) \to Ans.$$

3.3.3 *Conclusion.* The correctness of $\mathscr{C}_{\mathrm{v}}$ extends to the transformation of suspension constructs.

PROPOSITION 3. *For any program $p \in DFExp$ (i.e., closed expression of ground type),*

$$\mathscr{P}_{DFExp}[\![p]\!] = \mathscr{P}_{Exp}[\![@\mathscr{C}_{\mathrm{v}}[\![p]\!](\lambda v.v)]\!].$$

PROOF. Routine. □

4. CPS-TRANSFORMATION OF A LANGUAGE WITH STRICTNESS ANNOTATIONS

We now derive a new CPS-transformation $\mathscr{C}_{\mathrm{s}}$ by composing $\mathscr{C}_{\mathrm{v}}$ with $\mathscr{S}$ symbolically and by simplifying the result, inductively. The induction is performed first over the structure of terms and next over the structure of types. The induction hypotheses read as follows:

$$\begin{cases} \forall e \in AExp, \mathscr{C}_{\mathrm{v}} \circ \mathscr{S}[\![e]\!] = \mathscr{C}_{\mathrm{s}}[\![e]\!], \\ \forall \tau \in ATyp, \mathscr{C}_{\mathrm{v}} \circ \mathscr{S}[\![\tau]\!] = \mathscr{C}_{\mathrm{s}}[\![\tau]\!]. \end{cases}$$

4.1 Terms

We will make use of the following property:

Property 1. $@\mathscr{C}_{\mathrm{v}}[\![delay\ e]\!](\lambda v.\ldots) \Rightarrow @(\lambda v.\ldots)\mathscr{C}_{\mathrm{v}}[\![e]\!]$.

Let us now derive the CPS-transformation of four interesting constructs:

$$\begin{aligned} \mathscr{C}_{\mathrm{s}}[\![\dot{x}]\!] &= \mathscr{C}_{\mathrm{v}} \circ \mathscr{S}[\![\dot{x}]\!] \\ &= \mathscr{C}_{\mathrm{v}}[\![force\ x]\!] \\ &= \lambda\kappa.@\mathscr{C}_{\mathrm{v}}[\![x]\!](\lambda v.@v\kappa) \\ &\Rightarrow \lambda\kappa.@x\kappa \qquad \beta\text{-reduction (twice)} \\ &\Rightarrow x \qquad \eta\text{-reduction,} \end{aligned}$$

$$
\begin{aligned}
\mathscr{C}_s[\![\grave{@}e_0 e_1]\!] &= \mathscr{C}_v \circ \mathscr{S}[\![\grave{@}e_0 e_1]\!] \\
&= \mathscr{C}_v[\![@\mathscr{S}[\![e_0]\!](delay\ \mathscr{S}[\![e_1]\!])]\!] \\
&= \lambda\kappa.@(\mathscr{C}_v \circ \mathscr{S}[\![e_0]\!])(\lambda v_0.@\mathscr{C}_v[\![delay\ \mathscr{S}[\![e_1]\!]]\!](\lambda v_1.@(@v_0 v_1)\kappa)) \\
&\Rightarrow \lambda\kappa.@(\mathscr{C}_v \circ \mathscr{S}[\![e_0]\!]) && \text{Property 1} \\
&\qquad (\lambda v_0.@(\lambda v_1.@(@v_0 v_1)\kappa)(\mathscr{C}_v \circ \mathscr{S}[\![e_1]\!])) \\
&\Rightarrow \lambda\kappa.@(\mathscr{C}_v \mathscr{S}[\![e_0]\!])(\lambda v_0.@(@v_0(\mathscr{C}_v \mathscr{S}[\![e_1]\!]))\kappa) && \beta\text{-reduction} \\
&= \lambda\kappa.@\mathscr{C}_s[\![e_0]\!](\lambda v_0.@(@v_0\mathscr{C}_s[\![e_1]\!])\kappa) && \text{Induction}
\end{aligned}
$$

$$
\begin{aligned}
\mathscr{C}_s[\![pàir\, e_0 e_1]\!] &= \mathscr{C}_v \circ \mathscr{S}[\![pàir\, e_0 e_1]\!] \\
&= \mathscr{C}_v[\![pair\ (delay\ \mathscr{S}[\![e_0]\!])(delay\ \mathscr{S}[\![e_1]\!])]\!] \\
&= \lambda\kappa.@\mathscr{C}_v[\![delay\ \mathscr{S}[\![e_0]\!]]\!] \\
&\qquad (\lambda v_0.@\mathscr{C}_v[\![delay\ \mathscr{S}[\![e_1]\!]]\!] \\
&\qquad\qquad (\lambda v_1.@\kappa(pair\ v_0 v_1))) \\
&\Rightarrow \lambda\kappa.@(\lambda v_0.@(\lambda v_1.@\kappa(pair\ v_0 v_1)) && \text{Property 1 (twice)} \\
&\qquad\qquad (\mathscr{C}_v \circ \mathscr{S}[\![e_1]\!])) \\
&\qquad (\mathscr{C}_v \circ \mathscr{S}[\![e_0]\!]) \\
&\Rightarrow \lambda\kappa.@\kappa(pair\ (\mathscr{C}_v \circ \mathscr{S}[\![e_0]\!])(\mathscr{C}_v \circ \mathscr{S}[\![e_1]\!])) \\
&= \lambda\kappa.@\kappa(pair\ \mathscr{C}_s[\![e_0]\!]\mathscr{C}_s[\![e_1]\!])
\end{aligned}
$$

$$
\begin{aligned}
\mathscr{C}_s[\![\hat{fst}\, e]\!] &= \mathscr{C}_v \circ \mathscr{S}[\![\hat{fst}\, e_0]\!] \\
&= \mathscr{C}_v[\![force\ (fst\ \mathscr{S}[\![e]\!])]\!] \\
&= \lambda\kappa.@\mathscr{C}_v[\![fst\ \mathscr{S}[\![e]\!]]\!](\lambda v.@v\kappa) \\
&= \lambda\kappa.@(\lambda\kappa.@(\mathscr{C}_v \circ \mathscr{S}[\![e]\!])(\lambda v.@\kappa(fst\ v)))(\lambda v.@v\kappa) \\
&\Rightarrow \lambda\kappa.@(\mathscr{C}_v \circ \mathscr{S}[\![e]\!])(\lambda v.@(\lambda v.@v\kappa)(fst\ v)) \\
&\Rightarrow \lambda\kappa.@(\mathscr{C}_v \circ \mathscr{S}[\![e]\!])(\lambda v.@(fst\ v)\kappa) \\
&= \lambda\kappa.@\mathscr{C}_s[\![e]\!](\lambda v.@(fst\ v)\kappa)
\end{aligned}
$$

The full transformation is given in Figure 10. Again, the result of transforming an annotated term e into CPS in an empty context (represented by the identity function) is given by

$$@\mathscr{C}_s[\![e]\!](\lambda v.v),$$

where the v's are fresh variables. A careful examination of Figure 10 reveals that elements of both $\mathscr{C}_n$ and $\mathscr{C}_v$ are captured in $\mathscr{C}_s$. Essentially, the transformation of a nonstrict construct follows the pattern of the CBN CPS-transformation (cf. Figure 9), and the transformation of a strict construct follows the pattern of the CBV CPS-transformation (cf. Figure 8). This observation scales up to types.

4.2 Types

We also obtain $\mathscr{C}_s$ on types by symbolically composing $\mathscr{C}_v$ and $\mathscr{S}$. The derivations for four interesting constructs are given below:

$$
\begin{aligned}
\mathcal{C}_{\mathrm{n}} &: \mathit{Exp} \to \mathit{Exp} \\
\mathcal{C}_{\mathrm{v}}[\![c]\!] &= \lambda\kappa\,.\,@\,\kappa\,c \\
\mathcal{C}_{\mathrm{n}}[\![x]\!] &= x \\
\mathcal{C}_{\mathrm{n}}[\![\lambda x:\tau\,.\,e]\!] &= \lambda\kappa\,.\,@\,\kappa\,(\lambda x:(\mathcal{C}_{\mathrm{n}}[\![\tau]\!]\to \mathit{Ans})\to \mathit{Ans}\,.\,\mathcal{C}_{\mathrm{n}}[\![e]\!]) \\
\mathcal{C}_{\mathrm{n}}[\![@\,e_0\,e_1]\!] &= \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{n}}[\![e_0]\!]\,(\lambda v_0\,.\,@\,(@\,v_0\,\mathcal{C}_{\mathrm{n}}[\![e_1]\!])\,\kappa) \\
\mathcal{C}_{\mathrm{n}}[\![\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2]\!] &= \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{n}}[\![e_0]\!]\,(\lambda v_0\,.\,\mathit{let}\ \kappa'=\kappa \\
&\qquad\qquad \mathit{in\ if}\ v_0\ \mathit{then}\ @\,\mathcal{C}_{\mathrm{n}}[\![e_1]\!]\,\kappa'\ \mathit{else}\ @\,\mathcal{C}_{\mathrm{n}}[\![e_2]\!]\,\kappa') \\
\mathcal{C}_{\mathrm{n}}[\![\mathit{let}\ x=e_0\ \mathit{in}\ e_1]\!] &= \lambda\kappa\,.\,\mathit{let}\ x=\mathcal{C}_{\mathrm{n}}[\![e_0]\!]\ \mathit{in}\ @\,\mathcal{C}_{\mathrm{n}}[\![e_1]\!]\,\kappa \\
\mathcal{C}_{\mathrm{n}}[\![\mathit{pair}\ e_0\ e_1]\!] &= \lambda\kappa\,.\,@\,\kappa\,(\mathit{pair}\ \mathcal{C}_{\mathrm{n}}[\![e_0]\!]\,\mathcal{C}_{\mathrm{n}}[\![e_1]\!]) \\
\mathcal{C}_{\mathrm{n}}[\![\mathit{fst}\ e]\!] &= \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{n}}[\![e]\!]\,(\lambda v\,.\,@\,(\mathit{fst}\ v)\,\kappa) \\
\mathcal{C}_{\mathrm{n}}[\![\mathit{snd}\ e]\!] &= \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{n}}[\![e]\!]\,(\lambda v\,.\,@\,(\mathit{snd}\ v)\,\kappa)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_{\mathrm{n}} &: \mathit{Typ} \to \mathit{Typ} \\
\mathcal{C}_{\mathrm{n}}[\![\iota]\!] &= \iota \\
\mathcal{C}_{\mathrm{n}}[\![\tau_1\to\tau_2]\!] &= ((\mathcal{C}_{\mathrm{n}}[\![\tau_1]\!]\to \mathit{Ans})\to \mathit{Ans})\to(\mathcal{C}_{\mathrm{n}}[\![\tau_2]\!]\to \mathit{Ans})\to \mathit{Ans} \\
\mathcal{C}_{\mathrm{n}}[\![\tau_1\times\tau_2]\!] &= ((\mathcal{C}_{\mathrm{n}}[\![\tau_1]\!]\to \mathit{Ans})\to \mathit{Ans})\times((\mathcal{C}_{\mathrm{n}}[\![\tau_2]\!]\to \mathit{Ans})\to \mathit{Ans})
\end{aligned}
$$

Fig. 9. CBN CPS-transformation.

$$
\begin{aligned}
\mathscr{C}_{\mathrm{s}}[\![\iota]\!] &= \mathscr{C}_{\mathrm{v}}\circ\mathscr{S}[\![\iota]\!] \\
&= \iota
\end{aligned}
$$

$$
\begin{aligned}
\mathscr{C}_{\mathrm{s}}[\![\tau_1 \stackrel{\rightarrow}{\rightarrow} \tau_2]\!] &= \mathscr{C}_{\mathrm{v}}\circ\mathscr{S}[\![\tau_1 \stackrel{\rightarrow}{\rightarrow} \tau_2]\!] \\
&= \mathscr{C}_{\mathrm{v}}[\![\mathscr{S}[\![\tau_1]\!]\to\mathscr{S}[\![\tau_2]\!]]\!] \\
&= \mathscr{C}_{\mathrm{v}}\circ\mathscr{S}[\![\tau_1]\!]\to(\mathscr{C}_{\mathrm{v}}\circ\mathscr{S}[\![\tau_2]\!]\to \mathit{Ans})\to \mathit{Ans} \\
&= \mathscr{C}_{\mathrm{s}}[\![\tau_1]\!]\to(\mathscr{C}_{\mathrm{s}}[\![\tau_2]\!]\to \mathit{Ans})\to \mathit{Ans},
\end{aligned}
$$

$$
\begin{aligned}
\mathscr{C}_{\mathrm{s}}[\![\tau_1 \stackrel{\rightarrow}{\rightarrow} \tau_2]\!] &= \mathscr{C}_{\mathrm{v}}\circ\mathscr{S}[\![\tau_1 \to \tau_2]\!] \\
&= \mathscr{C}_{\mathrm{v}}[\![\mathscr{S}[\![\tau_1]\!]\to\mathscr{S}[\![\tau_2]\!]]\!] \\
&= ((\mathscr{C}_{\mathrm{v}}\circ\mathscr{S}[\![\tau_1]\!]\to \mathit{Ans})\to \mathit{Ans})\to(\mathscr{C}_{\mathrm{v}}\circ\mathscr{S}[\![\tau_2]\!]\to \mathit{Ans})\to \mathit{Ans} \\
&= ((\mathscr{C}_{\mathrm{s}}[\![\tau_1]\!]\to \mathit{Ans})\to \mathit{Ans})\to(\mathscr{C}_{\mathrm{s}}[\![\tau_2]\!]\to \mathit{Ans})\to \mathit{Ans},
\end{aligned}
$$

$$
\begin{aligned}
\mathscr{C}_{\mathrm{s}}[\![\tau_1 \hat{\times} \tau_2]\!] &= \mathscr{C}_{\mathrm{v}}\circ\mathscr{S}[\![\tau_1 \hat{\times} \tau_2]\!] \\
&= \mathscr{C}_{\mathrm{v}}[\![\mathscr{S}[\![\tau_1]\!]\times\mathscr{S}[\![\tau_2]\!]]\!] \\
&= \mathscr{C}_{\mathrm{v}}\circ\mathscr{S}[\![\tau_1]\!]\times((\mathscr{C}_{\mathrm{v}}\circ\mathscr{S}[\![\tau_2]\!]\to \mathit{Ans})\to \mathit{Ans}) \\
&= \mathscr{C}_{\mathrm{s}}[\![\tau_1]\!]\times((\mathscr{C}_{\mathrm{s}}[\![\tau_2]\!]\to \mathit{Ans})\to \mathit{Ans}).
\end{aligned}
$$

As can be observed, the transformation of a strict function follows the pattern of the CBV CPS-transformation (cf. Figure 8), and the transformation of a nonstrict function follows the pattern of the CBN CPS-transformation (cf.

$$
\begin{array}{rcl}
\mathcal{C}_{\mathrm{s}} & : & AExp \to Exp \\
\mathcal{C}_{\mathrm{s}}[\![c]\!] & = & \lambda\kappa\,.\,@\,\kappa\,c \\
\mathcal{C}_{\mathrm{s}}[\![\acute{x}]\!] & = & \lambda\kappa\,.\,@\,\kappa\,x \\
\mathcal{C}_{\mathrm{s}}[\![\grave{x}]\!] & = & x \\
\mathcal{C}_{\mathrm{s}}[\![\acute{\lambda}\,x:\tau\,.\,e]\!] & = & \lambda\kappa\,.\,@\,\kappa\,(\lambda\,x{:}\mathcal{C}_{\mathrm{s}}[\![\tau]\!]\,.\,\mathcal{C}_{\mathrm{s}}[\![e]\!]) \\
\mathcal{C}_{\mathrm{s}}[\![\grave{\lambda}\,x:\tau\,.\,e]\!] & = & \lambda\kappa\,.\,@\,\kappa\,(\lambda\,x{:}(\mathcal{C}_{\mathrm{s}}[\![\tau]\!]\to Ans)\to Ans\,.\,\mathcal{C}_{\mathrm{s}}[\![e]\!]) \\
\mathcal{C}_{\mathrm{s}}[\![\acute{@}\,e_0\,e_1]\!] & = & \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e_0]\!]\,(\lambda\,v_0\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e_1]\!]\,(\lambda\,v_1\,.\,@\,(@\,v_0\,v_1)\,\kappa)) \\
\mathcal{C}_{\mathrm{s}}[\![\grave{@}\,e_0\,e_1]\!] & = & \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e_0]\!]\,(\lambda\,v_0\,.\,@\,(@\,v_0\,\mathcal{C}_{\mathrm{s}}[\![e_1]\!])\,\kappa) \\
\mathcal{C}_{\mathrm{s}}[\![\acute{if}\,e_0\;then\;e_1\;else\;e_2]\!] & = & \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e_0]\!]\,(\lambda\,v_0\,.\,let\;\kappa'=\kappa \\
 & & \qquad in\;if\;v_0\;then\;@\,\mathcal{C}_{\mathrm{s}}[\![e_1]\!]\,\kappa'\;else\;@\,\mathcal{C}_{\mathrm{s}}[\![e_2]\!]\,\kappa') \\
\mathcal{C}_{\mathrm{s}}[\![\acute{let}\,x=e_0\;in\;e_1]\!] & = & \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e_0]\!]\,(\lambda\,v_0\,.\,let\;x=v_0\;in\;@\,\mathcal{C}_{\mathrm{s}}[\![e_1]\!]\,\kappa) \\
\mathcal{C}_{\mathrm{s}}[\![\grave{let}\,x=e_0\;in\;e_1]\!] & = & \lambda\kappa\,.\,let\;x=\mathcal{C}_{\mathrm{s}}[\![e_0]\!]\;in\;@\,\mathcal{C}_{\mathrm{s}}[\![e_1]\!]\,\kappa \\
\mathcal{C}_{\mathrm{s}}[\![\acute{pair}\,e_0\,e_1]\!] & = & \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e_0]\!]\,(\lambda\,v_0\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e_1]\!]\,(\lambda\,v_1\,.\,@\,\kappa\,(pair\;v_0\;v_1))) \\
\mathcal{C}_{\mathrm{s}}[\![\hat{pair}\,e_0\,e_1]\!] & = & \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e_0]\!]\,(\lambda\,v_0\,.\,@\,\kappa\,(pair\;v_0\,\mathcal{C}_{\mathrm{s}}[\![e_1]\!])) \\
\mathcal{C}_{\mathrm{s}}[\![\check{pair}\,e_0\,e_1]\!] & = & \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e_1]\!]\,(\lambda\,v_1\,.\,@\,\kappa\,(pair\,\mathcal{C}_{\mathrm{s}}[\![e_0]\!]\,v_1)) \\
\mathcal{C}_{\mathrm{s}}[\![\grave{pair}\,e_0\,e_1]\!] & = & \lambda\kappa\,.\,@\,\kappa\,(pair\,\mathcal{C}_{\mathrm{s}}[\![e_0]\!]\,\mathcal{C}_{\mathrm{s}}[\![e_1]\!]) \\
\mathcal{C}_{\mathrm{s}}[\![\acute{fst}\,e]\!] & = & \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e]\!]\,(\lambda\,v\,.\,@\,\kappa\,(fst\;v)) \\
\mathcal{C}_{\mathrm{s}}[\![\grave{fst}\,e]\!] & = & \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e]\!]\,(\lambda\,v\,.\,@\,(fst\;v)\,\kappa) \\
\mathcal{C}_{\mathrm{s}}[\![\acute{snd}\,e]\!] & = & \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e]\!]\,(\lambda\,v\,.\,@\,\kappa\,(snd\;v)) \\
\mathcal{C}_{\mathrm{s}}[\![\grave{snd}\,e]\!] & = & \lambda\kappa\,.\,@\,\mathcal{C}_{\mathrm{s}}[\![e]\!]\,(\lambda\,v\,.\,@\,(snd\;v)\,\kappa)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_{\mathrm{s}} & : & ATyp \to Typ \\
\mathcal{C}_{\mathrm{s}}[\![\iota]\!] & = & \iota \\
\mathcal{C}_{\mathrm{s}}[\![\tau_1 \acute{\to} \tau_2]\!] & = & \mathcal{C}_{\mathrm{s}}[\![\tau_1]\!]\to(\mathcal{C}_{\mathrm{s}}[\![\tau_2]\!]\to Ans)\to Ans \\
\mathcal{C}_{\mathrm{s}}[\![\tau_1 \grave{\to} \tau_2]\!] & = & ((\mathcal{C}_{\mathrm{s}}[\![\tau_1]\!]\to Ans)\to Ans)\to(\mathcal{C}_{\mathrm{s}}[\![\tau_2]\!]\to Ans)\to Ans \\
\mathcal{C}_{\mathrm{s}}[\![\tau_1 \acute{\times} \tau_2]\!] & = & \mathcal{C}_{\mathrm{s}}[\![\tau_1]\!]\times\mathcal{C}_{\mathrm{s}}[\![\tau_2]\!] \\
\mathcal{C}_{\mathrm{s}}[\![\tau_1 \hat{\times} \tau_2]\!] & = & \mathcal{C}_{\mathrm{s}}[\![\tau_1]\!]\times((\mathcal{C}_{\mathrm{s}}[\![\tau_2]\!]\to Ans)\to Ans) \\
\mathcal{C}_{\mathrm{s}}[\![\tau_1 \check{\times} \tau_2]\!] & = & ((\mathcal{C}_{\mathrm{s}}[\![\tau_1]\!]\to Ans)\to Ans)\times\mathcal{C}_{\mathrm{s}}[\![\tau_2]\!] \\
\mathcal{C}_{\mathrm{s}}[\![\tau_1 \grave{\times} \tau_2]\!] & = & ((\mathcal{C}_{\mathrm{s}}[\![\tau_1]\!]\to Ans)\to Ans)\times((\mathcal{C}_{\mathrm{s}}[\![\tau_2]\!]\to Ans)\to Ans)
\end{array}
$$

Fig. 10. CPS-transformation for a language with strictness annotations.

Figure 9). The same observation applies for strict pairs and for nonstrict pairs.

4.3 Conclusion

We have derived a CPS-transformation $\mathscr{C}_{\mathrm{s}}$ of strictness-annotated terms by composing the introduction of suspensions with (an extension of) the CBV CPS-transformation. The correctness of $\mathscr{C}_{\mathrm{s}}$ follows from the correctness of $\mathscr{S}$ (cf. Proposition 1) and of $\mathscr{C}_{\mathrm{v}}$ (cf. Proposition 3).

THEOREM 1. *For any program* $p \in AExp$ *(i.e., closed expression of ground type),*

$$\mathscr{P}_{AExp}[\![p]\!] = \mathscr{P}_{Exp}[\![@\mathscr{C}_s[\![p]\!](\lambda v.v)]\!].$$

Let us analyze two extreme cases:

(1) If the strictness analysis determines that *all* program constructs are strict, then no suspension is introduced, and $\mathscr{C}_s$ can be simplified into $\mathscr{C}_v$. Indeed, a simple inspection of $\mathscr{C}_s$ (cf. Figure 10) shows that its restriction to the syntax of strict programs coincides with $\mathscr{C}_v$ (cf. Figure 8).

(2) If the strictness analysis determines that *no* program construct is strict, then suspensions are introduced everywhere, and $\mathscr{C}_s$ can be simplified into $\mathscr{C}_n$. Indeed, a simple inspection of $\mathscr{C}_s$ shows that its restriction to the syntax of nonstrict programs coincides with $\mathscr{C}_n$ (cf. Figure 9).

Thus, $\mathscr{C}_s$ generalizes both $\mathscr{C}_v$ and $\mathscr{C}_n$.

The second extreme case coincides with our earlier result [6]: The CBN CPS-transformation can be factored into (1) introduction of thunks $\mathscr{T}$ and (2) CBV CPS-transformation:

$$\mathscr{C}_n = \mathscr{C}_v \circ \mathscr{T}.$$

5. TRANSFORMATION OF RECURSIVE BINDINGS

As, for example, Filinski points out [9], recursive definitions in a CBV language are often restricted to functions and syntactically sugared with *letrec*. In contrast, arbitrary expressions can be defined recursively in a CBN language. Therefore, we cannot naively translate strictness-annotated fixpoint operations into CBV recursive definitions.

Let us begin by pointing out that *fix*, in Figure 1, is a strict construct, just like *fst* and *snd*. Its argument must denote a function. This other function may be strict or nonstrict, which we know, courtesy of the strictness analysis. Therefore (and, again, just as for *fst* and *snd*), we annotate *fix* with the accent ´ when it is applied to an expression denoting a strict function and with ` when it is applied to a nonstrict function. This leads to the following definitions, which extend the syntax definition of Figure 3 and the type-checking rules of Figure 4:

$$e \in AExp \qquad \frac{\pi \vdash e : \tau \stackrel{\prime}{\to} \tau}{\pi \vdash \acute{f}ix\, e : \tau} \qquad \frac{\pi \vdash e : \tau \stackrel{`}{\to} \tau}{\pi \vdash \grave{f}ix\, e : \tau}$$

$$e ::= \cdots \mid \acute{f}ix\, e \mid \grave{f}ix\, e$$

5.1 Strict Functions

We first treat the obvious: *fix* is a diverging construct, since it should obey the usual unfolding property (unfolding is noted $\mathscr{U}$ in the following diagrams):

$$\acute{f}ix\, e \stackrel{u}{\to} \acute{@}e(\acute{f}ix\, e)$$

Note that, to preserve the well-formedness of the strictness annotations, the application must be strict, because e denotes a strict function.

This suggests the following encoding into CBV:

$$\mathcal{S}[\![\mathit{fix}\, e]\!] = \mathit{diverge}\, \mathcal{S}[\![e]\!],$$

where we can define the diverging function *diverge* as follows:

$$\mathit{diverge} = \mathit{letrec}\, \mathit{loop} = \lambda f.@\mathit{loop}\, f\, \mathit{in}\, \mathit{loop}.$$

5.2 Nonstrict Functions

We now treat the interesting case of fix, that is, the situation where the CBN fixpoint operator is applied to a nonstrict function. This operator obeys the usual unfolding property:

$$\mathit{fix}\, e \xrightarrow{u} \grave{@}e(\mathit{fix}\, e).$$

Note that the application must be nonstrict because e denotes a nonstrict function.

So we want an encoding that preserves the unfolding property:

$$\begin{array}{ccc} \mathit{fix}\, e & \xrightarrow{u} & \grave{@}e(\mathit{fix}\, e) \\ \mathcal{S}\downarrow & \downarrow\mathcal{S} & \downarrow \\ \mathcal{S}[\![\mathit{fix}\, e]\!] & \xrightarrow{u} & @\mathcal{S}[\![e]\!](\mathit{delay}\, \mathcal{S}[\![\mathit{fix}\, e]\!]). \end{array}$$

We adopt the following definition:

$$\mathcal{S}[\![\mathit{fix}\, e]\!] = @\mathit{rec}\mathcal{S}[\![e]\!],$$

where *rec* is a function that must verify

$$\forall e \in \mathit{Exp}, \qquad @\mathit{rec}\, e \xrightarrow{u} @e(\mathit{delay}\ (@\mathit{rec}\, e)),$$

to make the diagram above commute. Here is, for example, the definition of such a *rec*:

$$\mathit{rec} = \mathit{letrec}\, \mathit{loop} = \lambda f.@f(\mathit{delay}(@\mathit{loop}\, f))\ \mathit{in}\, \mathit{loop}.$$

5.3 Conclusion

We are now equipped to extend the CPS-transformation. We simply take the CBV CPS-transform of *diverge* and *rec*, noted as $\mathit{diverge}_c$ and rec_c, respectively, given the following transformation of *letrec* expressions:

$$\mathscr{C}_{\mathrm{v}}[\![\mathit{letrec}.f = \lambda x{:}\tau.e_0\ \mathit{in}\, e_1]\!] = \lambda\kappa.\mathit{letrec}\, f = \lambda x{:}\mathscr{C}_{\mathrm{v}}[\![\tau]\!].\mathscr{C}_{\mathrm{v}}[\![e_0]\!]\ \mathit{in}\ @\mathscr{C}_{\mathrm{v}}[\![e_1]\!]\kappa.$$

This makes $\mathscr{C}_{\mathrm{s}}$ preserve the unfolding properties of fix and fix.

Let us summarize the CPS-transformation of recursive bindings with strictness annotations:

$$\mathscr{C}_{\mathrm{s}}[\![\mathit{fix}\, e]\!] = \mathscr{C}_{\mathrm{v}}[\![@\mathit{diverge}\, \mathcal{S}[\![e]\!]]\!] = \lambda\kappa.@\mathscr{C}_{\mathrm{s}}[\![e]\!](\lambda v.@(@\mathit{diverge}_c\, v)\kappa),$$

$$\mathscr{C}_{\mathrm{s}}[\![\mathit{fix}\, e]\!] = \mathscr{C}_{\mathrm{v}}[\![@\mathit{rec}\, \mathcal{S}[\![e]\!]]\!] = \lambda\kappa.@\mathscr{C}_{\mathrm{s}}[\![e]\!](\lambda v.@(@\mathit{rec}_c\, v)\kappa).$$

$$\mathcal{C}_s^1 \;:\; AExp \rightarrow Exp$$

$$\mathcal{C}_s^1[\![\grave{x}]\!] = x$$
$$\mathcal{C}_s^1[\![e]\!] = \underline{\lambda}\, k\,.\, \overline{@}\, \mathcal{C}_s^2[\![e]\!]\, k$$

Fig. 11. Staged CPS-transformation for a language with strictness annotations (part 1).

6. A WELL-STAGED TRANSFORMATION OVER TERMS

In practice, the rewriting system of Figure 10 is only half of the CPS-transformation. The resulting term needs to be simplified to be of practical use. These simplifications are known as "administrative reductions" [18]. In an earlier work, Danvy and Filinski proposed a method for staging a CPS-transformation, by separating the administrative redexes from the syntax constructors [5]. Applying this method to the translation of Figure 10 over terms yields the translation of Figures 11–13. This translation yields terms without extraneous redexes, in one pass.[1] $\mathscr{C}_s^1[\![\cdot]\!]$ prevents suspensions from being suspended again. $\mathscr{C}_s^2[\![\cdot]\!]$ avoids extraneous η-redexes. $\mathscr{C}_s^3[\![\cdot]\!]$ is the main transformation function. The v's and a's are fresh variables.

The equations of Figures 11–13 can be read as a two-level specification à la Nielson and Nielson [14], and thus, they can be implemented directly in a functional language. Operationally, the overlined λ's and @'s correspond to functional abstractions and applications in the translation program, while only the underlined occurrences represent abstract-syntax constructors.

The result of transforming a term e into CPS in an empty context (represented by the identity function) is given by

$$\overline{@}\mathscr{C}_s^3[\![e]\!](\bar{\lambda} v.v).$$

With regard to complexity, the well-staged transformation combines the three linear-time transformations,

(1) suspension introduction $\mathscr{S}$,
(2) CBV CPS-transformation $\mathscr{C}_v$, and
(3) reduction of administrative redexes of the CPS-transformation,

into a single linear-time transformation. Alternatively, one can think of this transformation as combining the simultaneous execution of the linear CBN- and CBV-transformations with the reduction of administrative redexes.

---

[1] We leave it as an exercise for the reader to adapt Sabry and Felleisen's optimization over β-redexes [20]:

$$\mathscr{C}_s[\![\acute{@}(\acute{\lambda} x.e_1)e_0]\!] = \lambda\kappa.@\mathscr{C}_s[\![e_0]\!](\lambda x.@\mathscr{C}_s[\![e_1]\!]\kappa),$$
$$\mathscr{C}_s[\![\grave{@}(\grave{\lambda} x.e_1)e_0]\!] = \lambda\kappa.@(\lambda x.@\mathscr{C}_s[\![e_1]\!]\kappa)\mathscr{C}_s[\![e_0]\!].$$

Hint: Consider the translation of *let* expressions.

$$\mathcal{C}_s^2 \;:\; AExp \to \{k\} \to Exp$$

$$\begin{aligned}
\mathcal{C}_s^2[\![c]\!] &= \overline{\lambda}\,k\,.\,\underline{@}\,k\,c\\
\mathcal{C}_s^2[\![\acute{x}]\!] &= \overline{\lambda}\,k\,.\,\underline{@}\,k\,x\\
\mathcal{C}_s^2[\![\grave{x}]\!] &= \overline{\lambda}\,k\,.\,\underline{@}\,x\,k\\
\mathcal{C}_s^2[\![\acute{\lambda}\,x:\tau\,.\,e]\!] &= \overline{\lambda}\,k\,.\,\underline{@}\,k\,(\underline{\lambda}\,x:\mathcal{C}_s[\![\tau]\!]\,.\,\mathcal{C}_s^1[\![e]\!])\\
\mathcal{C}_s^2[\![\grave{\lambda}\,x:\tau\,.\,e]\!] &= \overline{\lambda}\,k\,.\,\underline{@}\,k\,(\underline{\lambda}\,x:(\mathcal{C}_s[\![\tau]\!]\to Ans)\to Ans\,.\,\mathcal{C}_s^1[\![e]\!])\\
\mathcal{C}_s^2[\![\acute{@}\,e_0\,e_1]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e_0]\!]\,(\overline{\lambda}\,v_0\,.\,\overline{@}\,\mathcal{C}_s^3[\![e_1]\!]\,(\overline{\lambda}\,v_1\,.\,\underline{@}\,(\underline{@}\,v_0\,v_1)\,k))\\
\mathcal{C}_s^2[\![\grave{@}\,e_0\,e_1]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e_0]\!]\,(\overline{\lambda}\,v_0\,.\,\underline{@}\,(\underline{@}\,v_0\,\mathcal{C}_s^1[\![e_1]\!])\,k)\\
\mathcal{C}_s^2[\![\acute{if}\;e_0\;then\;e_1\;else\;e_2]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e_0]\!]\,(\overline{\lambda}\,v_0\,.\,\underline{if}\;v_0\;\underline{then}\;\overline{@}\,\mathcal{C}_s^2[\![e_1]\!]\,k\;\underline{else}\;\overline{@}\,\mathcal{C}_s^2[\![e_2]\!]\,k)\\
\mathcal{C}_s^2[\![\acute{let}\;x=e_0\;in\;e_1]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e_0]\!]\,\overline{\lambda}\,v_0\,.\,\underline{let}\;x=v_0\;\underline{in}\;\overline{@}\,\mathcal{C}_s^2[\![e_1]\!]\,k\\
\mathcal{C}_s^2[\![\grave{let}\;x=e_0\;in\;e_1]\!] &= \overline{\lambda}\,k\,.\,\underline{let}\;x=\mathcal{C}_s^1[\![e_0]\!]\;\underline{in}\;\overline{@}\,\mathcal{C}_s^2[\![e_1]\!]\,k\\
\mathcal{C}_s^2[\![\acute{pair}\;e_0\;e_1]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e_0]\!]\,(\overline{\lambda}\,v_0\,.\,\overline{@}\,\mathcal{C}_s^3[\![e_1]\!]\,(\overline{\lambda}\,v_1\,.\,\underline{@}\,k\,(\underline{pair}\;v_0\;v_1)))\\
\mathcal{C}_s^2[\![\hat{pair}\;e_0\;e_1]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e_0]\!]\,(\overline{\lambda}\,v_0\,.\,\underline{@}\,k\,(\underline{pair}\;v_0\,\mathcal{C}_s^1[\![e_1]\!]))\\
\mathcal{C}_s^2[\![\check{pair}\;e_0\;e_1]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e_1]\!]\,(\overline{\lambda}\,v_1\,.\,\underline{@}\,k\,(\underline{pair}\;\mathcal{C}_s^1[\![e_0]\!]\,v_1))\\
\mathcal{C}_s^2[\![\grave{pair}\;e_0\;e_1]\!] &= \overline{\lambda}\,k\,.\,\underline{@}\,k\,(\underline{pair}\;\mathcal{C}_s^1[\![e_0]\!]\,\mathcal{C}_s^1[\![e_1]\!])\\
\mathcal{C}_s^2[\![\acute{fst}\;e]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e]\!]\,(\overline{\lambda}\,v\,.\,\underline{@}\,k\,(\underline{fst}\;v))\\
\mathcal{C}_s^2[\![\grave{fst}\;e]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e]\!]\,(\overline{\lambda}\,v\,.\,\underline{@}\,(\underline{fst}\;v)\,k)\\
\mathcal{C}_s^2[\![\acute{snd}\;e]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e]\!]\,(\overline{\lambda}\,v\,.\,\underline{@}\,k\,(\underline{snd}\;v))\\
\mathcal{C}_s^2[\![\grave{snd}\;e]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e]\!]\,(\overline{\lambda}\,v\,.\,\underline{@}\,(\underline{snd}\;v)\,k)\\
\mathcal{C}_s^2[\![\acute{fix}\;e]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e]\!]\,(\overline{\lambda}\,v\,.\,\underline{@}\,(\underline{@}\;diverge_c\;v)\,k)\\
\mathcal{C}_s^2[\![\hat{fix}\;e]\!] &= \overline{\lambda}\,k\,.\,\overline{@}\,\mathcal{C}_s^3[\![e]\!]\,(\overline{\lambda}\,v\,.\,\underline{@}\,(\underline{@}\;rec_c\;v)\,k)
\end{aligned}$$

Fig. 12. Staged CPS-transformation for a language with strictness annotations (part 2).

7. CONCLUSION AND ISSUES

Strictness analysis enables one to transform a program with CBN applications only into a program with both CBN and CBV applications. Suspensions allow one to transform the remaining CBN applications to CBV applications. We have composed the introduction of suspensions with the CBV CPS-transformation, thereby specifying how to transform a λ-term with strictness annotations into CPS directly. The resulting transformation generalizes both the CBN and CBV CPS-transformations, in that restricting it to nonstrict constructs gives the CBN CPS-transformation and restricting it to strict constructs gives the CBV CPS-transformation.

This new transformation enables one to implement a CBN language by using an existing CPS-based compiler [2, 22] or an existing program-transformation system [10, 12]. The method could be extended to call-by-need, for example, by using Okasaki et al.'s technique [16].

$$
\mathcal{C}_s^3 \;:\; AExp \rightarrow [Exp \rightarrow Exp] \rightarrow Exp
$$

$$
\begin{array}{rcl}
\mathcal{C}_s^3\llbracket c \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,c \\
\mathcal{C}_s^3\llbracket \acute{x} \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,x \\
\mathcal{C}_s^3\llbracket \grave{x} \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\underline{@}\,x\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a) \\
\mathcal{C}_s^3\llbracket \acute{\lambda}\,x:\tau\,.\,e \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,(\underline{\lambda}\,x:\mathcal{C}_\mathrm{s}\llbracket\tau\rrbracket\,.\,\mathcal{C}_s^1\llbracket e\rrbracket) \\
\mathcal{C}_s^3\llbracket \grave{\lambda}\,x:\tau\,.\,e \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,(\underline{\lambda}\,x:(\mathcal{C}_\mathrm{s}\llbracket\tau\rrbracket \rightarrow Ans) \rightarrow Ans\,.\,\mathcal{C}_s^1\llbracket e\rrbracket) \\
\mathcal{C}_s^3\llbracket \acute{@}\,e_0\,e_1 \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e_0\rrbracket\,(\overline{\lambda}\,v_0\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e_1\rrbracket\,(\overline{\lambda}\,v_1\,.\,\underline{@}\,(\underline{@}\,v_0\,v_1)\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a))) \\
\mathcal{C}_s^3\llbracket \grave{@}\,e_0\,e_1 \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e_0\rrbracket\,(\overline{\lambda}\,v_0\,.\,\underline{@}\,(\underline{@}\,v_0\,\mathcal{C}_s^1\llbracket e_1\rrbracket)\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a)) \\
\mathcal{C}_s^3\llbracket \acute{if}\ e_0\ then\ e_1\ else\ e_2 \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e_0\rrbracket\,(\overline{\lambda}\,v_0\,.\,\underline{let}\ k\ =\ \underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a \\
 & & \qquad \underline{in}\ \underline{if}\ v_0\ \underline{then}\ \overline{@}\,\mathcal{C}_s^2\llbracket e_1\rrbracket\,k\ \underline{else}\ \overline{@}\,\mathcal{C}_s^2\llbracket e_2\rrbracket\,k) \\
\mathcal{C}_s^3\llbracket \acute{let}\ x = e_0\ in\ e_1 \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e_0\rrbracket\,(\overline{\lambda}\,v_0\,.\,\underline{let}\ x = v_0\ \underline{in}\ \overline{@}\,\mathcal{C}_s^1\llbracket e_1\rrbracket\,\kappa) \\
\mathcal{C}_s^3\llbracket \grave{let}\ x = e_0\ in\ e_1 \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\underline{let}\ x = \mathcal{C}_s^1\llbracket e_0\rrbracket\ \underline{in}\ \overline{@}\,\mathcal{C}_s^1\llbracket e_1\rrbracket\,\kappa \\
\mathcal{C}_s^3\llbracket \acute{pair}\ e_0\ e_1 \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e_0\rrbracket\,(\overline{\lambda}\,v_0\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e_1\rrbracket\,(\overline{\lambda}\,v_1\,.\,\overline{@}\,\kappa\,(\underline{pair}\ v_0\,v_1))) \\
\mathcal{C}_s^3\llbracket \hat{pair}\ e_0\ e_1 \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e_0\rrbracket\,(\overline{\lambda}\,v_0\,.\,\overline{@}\,\kappa\,(\underline{pair}\ v_0\,\mathcal{C}_s^1\llbracket e_1\rrbracket)) \\
\mathcal{C}_s^3\llbracket \check{pair}\ e_0\ e_1 \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e_1\rrbracket\,(\overline{\lambda}\,v_1\,.\,\overline{@}\,\kappa\,(\underline{pair}\ \mathcal{C}_s^1\llbracket e_0\rrbracket\,v_1)) \\
\mathcal{C}_s^3\llbracket \grave{pair}\ e_0\ e_1 \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,(\underline{pair}\ \mathcal{C}_s^1\llbracket e_0\rrbracket\,\mathcal{C}_s^1\llbracket e_1\rrbracket) \\
\mathcal{C}_s^3\llbracket \acute{fst}\ e \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e\rrbracket\,(\overline{\lambda}\,v\,.\,\overline{@}\,\kappa\,(\underline{fst}\ v)) \\
\mathcal{C}_s^3\llbracket \grave{fst}\ e \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e\rrbracket\,(\overline{\lambda}\,v\,.\,\underline{@}\,(\underline{fst}\ v)\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a)) \\
\mathcal{C}_s^3\llbracket \acute{snd}\ e \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e\rrbracket\,(\overline{\lambda}\,v\,.\,\overline{@}\,\kappa\,(\underline{snd}\ v)) \\
\mathcal{C}_s^3\llbracket \grave{snd}\ e \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e\rrbracket\,(\overline{\lambda}\,v\,.\,\underline{@}\,(\underline{snd}\ v)\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a)) \\
\mathcal{C}_s^3\llbracket \acute{fix}\ e \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e\rrbracket\,(\overline{\lambda}\,v\,.\,\underline{@}\,(\underline{@}\ diverge_c\ v)\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a)) \\
\mathcal{C}_s^3\llbracket \grave{fix}\ e \rrbracket & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\llbracket e\rrbracket\,(\overline{\lambda}\,v\,.\,\underline{@}\,(\underline{@}\ rec_c\ v)\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a))
\end{array}
$$

Fig. 13. Staged CPS-transformation for a language with strictness annotations (part 3).

Also note that the encoding of suspensions is more natural in CPS than in direct style, since encoding a "parameterless" procedure as a direct-style λ-abstraction requires this λ-abstraction to take a dummy parameter. On the other hand, in CPS, a suspension is naturally written as an expression deprived of continuation, and thus, no dummy parameter is necessary.

This paper is motivated by our study of the direct-style transformation [4, 7]. Based on the extended CPS-transformation presented in this paper, we are currently deriving its inverse mapping: a transformer toward direct-style terms that are annotated with strictness information.

ACKNOWLEDGMENTS

We are grateful for the referee's suggestions. This paper also benefited from comments made by Andrzej Filinski, Julia Lawall, Karoline Malmkjær, and David Schmidt. The diagrams of Section 5 were drawn with Kristoffer Rose's XY-pic package.

REFERENCES

1. ACM. Proceedings of the 1992 ACM Conference on Lisp and Functional Programming. *LISP Pointers V*, 1 (June 1992).
2. APPEL, A. W. *Compiling with Continuations*. Cambridge University Press, New York, 1992.
3. BURN, G., AND LE MÉTAYER, D. L. Proving the correctness of compiler optimisations based on a global program analysis. Tech. Rep. Doc 92/20, Dept. of Computing, Imperial College of Science, Technology and Medicine, London, 1992.
4. DANVY, O. Back to direct style. In *Proceedings of the 4th European Symposium on Programming*, B. Krieg-Brückner, Ed. Lecture Notes in Computer Science, vol. 582. Springer-Verlag, New York, 1992, pp. 130–150. (Extended version to appear in *Sci. Comput. Program.*)
5. DANVY, O., AND FILINSKI, A. Representing control, a study of the CPS transformation. *Math. Structures Comput. Sci. 2*, 4, 361–391.
6. DANVY, O., AND HATCLIFF, J. Thunks (continued). In *Proceedings of the Workshop on Static Analysis WSA'92. Bigre J.* 81–82 (Sept. 1992), 3–11. (Extended version available as Tech. Rep. CIS-92-28, Kansas State Univ., Manhattan, 1992.)
7. DANVY, O., AND LAWALL, J. L. Back to direct style II: First-class continuations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming. LISP Pointers V*, 1 (June 1992), 299–310.
8. DANVY, O., AND TALCOTT, C. L., EDS. *Proceedings of the ACM SIGPLAN Workshop on Continuations*. Tech. Rep. STAN-CS-92-1426, Stanford Univ., Calif., June 1992.
9. FILINSKI, A. Recursion from iteration. In *Proceedings of the ACM SIGPLAN Workshop on Continuations*, O. Danvy and C. L. Talcott, Eds. Tech. Rep. STAN-CS-92-1426, Stanford Univ., Calif., June 1992, pp. 3–11.
10. FRADET, P., AND LE MÉTAYER, D. Compilation of functional languages by program transformation. *ACM Trans. Program. Lang. Syst. 13* (1991), 21–51.
11. INGERMAN, P. Z. Thunks, a way of compiling procedure statements with some comments on procedure declarations. *Commun. ACM 4*, 1 (Jan. 1961), 55–58.
12. KELSEY, R., AND HUDAK, P. Realistic compilation by program transformation. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages* (Austin, Tex., Jan.). ACM, New York, 1989, pp. 281–292.
13. MEYER, A. R., AND WAND, M. Continuation semantics in typed lambda-calculi (summary). In *Logics of Programs—Proceedings*, R. Parikh, Ed. Lecture Notes in Computer Science, vol. 193, Springer-Verlag, New York, 1985, pp. 219–224.
14. NIELSON, F., AND NIELSON, H. R. *Two-Level Functional Languages*. Cambridge Tracts in Theoretical Computer Science, vol. 34. Cambridge University Press, New York, 1992.
15. NIELSON, H. R., AND NIELSON, F. *Semantics with Applications, a Formal Introduction*. Wiley, New York, 1992.
16. OKASAKI, C., LEE, P., AND TARDITI, D. Graph reduction and lazy continuation-passing style. In *Proceedings of the ACM SIGPLAN Workshop on Continuations*, O. Danvy and C. L. Talcott, Eds. Tech. Rep. STAN-CS-92-1426, Stanford Univ., Calif., June 1992, pp. 91–101.
17. PEYTON JONES, S. L. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice-Hall, Englewood Cliffs, N.J., 1987.
18. PLOTKIN, G. D. Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci. 1* (1975), 125–159.
19. REYNOLDS, J. C. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference* (Boston, Mass.). ACM, New York, 1972, pp. 717–740.
20. SABRY, A., AND FELLEISEN, M. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming. LISP Pointers V*, 1 (June 1992), 288–298.
21. SCHMIDT, D. A. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Boston, Mass., 1986.
22. STEELE, G. L., JR. Rabbit: A compiler for Scheme. Tech. Rep. AI-TR-474, Artificial Intelligence Lab., Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

Received September 1992; revised November 1992; accepted January 1993