Back to Direct Style

Olivier Danvy
Department of Computing and Information Sciences
Kansas State University *
danvy@cis.ksu.edu

Abstract

While a great deal of attention has been devoted to transforming direct-style (DS) functional programs into continuation-passing style (CPS), to the best of our knowledge, the transformation of CPS programs into direct style has not been investigated. This paper describes the mapping of continuation-passing λ-terms to their applicative-order direct style counterpart. We set up foundations and outline applications of the direct style transformation.

We derive the direct style transformer from a non-standard denotational semantics of the untyped λ_v-calculus, that we prove congruent to the standard one.

Under precise conditions (linear occurrences of continuation parameters and no first-class use of continuations due to control operators such as `call/cc`), we show the DS and the CPS transformations to be inverse.

The direct style transformation can be used in partial evaluation, based on the fact that semantics-based program manipulation performs better when source programs are first transformed into CPS. As a result, specialized programs are expressed in CPS as well. The DS transformation maps them back to direct style.

Keywords

Direct style transformation, continuation-passing style transformation, λ_v-calculus, Scheme.

*Manhattan, Kansas 66506, USA. Part of this work was supported by NSF under grant CCR-9102625. Another part was carried out while visiting Xerox PARC in summer 1991.

1 Introduction

A considerable amount of work is dedicated to the continuation-passing style (CPS) transformation: van Wijngaarden transforms ALGOL 60 programs to eliminate labels [34]. Mazurkiewicz proves algorithms [24]. Fischer compares the generalities of implementations based on retention and deletion strategies [13]. Strachey and Wadsworth formalize control flow in the denotational semantics of programming languages with jumps [33]. Reynolds and Plotkin identify the evaluation order independence of CPS terms [30, 29]. Felleisen bases several syntactic theories of sequential control on continuations [10, 11]. Steele reveals CPS to be a well-suited intermediate representation for compiling strict functional languages [32] and Appel builds his implementation of SML on a preliminary transformation into CPS [1]. The CPS representation is at the core of Wand's combinator-based compilers [36] and is also used for compiling programs by program transformation [21, 14]. Together with first-class continuations as a programming paradigm [15], continuations are generally agreed to be an essential item in programming languages [16]. They also are ubiquitous in many areas of Computer Science: Moggi formalizes them using monads [25]. Filinski discovers values and continuations to be dual in a category theoretical sense [12]. Griffin and Murthy point out that the CPS transformation corresponds to the double negation translation in proof theory [17, 26].

Yet to the best of our knowledge, the inverse transformation, *i.e.*, transforming CPS terms into direct style (DS), has not been investigated despite its obvious value, *e.g.*, to disassemble CPS programs into something readable. Yet there is a need for the DS transformation, for example in the area of partial evaluation (*cf.* Section 5). Moreover, like the CPS transformation [8], the DS transformation can be derived formally.

1.1 Towards a direct style transformer

In a CPS term, continuations are produced by function composition and consumed by function application. In an empty context, the continuation is initialized with the identity function, *i.e.*, the identity element for function composition.

For example, if *dfac* and *cfac* respectively denote the DS and the CPS definitions of the factorial function, they are related by the traditional congruence relation

$$\forall n \in Nat, \forall k \in [Nat \to Ans],\ \ cfac(n,k) = k(dfac(n))$$

where *dfac*: $Nat \to Nat$ and *cfac*: $Nat \times [Nat \to Ans] \to Ans$ for some domain of final answers. An answer is the result of the expression that introduced the initial continuation and whose computation involved calling *cfac*.

$$cfac \stackrel{\text{rec}}{=} \lambda(n,k).\, n = 0 \to k(1),\ cfac(n-1, \lambda v.\, k(n \times v))$$

Intuitively, we want to transform a CPS term into a DS term by symbolically applying the continuation of an expression such as the recursive call

$$cfac(n-1, \lambda v.\, k(n \times v))$$

to this expression, letting the continuation of this expression be the identity function. For example, using this intuition on the factorial function yields the following definition.

$$fac \stackrel{\text{rec}}{=} \lambda(n,k).\, n = 0 \to k(1),\ (\lambda v.\, k(n \times v))\,(fac(n-1, \lambda v.\, v))$$

This definition can be simplified into

$$fac \stackrel{\text{rec}}{=} \lambda(n,k).\, n = 0 \to k(1),\ k(n \times (fac(n-1, \lambda v.\, v)))$$

If we carry out this transformation for all expressions, k will always denote the identity function and therefore we only need to apply the functions extending the continuation to all the expressions where continuations are extended (in practice: applications and conditional expressions), letting their continuation be identity. As a final step, we can get rid of all the continuation arguments (since they only denote identity). Going back to the example, this treatment yields the usual (direct style) definition of the factorial function.

$$dfac \stackrel{\text{rec}}{=} \lambda(n).\, n = 0 \rightarrow 1,\ n \times (dfac\,(n-1))$$

1.2 Overview

This paper is organized as follows. Section 2 reviews the CPS transformation and specifies the BNF of CPS terms. The derivation of the DS transformer for λ_v-terms (*i.e.*, call-by-value λ-terms) is described in Section 3. The CPS transformer and the DS transformer are inverses of each other. This point is addressed in Section 4. The need for going back to direct style is illustrated with semantics-based program manipulation in Section 5. Section 5 also illustrates the use of shifting back and forth between direct and continuation-passing styles. Section 6 situates our approach among related work. Finally Section 7 puts this investigation into perspective.

2 Transformation into Direct vs. into Continuation-Passing Style

The CPS transformation is generally perceived to be a complicated affair. Similarly, CPS terms (such as in a continuation semantics) are perceived as unfriendly to read and to understand, unless one develops a particular skill for it. CPS terms do not appear amenable to an inverse transformation into direct style,[1] otherwise this transformation would already be part of the debugging package of a compiler. These observations motivated us to investigate the direct style transformation. In particular, we wanted to derive it soundly rather than coming up with just another algorithm. Since the CPS transformation can be derived from a continuation semantics of the λ_v-calculus [2, 8, 35], we chose to derive the DS transformer from a direct semantics of the λ_v-calculus dedicated to CPS λ-terms.

2.1 The CPS transformation

Here is the abstract syntax for λ-terms (where the symbol @ denotes an application)

$e \in$ Exp	— domain of expressions
$l \in$ Lam	— domain of λ-abstractions
$op \in$ Opr	— domain of primitive operators
$c \in$ Cst	— domain of first-order constant values
$i \in$ Ide	— domain of identifiers

$$e ::= c \mid x \mid l \mid @\, e_0\,(e_1, ..., e_n) \mid e_0 \rightarrow e_1, e_2 \mid op\,(e_1, ..., e_m)$$
$$\mid \text{let}\,(i_1, ..., i_n) = (e_1, ..., e_n)\,\text{in}\,e_0 \mid \text{letrec}\,(i_1, ..., i_n) = (l_1, ..., l_n)\,\text{in}\,e_0$$
$$l ::= \lambda(i_1, ..., i_n).\, e$$

Let us present a one-pass CPS transformer for applicative order (call-by-value) λ-terms as we derived it in an earlier work [8]. This transformer is an optimized version of Fischer & Plotkin's CPS transformer [13, 29].

[1] For example, one could try to specify a relation between DS and CPS terms, express it in Prolog, and try to give Prolog a CPS term in the hope of producing the corresponding DS term. Unfortunately, our experience shows that this process is prone to looping.

$$
\begin{aligned}
[\![c]\!] &= \overline{\lambda}\kappa.\overline{@}\kappa\, c \\
[\![x]\!] &= \overline{\lambda}\kappa.\overline{@}\kappa\, x \\
[\![\lambda\,(x_1, ..., x_n)\,.\,e]\!] &= \overline{\lambda}\kappa.\overline{@}\kappa\,(\underline{\lambda}(x_1, ..., x_n, k).\overline{@}[\![e]\!]\,(\overline{\lambda}v.\underline{@}k\,v)) \\
[\![@\,e_0\,(e_1, ..., e_n)]\!] &= \overline{\lambda}\kappa.\,\overline{@}\,[\![e_0]\!] \\
&\qquad \overline{\lambda}v_0.\,\overline{@}\,[\![e_1]\!] \\
&\qquad\quad \overline{\lambda}v_1. ... \overline{@}\,[\![e_n]\!] \\
&\qquad\qquad \overline{\lambda}v_n.\underline{@}v_0\,(v_1, ..., v_n, \underline{\lambda}v.\overline{@}\kappa\,v) \\
[\![e_0 \rightarrow e_1, e_2]\!] &= \overline{\lambda}\kappa.\underline{\text{let}}\ k = \underline{\lambda}a.\overline{@}\kappa\,a \\
&\qquad \underline{\text{in}}\ \overline{@}[\![e_0]\!]\,(\overline{\lambda}p.p \underline{\rightarrow} \overline{@}[\![e_1]\!]\,(\overline{\lambda}m.\underline{@}k\,m)\,, \\
&\qquad\qquad \overline{@}[\![e_2]\!]\,(\overline{\lambda}n.\underline{@}k\,n)) \\
[\![op\,(e_1, ..., e_m)]\!] &= \overline{\lambda}\kappa.\,\overline{@}\,[\![e_1]\!] \\
&\qquad \overline{\lambda}v_1. ... \overline{@}\,[\![e_m]\!] \\
&\qquad\quad \overline{\lambda}v_m.\overline{@}\kappa\,(\underline{op}\,(v_1, ..., v_m)) \\
[\![\text{let}\,(i_1, ..., i_n) = (e_1, ..., e_n)\,\text{in}\,e_0]\!] &= \overline{\lambda}\kappa.\,\overline{@}\,[\![e_1]\!] \\
&\qquad \overline{\lambda}v_1. ... \overline{@}\,[\![e_n]\!] \\
&\qquad\quad \overline{\lambda}v_n.\overline{@}\kappa\,(\underline{\text{let}}\,(i_1, ..., i_n) = (v_1, ..., v_n) \\
&\qquad\qquad \underline{\text{in}}\ \overline{@}[\![e_0]\!]\,\kappa) \\
[\![\text{letrec}\,(i_1, ..., i_n) = (l_1, ..., l_n)\,\text{in}\,e_0]\!] &= \overline{\lambda}\kappa.\underline{\text{letrec}}\,(i_1, ..., i_n) = (\overline{@}[\![l_1]\!]\,(\overline{\lambda}f_1.f_1), \\
&\qquad\qquad\qquad\qquad ..., \\
&\qquad\qquad\qquad\qquad \overline{@}[\![l_n]\!]\,(\overline{\lambda}f_n.f_n)) \\
&\qquad \underline{\text{in}}\ \overline{@}[\![e_0]\!]\,\kappa
\end{aligned}
$$

where k is a fresh variable.

Figure 1: The CPS transformation

These equations can be read as a two-level specification *à la* Nielson and Nielson [28]. 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 any context (represented with a unary function: the continuation) is given by

$$\underline{\lambda}k.\overline{@}[\![e]\!]\,(\overline{\lambda}v.\underline{@}k\,v)$$

By construction, if the CPS transformer produces the term $\lambda\,k\,.\,e$, then the term $@\,(\lambda\,k\,.\,e)(\lambda\,v\,.\,v)$ has the same meaning as the original DS term.

2.2 Simplified CPS terms

In the transformation above as in the present work, we disregard the simplifications due to tail-calls, η-reductions, or identity let expressions. For example, the direct style conditional expression

$$\lambda\,(x)\,.\,x \rightarrow y,\ @\,f\,(z)$$

is transformed into

$$\lambda\,k\,.\,@\,k\,(\lambda\,(x, k')\,.\,\text{let}\,k'' = \lambda\,v\,.\,@\,k'\,v\,\text{in}\,x \rightarrow @\,k''\,y,\ @\,f\,(z, \lambda\,v\,.\,@\,k''\,v))$$

Of course, in practice, we reduce this to

$$\lambda\, k\,.\, @\, k\, (\lambda\, (x,\, k')\,.\, x \;\rightarrow\; @\, k'\, y,\; @\, f\, (z,\, k'))$$

instead,[2] but these simplifications actually complicate the development of this section — hence the purity. In practice, our CPS transformer produces such simplified terms [8] and our DS transformer handles them as well, using a refinement of the following BNF.[3]

2.3 Abstract syntax of CPS terms

Let us give the BNF of CPS terms as they are produced by the CPS transformer of last section. Since a fresh variable k is introduced for each abstraction and each conditional expression, let us index each non-terminal with this k as an inherited attribute. Much as Reynolds [30], we distinguish between "serious" and "trivial" expressions. Serious expressions e^k inherit a continuation k and trivial expressions t^k denote values that are passed to a continuation k.

A CPS term is of the form $\lambda\, k\,.\, e^k$, where e^k is defined by the following attribute grammar.

$e \in \text{Exp}$	— domain of expressions
$l \in \text{Lam}$	— domain of λ-abstractions
$op \in \text{Opr}$	— domain of primitive operators
$c \in \text{Cst}$	— domain of first-order constant values
$i, v, k \in \text{Ide}$	— domain of identifiers

$$
\begin{array}{lll}
e^k ::= & @\, k\, t^k & \\
 & \mid @\, t_0^k\, (t_1^k,\, ...,\, t_n^k,\, \lambda\, v\,.\, e^k) & \text{where } v \neq k \\
 & \mid \text{let } k' = \lambda\, v\,.\, e_1^k \text{ in } t^{k'} \;\rightarrow\; e_2^{k'},\; e_3^{k'} & \text{where } v \neq k \text{ and } k \notin FV(t^{k'} \;\rightarrow\; e_2^{k'},\; e_3^{k'}) \\
 & \mid \text{let } (i_1,\, ...,\, i_n) = (t_1^k,\, ...,\, t_n^k) \text{ in } e^k & \text{where } \forall j \in \{1, ..., n\}, i_j \neq k \\
 & \mid \text{letrec } (i_1,\, ...,\, i_n) = (l_1^k,\, ...,\, l_n^k) \text{ in } e^k & \text{where } \forall j \in \{1, ..., n\}, i_j \neq k \\
t^k ::= & c & \\
 & \mid i & \text{where } i \neq k \\
 & \mid l^k & \\
 & \mid op\, (t_1^k,\, ...,\, t_m^k) & \\
l^k ::= & \lambda\, (i_1,\, ...,\, i_n,\, k')\,.\, e^{k'} & \text{where } k \notin FV(\lambda\, (i_1,\, ...,\, i_n,\, k')\,.\, e^{k'})
\end{array}
$$

Each non-terminal is indexed with an identifier k denoting the current continuation. This attribute k allows us to restrict the BNF to CPS terms that correspond to purely functional terms. For example, a λ-term such as $\lambda\, (x,\, k)\,.\, @\, k'\, x$ is not produced by this BNF. We refer to this constraint as the *passing constraint*. The passing constraint is embodied in the first production $e^k ::= @\, k\, t^k$, in that *only the current continuation can be applied.*

2.4 Syntactic properties of CPS terms

The distinction between serious expressions e and trivial expressions t (and l) is captured in the following property. This property can be checked with the attribute grammar.

Proposition 1 $k \notin FV(t^k)$ □

[2] Sometimes, the form $\lambda\, xk'\,.\, x \;\rightarrow\; k'y,\; fzk'$ is preferred if the term occurs in an empty context, since the continuation corresponding to an empty context is the identity function — the point here is only the dropping of $\lambda\, k\,.\, @\, k$ ...

[3] For a simpler fix: the various η-redexes can be restored at syntax analysis time.

Corollary 1 $k \notin FV(l^k)$ □

The following syntactic property of CPS λ-terms is motivated by the passing constraint — at any stage in an evaluation, there is only one "current" continuation.

Proposition 2 *A CPS term resulting from the CPS transformation of a pure λ-term (i.e., a λ-term without* `call/cc`*) needs only one variable k to denote the current continuation.*

Proof: By structural induction over the attribute grammar above.

Here are the two productions where a new continuation is introduced.

$$e^k ::= \ldots \mid \text{let } k' = \lambda\, v\,.\, e_1^k \text{ in } t^{k'} \;\rightarrow\; e_2^{k'},\, e_3^{k'} \quad \textit{where } v \neq k \textit{ and } k \notin FV(t^{k'} \;\rightarrow\; e_2^{k'},\, e_3^{k'})$$

$$\mid \ldots$$

$$l^k ::= \lambda\,(i_1, \ldots, i_n, k')\,.\, e^{k'} \quad \textit{where } k \notin FV(\lambda\,(i_1, \ldots, i_n, k')\,.\, e^{k'})$$

These two productions come with the condition that k does not occur free in the body of the let expression and in the λ-abstraction. If we replace k' by k as a formal parameter of the let and of the λ, then all occurrences of k in the bodies will refer to this k, by virtue of lexical scope. *A fortiori*, they will not occur free in the let expression nor in the λ-abstraction. □

In practice, a fresh identifier k (*i.e.*, an identifier that does not occur free in the DS term to be CPS-transformed) is provided by the CPS transformer. We use this fresh identifier as a BNF attribute to ensure that only the current continuation is applied and also to distinguish the identifier representing the current continuation from the other identifiers. Its unicity (*cf.* Proposition 2) motivates the following revision.

2.5 Revised abstract syntax of CPS terms

Let us state the BNF of CPS λ-terms where continuations are denoted with a unique and fresh identifier k. The whole BNF is parameterized with this identifier k. Since the k-attribute is now unnecessary, we have stripped it off.

A CPS term is of the form $\lambda\, k\,.\, e$, where e is defined by the following grammar.

$e \in \text{Exp}_k$	— domain of expressions
$l \in \text{Lam}_k$	— domain of λ-abstractions
$op \in \text{Opr}$	— domain of primitive operators
$c \in \text{Cst}$	— domain of first-order constant values
$i, v \in \text{Ide}$	— domain of identifiers s.t. $k \notin \text{Ide}$
$k \in \{k\}$	— singleton domain of continuation identifiers

$$\begin{aligned}
e ::=\ & @\, k\, t \\
\mid\ & @\, t_0\,(t_1, \ldots, t_n, \lambda\, v\,.\, e) \\
\mid\ & \text{let } k = \lambda\, v\,.\, e_1 \text{ in } t \;\rightarrow\; e_2,\, e_3 \\
\mid\ & \text{let } (i_1, \ldots, i_n) = (t_1, \ldots, t_n) \text{ in } e \\
\mid\ & \text{letrec } (i_1, \ldots, i_n) = (l_1, \ldots, l_n) \text{ in } e \\
t ::=\ & c \\
\mid\ & i \\
\mid\ & l \\
\mid\ & op\,(t_1, \ldots, t_m) \\
l ::=\ & \lambda\,(i_1, \ldots, i_n, k)\,.\, e
\end{aligned}$$

2.6 A linearity property

CPS terms also satisfy the following linearity property.

Proposition 3 *In a CPS term, a continuation parameter occurs linearly.* □

This property can be proven by structural induction over the CPS transformation of Section 2.1. The linearity can be characterized with the following inference rules (where "⊗" stands for "exclusive or"). A continuation $\lambda v . e$ is linear in its parameter v when the following judgement

$$v \vdash e$$

is satisfied. The linearity property will be used last in the derivation of the DS transformer.

$$\frac{v \vdash t}{v \vdash @\, k\, t}$$

$$\frac{(\otimes_{i=0}^{n} v \vdash t_i) \otimes (v \neq w \ \wedge \ v \vdash e)}{v \vdash @\, t_0\, (t_1, ..., t_n, \lambda w . e)}$$

$$\frac{(v \neq w \ \wedge \ v \vdash e_1) \otimes (v \vdash t) \qquad v \notin FV(e_2) \qquad v \notin FV(e_3)}{v \vdash \text{let } k = \lambda w . e_1 \text{ in } t \rightarrow e_2, e_3}$$

$$\frac{(\otimes_{i=1}^{n} v \vdash t_i) \otimes (\forall j \in \{1, ..., n\}, i_j \neq v \ \wedge \ v \vdash e)}{v \vdash \text{let } (i_1, ..., i_n) = (t_1, ..., t_n) \text{ in } e}$$

$$\frac{\forall j \in \{1, ..., n\}, i_j \neq v \qquad \forall j \in \{1, ..., n\}, v \notin FV(l_j) \qquad v \vdash e}{v \vdash \text{letrec } (i_1, ..., i_n) = (l_1, ..., l_n) \text{ in } e}$$

$$v \vdash v$$

$$\frac{\otimes_{i=0}^{n} v \vdash t_i}{v \vdash op\, (t_1, ..., t_m)}$$

Figure 2: Linearity conditions over the continuation $\lambda v . e$

3 Derivation of the Direct Style Transformer

This section is organized as follows. First we define the denotational semantics of CPS terms from the usual denotational semantics of the λ_v-calculus (reproduced in Appendix A). We express it as a core semantics and a standard interpretation. We prove a property of this specification. Then we successively present non-standard interpretations (together with their congruence proofs) that are increasingly better suited to the derivation of a direct style transformer. Finally, we

view the last non-standard semantics as a mapping from syntax to syntax (since denotations are also expressed using λ-expressions). Improving its binding times yields the direct style transformer.

3.1 Core semantics and its standard interpretation

Based on the usual denotational semantics of λ_v-calculus (*cf.* Appendix A), let us derive the meaning of CPS terms in an empty context. The meaning of the term

$$[\![@\,(\lambda\, k\,.\,e)\,(\lambda\, v\,.\,v)]\!]$$

is given by the standard interpretation of e

$$\mathcal{E}_k[\![e]\!]\,\rho_{init}[k \mapsto Id]$$

where Id denotes the identity function and is the continuation representing an empty context.

This way we can give a denotation for each of the terms defined by the BNF. We stick to the distinction between serious and trivial terms by mapping them to their meaning using two valuation functions $\mathcal{E}_k$ and $\mathcal{T}_k$. These valuation functions can be derived from the usual valuation function $\mathcal{E}$ shown in Appendix A.

$$\begin{array}{rcl} \mathrm{Ide}_k & = & \mathrm{Ide} \cup \{k\} \\ Var_k & = & Var \cup \{k\} \\ Val & = & (Cst + Fun)_\perp \\ Env_k & = & Var_k \to Val \\ Fun & = & Val^+ \to Val \end{array}$$

$$\begin{array}{rcl} \mathcal{E}_k & : & \mathrm{Exp}_k \to Env_k \to Val \\ \mathcal{T}_k & : & \mathrm{Triv}_k \to Env_k \to Val \\ \mathcal{L}_k & : & \mathrm{Lam}_k \to Env_k \to Fun \\ \mathcal{C} & : & \mathrm{Cst} \to Val \\ \mathcal{I}_k & : & \mathrm{Ide}_k \to Var_k \\ \mathcal{O} & : & \mathrm{Opr} \to Val^* \to Val \end{array}$$

where $Id = \lambda(v)\,.\,v \in Fun$

For simplicity, and as in appendix A, we will identify the syntactic domain of identifiers Ide_k and the semantic domain of variables Var_k. Therefore, a syntactic identifier i is mapped to a semantic variable i.

There is only one point to be noted in the following equations. We have parameterized them with four combinators Send, App, Close, and Cond:

Send: $Val \to Fun \to Val$
App: $Fun \to Val^+ \to Val$
Close: $Var_k^+ \to [Env_k \to Val] \to Env_k \to Fun$
Cond: $[Env_k \to Val] \to [Env_k \to Val] \to [Env_k \to Val] \to Env_k \to \{k\} \to Fun \to Val$

This technique of defining a core semantics and a standard interpretation can be found in Nielson's work on data flow analysis by abstract interpretation [27] and in Jones and Mycroft's work on Minimal Function Graphs [18]. Today Jones and Nielson are developing this technique as a convenient format for specifying abstract interpretations [19].

$$\begin{array}{rcl} \mathcal{E}_k[\![@\,k\,t]\!]\,\rho & = & \textbf{let } v = \mathcal{T}_k[\![t]\!]\,\rho \textbf{ in } \mathrm{Send}\; v\;(\rho\, k) \\ \mathcal{E}_k[\![@\,t_0\,(t_1,\,...,\,t_n,\,\lambda\, v\,.\,e)]\!]\,\rho & = & \textbf{let } v_0 = \mathcal{T}_k[\![t_0]\!]\,\rho,\; v_1 = \mathcal{T}_k[\![t_1]\!]\,\rho,\; ...,\; v_n = \mathcal{T}_k[\![t_n]\!]\,\rho \\ & & \textbf{in } \mathrm{App}\; v_0\;(v_1,\,...,\,v_n,\,\lambda(a).\mathcal{E}_k[\![e]\!]\,\rho[v \mapsto a]) \\ \mathcal{E}_k[\![\textbf{let } k = \lambda\, v\,.\,e_1 \textbf{ in } t \to e_2,\, e_3]\!]\,\rho & = & \textbf{let } \kappa = \lambda(a).\mathcal{E}_k[\![e_1]\!]\,\rho[v \mapsto a] \\ & & \textbf{in } \mathrm{Cond}\;(\mathcal{T}_k[\![t]\!])\;(\mathcal{E}_k[\![e_2]\!])\;(\mathcal{E}_k[\![e_3]\!])\;\rho\; k\; \kappa \end{array}$$

$$
\begin{aligned}
\mathcal{E}_k[\![\text{let}\,(i_1,\,...,\,i_n) = (t_1,\,...,\,t_n)\,\text{in}\,e]\!]\,\rho &= \textbf{let}\; v_1 = \mathcal{T}_k[\![t_1]\!]\,\rho,\,...,\,v_n = \mathcal{T}_k[\![t_n]\!]\,\rho \\
&\quad \textbf{in}\; \mathcal{E}_k[\![e]\!]\,\rho[i_1 \mapsto v_1,\,...,\,i_n \mapsto v_n] \\
\mathcal{E}_k[\![\text{letrec}\,(i_1,\,...,\,i_n) = (l_1,\,...,\,l_n)\,\text{in}\,e]\!]\,\rho &= \textbf{letrec}\,(f_1,\,...,\,f_n) = (\mathcal{L}_k[\![l_1]\!]\,\rho[i_1 \mapsto f_1,\,...,\,i_n \mapsto f_n], \\
&\qquad ..., \\
&\qquad \mathcal{L}_k[\![l_n]\!]\,\rho[i_1 \mapsto f_1,\,...,\,i_n \mapsto f_n]) \\
&\quad \textbf{in}\; \mathcal{E}_k[\![e]\!]\,\rho[i_1 \mapsto f_1,\,...,\,i_n \mapsto f_n] \\
\mathcal{T}_k[\![c]\!]\,\rho &= \mathcal{C}[\![c]\!] \\
\mathcal{T}_k[\![i]\!]\,\rho &= \rho\, i \\
\mathcal{T}_k[\![l]\!]\,\rho &= \mathcal{L}_k[\![l]\!]\,\rho \\
\mathcal{T}_k[\![op\,(t_1,\,...,\,t_m)]\!]\,\rho &= \textbf{let}\; v_1 = \mathcal{T}_k[\![t_1]\!]\,\rho,\,...,\,v_m = \mathcal{T}_k[\![t_m]\!]\,\rho \\
&\quad \textbf{in}\; \mathcal{O}[\![op]\!]\,(v_1,\,...,\,v_m) \\
\mathcal{L}_k[\![\lambda\,(i_1,\,...,\,i_n,\,k)\,.\,e]\!]\,\rho &= \text{Close}\,(i_1,\,...,\,i_n,\,k)\,(\mathcal{E}_k[\![e]\!])\,\rho
\end{aligned}
$$

The following four combinators specify the standard interpretation of this core semantics.

$$\text{Send}: \mathit{Val} \rightarrow \mathit{Fun} \rightarrow \mathit{Val}$$
$$\text{Send}\; v\, \kappa \;=\; \kappa(v)$$

$$\text{App}: \mathit{Fun} \rightarrow \mathit{Val}^{+} \rightarrow \mathit{Val}$$
$$\text{App}\; f\; (v_1,\,...,\,v_n,\,\kappa) \;=\; f(v_1,\,...,\,v_n,\,\kappa)$$

$$\text{Close}: \text{Var}_k^{+} \rightarrow [\mathit{Env}_k \rightarrow \mathit{Val}] \rightarrow \mathit{Env}_k \rightarrow \mathit{Fun}$$
$$\text{Close}\,(i_1,\,...,\,i_n,\,k)\,\theta\,\rho \;=\; \lambda(v_1,\,...,\,v_n,\kappa).\theta\,\rho[i_1 \mapsto v_1,\,...,\,i_n \mapsto v_n,\,k \mapsto \kappa]$$

$$\text{Cond}: [\mathit{Env}_k \rightarrow \mathit{Val}] \rightarrow [\mathit{Env}_k \rightarrow \mathit{Val}] \rightarrow [\mathit{Env}_k \rightarrow \mathit{Val}] \rightarrow \mathit{Env}_k \rightarrow \{k\} \rightarrow \mathit{Fun} \rightarrow \mathit{Val}$$
$$\text{Cond}\,\theta\,\theta_2\,\theta_3\,\rho\,k\,\kappa \;=\; \theta\,\rho[k \mapsto \kappa] \rightarrow \theta_2\,\rho[k \mapsto \kappa],\,\theta_3\,\rho[k \mapsto \kappa]$$

Actually, this last combinator can be refined, as captured in the following property.

Proposition 4 *For all expressions t and legal environments $\rho[k \mapsto \kappa]$,*

$$\mathcal{T}_k[\![t]\!]\,\rho[k \mapsto \kappa] \;=\; \mathcal{T}_k[\![t]\!]\,\rho$$

Proof: By construction of any trivial term t, k does not occur free in t (*cf.* Proposition 1). Hence, evaluating t in an environment is insensitive as to whether this environment binds k or not. □

Therefore we can rewrite the definition of Cond as follows.

$$\text{Cond}: [\mathit{Env}_k \rightarrow \mathit{Val}] \rightarrow [\mathit{Env}_k \rightarrow \mathit{Val}] \rightarrow [\mathit{Env}_k \rightarrow \mathit{Val}] \rightarrow \mathit{Env}_k \rightarrow \{k\} \rightarrow \mathit{Fun} \rightarrow \mathit{Val}$$
$$\text{Cond}\,\theta\,\theta_2\,\theta_3\,\rho\,k\,\kappa \;=\; \theta\,\rho \rightarrow \theta_2\,\rho[k \mapsto \kappa],\,\theta_3\,\rho[k \mapsto \kappa]$$

3.2 Towards a direct style transformer (revisited)

Intuitively, if a procedure terminates, its continuation is guaranteed to be sent the "result" of this procedure (this passing constraint over continuations was captured in the k attribute). This intuition can be extended to arbitrary expressions: if evaluating an expression terminates, its continuation is guaranteed to be sent the corresponding value.

Suppose that, instead of the denotation of the current continuation, we pass the denotation of the identity procedure (*i.e.*, the identity function) when we call a procedure. This function

would then be applied to the "result" of this procedure and would return it. If we send this result to the current continuation, the computation would proceed as before.

This intuition can be extended to arbitrary expressions: if we evaluate a terminating expression in an environment where the continuation identifier is bound to the identity function, this function will be applied to an intermediate value and will return it. Should we send this value to the current continuation, the computation would continue as before.

Regarding non-termination, this intuition still holds: if a procedure does not terminate, its continuation will never be applied to any "result." Therefore substituting the identity function for its continuation does not change the (absence of) result of the whole computation. Ditto for expressions whose evaluation does not terminate — substituting the identity function for their continuation in the environment will make the evaluation diverge as well.

The following section formalizes these intuitions as properties of the denotational semantics above.

3.3 Semantic properties of CPS λ-terms

Definition 1 *A value $f \in Val$ is* well-behaved *if*

$$f = \lambda(v_1, ..., v_n, \kappa).\ \textbf{let}\ v = f\,(v_1, ..., v_n, Id)\ \textbf{in}\ \kappa(v)$$

whenever $f \in Fun$, $f \in Val^{n+1} \to Val$, and $n > 0$. (NB: as in Appendix A, the **let** *construct is strict.)*

Definition 2 *An environment $\rho \in Env$ is* well-behaved *if $\forall i \in Var$, $\rho\, i$ is well-behaved.*

Based on these two definitions, let us prove the three following properties.

Proposition 5 ($\mathcal{E}$-property) *For all expressions $e \in \mathrm{Exp}_k$ and well-behaved environments $\rho[k \mapsto \kappa]$ binding k to some $\kappa \in Val \to Val$,*

$$\mathcal{E}_k[\![e]\!]\,\rho[k \mapsto \kappa] \;=\; \textbf{let}\ v = \mathcal{E}_k[\![e]\!]\,\rho[k \mapsto Id]\ \textbf{in}\ \kappa(v)$$

Proposition 6 ($\mathcal{L}$-property) *For all expressions $\lambda\,(i_1, ..., i_n, k).\,e \in \mathrm{Lam}_k$ and well-behaved environments ρ,*

$$\mathcal{L}_k[\![\lambda\,(i_1, ..., i_n, k).\,e]\!]\,\rho \;=\; \lambda(v_1, ..., v_n, \kappa).\ \textbf{let}\ v = \mathcal{E}_k[\![e]\!]\,\rho[i_1 \mapsto v_1, ..., i_n \mapsto v_n, k \mapsto Id]\ \textbf{in}\ \kappa(v)$$

Proposition 7 ($\mathcal{T}$-property) *For all expressions $t \in \mathrm{Triv}_k$ and well-behaved environments ρ, $\mathcal{T}_k[\![t]\!]\,\rho$ is well-behaved.*

Proof: By mutual structural induction over the syntactic categories e, t, and l [31, Section 1.2]. In particular, the equality

$$\begin{aligned}&\mathcal{E}_k[\![\text{letrec}\,(i_1, ..., i_n) = (l_1, ..., l_n)\,\text{in}\,e]\!]\,\rho[k \mapsto \kappa]\\ =\ &\textbf{let}\ v = \mathcal{E}_k[\![\text{letrec}\,(i_1, ..., i_n) = (l_1, ..., l_n)\,\text{in}\,e]\!]\,\rho[k \mapsto Id]\\ &\textbf{in}\ \kappa(v)\end{aligned}$$

is proved by fixpoint induction. This requires proving the two following lemmas [31, Section 6.7].

Lemma 1 *The predicate "is well-behaved" is inclusive over the domain $Val^{n+1} \to Val$.*

Lemma 2 *$\bot_{Fun}$ is well-behaved.*

□

3.4 Non-standard interpretation and its congruence proof

The continuation gets extended with another function only for applications and for conditional expressions, whose meanings are defined by the combinators App and Cond. The properties above suggest the following non-standard combinators.

$$\begin{aligned}&\mathrm{App}'\colon \mathit{Fun} \to \mathit{Val}^+ \to \mathit{Val}\\ &\mathrm{App}'\, f\,(v_1, ..., v_n, \kappa) \;=\; \textbf{let } v = f(v_1, ..., v_n, \mathit{Id}) \textbf{ in } \kappa(v)\end{aligned}$$

$$\begin{aligned}&\mathrm{Cond}'\colon [\mathit{Env}_k \to \mathit{Val}] \to [\mathit{Env}_k \to \mathit{Val}] \to [\mathit{Env}_k \to \mathit{Val}] \to \mathit{Env}_k \to \{k\} \to \mathit{Fun} \to \mathit{Val}\\ &\mathrm{Cond}'\,\theta\,\theta_2\,\theta_3\,\rho\,k\,\kappa \;=\; \theta\,\rho \to \textbf{let } v_2 = \theta_2\,\rho[k \mapsto \mathit{Id}] \textbf{ in } \kappa(v_2),\\ &\qquad\qquad\qquad\qquad\quad \textbf{let } v_3 = \theta_3\,\rho[k \mapsto \mathit{Id}] \textbf{ in } \kappa(v_3)\end{aligned}$$

Together with Send′ and Close′ (which do not change)

$$\begin{aligned}&\mathrm{Send}' = \mathrm{Send}\\ &\mathrm{Close}' = \mathrm{Close}\end{aligned}$$

App′ and Cond′ define a non-standard interpretation of the core semantics.

Proposition 8 *The standard and the non-standard interpretations define the same language.*

Proof: by structural induction, using the $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{T}$-properties. Here are the only interesting cases.
Let f be well-behaved.

$$\begin{aligned}&\mathrm{App}\, f\,(v_1, ..., v_n, \kappa)\\ &= f(v_1, ..., v_n, \kappa) && \text{— definition of App}\\ &= (\lambda(v_1, ..., v_n, \kappa).\, \textbf{let } v = f(v_1, ..., v_n, \mathit{Id}) \textbf{ in } \kappa(v))(v_1, ..., v_n, \kappa) && \text{— } f \text{ is well-behaved}\\ &= \textbf{let } v = f(v_1, ..., v_n, \mathit{Id}) \textbf{ in } \kappa(v) && \text{— } \beta\text{-reduction}\\ &= \mathrm{App}'\, f\,(v_1, ..., v_n, \kappa) && \text{— definition of App}'\end{aligned}$$

$$\begin{aligned}&\mathrm{Cond}\,(\mathcal{T}_k[\![t]\!])\,(\mathcal{E}_k[\![e_2]\!])\,(\mathcal{E}_k[\![e_3]\!])\,\rho\,k\,\kappa\\ &= \mathcal{T}_k[\![t]\!]\,\rho \to \mathcal{E}_k[\![e_2]\!]\,\rho[k \mapsto \kappa],\\ &\qquad\qquad \mathcal{E}_k[\![e_3]\!]\,\rho[k \mapsto \kappa] && \text{— definition of Cond}\\ &= \mathcal{T}_k[\![t]\!]\,\rho \to \textbf{let } v_2 = \mathcal{E}_k[\![e_2]\!]\,\rho[k \mapsto \mathit{Id}] \textbf{ in } \kappa(v_2),\\ &\qquad\qquad \textbf{let } v_3 = \mathcal{E}_k[\![e_3]\!]\,\rho[k \mapsto \mathit{Id}] \textbf{ in } \kappa(v_3) && \text{— } \mathcal{E}\text{-property}\\ &= \mathrm{Cond}'\,(\mathcal{T}_k[\![t]\!])\,(\mathcal{E}_k[\![e_2]\!])\,(\mathcal{E}_k[\![e_3]\!])\,\rho\,k\,\kappa && \text{— definition of Cond}'\end{aligned}$$

□

Under the present interpretation, and intuitively, k denotes Id at every point. The following section captures this intuition in another interpretation of the core semantics that we prove congruent to the present one.

3.5 One step further

Let $\mathit{Env}'_k \rightleftharpoons \langle \{\rho \in \mathit{Env}_k \mid \rho\, k = \mathit{Id}\}, \sqsubseteq_{\mathit{Env}_k}\rangle$. Env'_k is a cpo.
Let us define three new valuation functions.

$$\begin{aligned}&\mathcal{E}'_k : \mathrm{Exp}_k \to \mathit{Env}'_k \to \mathit{Val}\\ &\mathcal{T}'_k : \mathrm{Triv}_k \to \mathit{Env}'_k \to \mathit{Val}\\ &\mathcal{L}'_k : \mathrm{Lam}_k \to \mathit{Env}'_k \to \mathit{Fun}\end{aligned}$$

as a $\mathcal{E}_k$, $\mathcal{T}_k$, and $\mathcal{L}_k$ interpretation with Env'_k domain and App′ and Cond′ as combinators.

Proposition 9 *$\mathcal{E}'_k$, $\mathcal{T}'_k$, and $\mathcal{L}'_k$ are well defined and*

$$\begin{cases} \mathcal{E}'_k[\![e]\!]\rho &= \mathcal{E}_k[\![e]\!]\rho & \text{whenever } e \in \mathrm{Exp}_k \\ \mathcal{T}'_k[\![t]\!]\rho &= \mathcal{T}_k[\![t]\!]\rho & \text{whenever } t \in \mathrm{Triv}_k \\ \mathcal{L}'_k[\![l]\!]\rho &= \mathcal{L}_k[\![l]\!]\rho & \text{whenever } l \in \mathrm{Lam}_k \end{cases}$$

and whenever $\rho \in Env'_k$.

Proof: By structural induction on the syntax of Exp_k, Triv_k, and Lam_k. We must verify that all environments built within the right-hand sides of semantic equations are in Env'_k.

Here is the main step. There are only two cases where a new k is introduced: in λ-abstractions and in conditional expressions. The $\mathcal{L}$-property tells us that the environment is extended with the continuation parameter denoting *Id*. In the case of conditional expressions, Cond′ tells us that the consequent and alternative expressions are evaluated in an environment where the new continuation parameter denotes *Id*.

Equality immediately follows from well definedness since $\mathcal{E}'_k$ uses the same semantic equations as $\mathcal{E}_k$.

Finally let us notice that the meaning of the term

$$[\![@\,(\lambda\, k\,.\, e)\,(\lambda\, v\,.\, v)]\!]$$

is given by the standard interpretation of e

$$\mathcal{E}_k[\![e]\!]\rho_{init}[k \mapsto Id]$$

where k denotes *Id*. This establishes the "initial environment" for an expression. □

Let us use Env'_k from now on. This suggests going one step further with a new interpretation where the identity function is *not* passed at call sites but is introduced at definition sites instead. This intuition is captured in the following domain and combinators.

$$Fun'' = Val^* \to Val$$

$$\begin{aligned} &\mathrm{App}'' \colon Fun'' \to Val^+ \to Val \\ &\mathrm{App}''\, f\,(v_1, \ldots, v_n, \kappa) \;=\; \textbf{let } v = f(v_1, \ldots, v_n) \textbf{ in } \kappa(v) \end{aligned}$$

$$\begin{aligned} &\mathrm{Close}'' \colon Var^+ \to [Env'_k \to Val] \to Env'_k \to Fun'' \\ &\mathrm{Close}''\,(i_1, \ldots, i_n, k)\, b\, \rho \;=\; \lambda(v_1, \ldots, v_n)\,.\, b\, \rho[i_1 \mapsto v_1, \ldots, i_n \mapsto v_n, k \mapsto Id] \end{aligned}$$

$$\begin{aligned} &\mathrm{Send}'' = \mathrm{Send}' \\ &\mathrm{Cond}'' = \mathrm{Cond}' \end{aligned}$$

Proposition 10 *$\mathcal{E}'_k$, $\mathcal{T}'_k$, and $\mathcal{L}'_k$ with* Fun″, App″, *and* Close″ *are well-defined and equal $\mathcal{E}'_k$, $\mathcal{T}'_k$, and $\mathcal{L}'_k$ with* Fun, App′, *and* Close′.

Proof: By structural induction on the syntax of Exp_k, Triv_k, and Lam_k. Here is the essential step.

$\mathrm{App}'\,(\mathrm{Close}'\,(i_1, ..., i_n, k)\,b\,\rho)\,(v_1, ..., v_n, \kappa)$
$= \textbf{let}\ v = (\lambda(v_1, ..., v_n, \kappa).\,b\,\rho[i_1 \mapsto v_1, ..., i_n \mapsto v_n, k \mapsto \kappa])(v_1, ..., v_n, Id)$
$\quad \textbf{in}\ \kappa\,(v)$ — *definition of* App′ *and* Close′
$= \textbf{let}\ v = b\,\rho[i_1 \mapsto v_1, ..., i_n \mapsto v_n, k \mapsto Id]$
$\quad \textbf{in}\ \kappa\,(v)$ — *β-reduction*
$= \textbf{let}\ v = (\lambda(v_1, ..., v_n).\,b\,\rho[i_1 \mapsto v_1, ..., i_n \mapsto v_n, k \mapsto Id])(v_1, ..., v_n)$
$\quad \textbf{in}\ \kappa\,(v)$ — *abstraction*
$= \mathrm{App}''\,(\mathrm{Close}''\,(i_1, ..., i_n, k)\,b\,\rho)\,(v_1, ..., v_n)$ — *definition of* App″ *and* Close″

□

Let us go back to the denotation of $[\![@\,k\,t]\!]$.

$$\mathcal{E}'_k[\![@\,k\,t]\!]\,\rho \;=\; \textbf{let}\ v = \mathcal{T}'_k[\![t]\!]\,\rho\ \textbf{in}\ \mathrm{Send}'\,v\,(\rho\,k)$$

By definition of Env'_k, the second argument of Send′ is always *Id*. Let us capture this property in a new definition of this combinator:

$$\begin{aligned}&\mathrm{Send}'' \colon Val \to Fun \to Val\\ &\mathrm{Send}''\,v\,(\rho\,k) = (\rho\,k)(v)\\ &\qquad\qquad = Id(v)\\ &\qquad\qquad = v\end{aligned}$$

Proposition 11 *$\mathcal{E}'_k$ with* Send″ *is well-defined and equals $\mathcal{E}'_k$ with* Send′. □

This leads us to the following equivalent denotation of $[\![@\,k\,t]\!]$

$$\begin{aligned}\mathcal{E}'_k[\![@\,k\,t]\!]\,\rho &= \textbf{let}\ v = \mathcal{T}'_k[\![t]\!]\,\rho\ \textbf{in}\ \mathrm{Send}''\,v\,(\rho\,k)\\ &= \textbf{let}\ v = \mathcal{T}'_k[\![t]\!]\,\rho\ \textbf{in}\ v\\ &= \mathcal{T}'_k[\![t]\!]\,\rho\end{aligned}$$

The syntactic continuation k is only looked up in the environment in the denotation of $[\![@\,k\,t]\!]$, which, we just saw, can be simplified into an expression where k is *not* looked up in the environment. Therefore at this point, k is completely useless.

Now we are equipped enough to actually derive the direct style transformer.

3.6 From the non-standard denotational specification to a definitional interpreter, and from the interpreter to a compiler based on binding time analysis

It is possible to take the denotational specification of the last section literally as a functional program, following Reynolds's definitional interpreter insight [30]; and then to analyze its binding times, as customary in partial evaluation, to compile and to generate the corresponding compiler [20, 5].

Although straightforward, the derivation is a bit lengthy, so for conciseness, let us instead consider the denotational specification above as a rewriting system from syntax to semantics. Still with an eye on binding times, we will alter this rewrite system in a meaning-preserving way, yielding the final direct style transformer. This is done in the following section.

3.7 Viewing the denotational specification as a rewriting system

Let us take the homomorphic metaphor of denotational semantics (*i.e.*, from syntax to semantics) literally, based on the fact that both the syntax and the semantics are expressed as λ-terms. Taking the equations above as specifying a syntactic rewrite system and letting variables denote themselves (which makes the environment useless) leads to the direct style transformer. We decide that our target language is to be the λ_v-calculus, so there is no need for the strict **let** expressions anymore; we unfold them.

The term $@\,(\lambda\, k\,.\, e)(\lambda\, v\,.\, v)$ is rewritten as $[\![e]\!]$, where the rewrite function $[\![\ ]\!]$ is defined inductively as follows (for concision, we have unfolded the Send, App, Close, and Cond combinators of the last non-standard interpretation).

$$
\begin{aligned}
[\![@\, k\, t]\!] &= [\![t]\!] \\
[\![@\, t_0\, (t_1, \ldots, t_n, \lambda\, v\,.\, e)]\!] &= @\,(\lambda v.[\![e]\!])\,(@\,[\![t_0]\!]\,([\![t_1]\!], \ldots, [\![t_n]\!])) \\
[\![\text{let } k = \lambda\, v\,.\, e_1 \text{ in } t \rightarrow e_2, e_3]\!] &= @\,(\lambda v.[\![e_1]\!])\,([\![t]\!] \rightarrow [\![e_2]\!], [\![e_3]\!]) \\
[\![\text{let } (i_1, \ldots, i_n) = (t_1, \ldots, t_n) \text{ in } e]\!] &= \text{let } (i_1, \ldots, i_n) = ([\![t_1]\!], \ldots, [\![t_n]\!]) \text{ in } [\![e]\!] \\
[\![\text{letrec } (i_1, \ldots, i_n) = (l_1, \ldots, l_n) \text{ in } e]\!] &= \text{letrec } (i_1, \ldots, i_n) = ([\![l_1]\!], \ldots, , [\![l_n]\!]) \text{ in } [\![e]\!] \\
[\![c]\!] &= c \\
[\![i]\!] &= i \\
[\![op\,(t_1, \ldots, t_m)]\!] &= op([\![t_1]\!], \ldots, [\![t_m]\!]) \\
[\![\lambda\,(i_1, \ldots, i_n, k)\,.\, e]\!] &= \lambda\,(i_1, \ldots, i_n)\,.\,[\![e]\!]
\end{aligned}
$$

Figure 3: Syntax-directed transformation into direct style

Alternatively, this rewriting can be seen as going from an environment model to a Church-style encoding of binding relations. Wand has proven that these two encodings are equivalent [37].

3.8 The actual direct style transformer

The rewriting system of Section 3.7 can be subjected to an important binding time improvement based on the linearity property of continuation parameters. Since the body of a continuation is linear in its argument (*cf.* Proposition 3), and due to the call-by-value nature of our setting, the outer redex produced by the translation of applications can actually be reduced *at translation time*. Based on this improvement, it is possible to map a term such as

$$\lambda k.k(\lambda x k.g x(\lambda v.f v(\lambda a.k a)))$$

into

$$\lambda x.f(g x)$$

instead of mapping it to

$$\lambda x.((\lambda v.(\lambda a.a)(f v))(g x))$$

only. By the same token, the β-redex produced by the translation of conditional expressions should be reduced at translation time as well.

Using the same two-level notation as in Section 2, let us reexpress the DS transformation, distinguishing between translation time and run time constructs.

$$
\begin{aligned}
[\![@\, k\, t]\!] &= [\![t]\!] \\
[\![@\, t_0\, (t_1, ..., t_n, \lambda\, v\, .\, e)]\!] &= \overline{@}(\overline{\lambda} v.[\![e]\!])\,(\underline{@}[\![t_0]\!]\,([\![t_1]\!], ..., [\![t_n]\!])) \\
[\![\text{let } k = \lambda\, v\, .\, e_1 \text{ in } t \rightarrow e_2, e_3]\!] &= \overline{@}(\overline{\lambda} v.[\![e_1]\!])\,([\![t]\!] \underline{\rightarrow} [\![e_2]\!], [\![e_3]\!]) \\
[\![\text{let } (i_1, ..., i_n) = (t_1, ..., t_n) \text{ in } e]\!] &= \underline{\text{let}}\ (i_1, ..., i_n) = ([\![t_1]\!], ..., [\![t_n]\!])\ \underline{\text{in}}\ [\![e]\!] \\
[\![\text{letrec } (i_1, ..., i_n) = (l_1, ..., l_n) \text{ in } e]\!] &= \underline{\text{letrec}}\ (i_1, ..., i_n) = ([\![l_1]\!], ..., , [\![l_n]\!])\ \underline{\text{in}}\ [\![e]\!] \\
[\![c]\!] &= c \\
[\![i]\!] &= i \\
[\![op\, (t_1, ..., t_m)]\!] &= \underline{op}\, ([\![t_1]\!], ..., [\![t_m]\!]) \\
[\![\lambda\, (i_1, ..., i_n, k)\, .\, e]\!] &= \underline{\lambda}(i_1, ..., i_n).[\![e]\!]
\end{aligned}
$$

Figure 4: The DS transformation

4 Are the CPS and DS Transformations Inverse?

Proposition 12 *The DS and the CPS transformations are inverses of each other, up to α-conversion.*

Proof: This proof is not immediate because of the translation-time simplifications (specified by the overlined @ and λ in Sections 1 and 4). Therefore we cannot line up producers and consumers together and simplify the composition of these two transformations. Instead, let us stage these two transformations.

Let $\mathcal{D}$ denote the DS transformation. We stage $\mathcal{D}$ as follows.

$$\mathcal{D} = \mathcal{D}_2 \circ \mathcal{D}_1$$

$\mathcal{D}_1$ is specified in Section 3.7. It maps a CPS term into a non-simplified DS term (*i.e.*, a λ-term with $\overline{@}$ and $\overline{\lambda}$). $\mathcal{D}_2$ carries the β-reductions involving $\overline{@}$ and $\overline{\lambda}$.

Correspondingly, let $\mathcal{C}$ denote the CPS transformation. We stage $\mathcal{C}$ as follows.

$$\mathcal{C} = \mathcal{C}_2 \circ \mathcal{C}_1$$

$\mathcal{C}_1$ maps a DS term into a term specified by the following BNF.

$$
\begin{aligned}
e ::=&\ \ s \mid t \\
s ::=&\ \ \overline{@}(\overline{\lambda} v.e)\,(@\, t_0\, (t_1, ..., t_n)) \\
&\mid \overline{@}(\overline{\lambda} v.e_1)\,(t \rightarrow e_2, e_3) \\
&\mid \text{let } (i_1, ..., i_n) = (t_1, ..., t_n) \text{ in } e \\
&\mid \text{letrec } (i_1, ..., i_n) = (t_1, ..., t_n) \text{ in } e \\
t ::=&\ \ c \\
&\mid i \\
&\mid op\, (t_1, ..., t_m) \\
&\mid \lambda\, (i_1, ..., i_n)\, .\, e
\end{aligned}
$$

Intuitively, $\mathcal{C}_1$ transforms a λ-term into a head form by introducing a bunch of β-redexes and then sequentializing them [6]. $\mathcal{C}_1$ and $\mathcal{D}_2$ are inverses of each other.

Using a unique identifier k, $\mathcal{C}_2$ introduces continuations. It is the inverse of $\mathcal{D}_1$. □

5 Semantics-Based Program Manipulation

CPS matters when one manipulates programs based on their semantics because it makes flow analyses yield more precise results [27]. As a program specialization technique, partial evaluation benefits from pre-transforming source programs into CPS [4]. Since residual programs are expressed in CPS, they are good candidates for the DS transformation, if partial evaluation is to be seen as a source-to-source transformation.

6 Comparison with Related Work

Properly speaking, there are no related works since (again to the author's best knowledge) transforming continuation-passing terms into direct style has not been explored so far. There appears to be two main classes of applications for the CPS transformation: for program analysis and transformation, and for functional reasoning about control operators. Our DS transformation covers the first class but not the second. It can be extended by relaxing the passing constraint over CPS terms, as investigated in "Back to Direct Style II" [9]. Further, relaxing the linearity property on continuation parameters is handled by inserting a let or a sequence expression. Finally, relaxing the CPS texture amounts to introducing control operators such as `shift` and `reset` [7].

7 Conclusions and Issues

Work in semantics-based program manipulation revealed the need for a transformation into direct style. We have shown that such transformations exist and we have derived one for call-by-value soundly. In the area of partial evaluation, we have applied the CPS transformation to source programs and the DS transformation to specialized programs, obtaining substantial improvements.

A number of issues remain to be explored. Here are a few of them.

- Currently the DS transformer assumes continuations to occur as the last parameter. However nothing in a general-purpose partial evaluator ensures that residual continuations occur last. How could the DS transformer cope with continuations occurring anywhere?
- Can the DS transformer be extended to produce DS terms including control operators such as `call/cc`? (NB: in collaboration with Lawall, we have extended the DS transformer to handle first-class continuations [9].)
- Does there exist a DS transformer towards λ_n-terms? (NB: we have derived one at this time.)
- The DS and the CPS transformations are too strong in that they are global. Often we know that parts of our programs are "trivial" in Reynolds's sense [30] and therefore they do not need to be transformed. Can we minimize the extent of the DS and CPS transformations? We understand that Wadler's use of monads corresponds to this, together with instrumenting the continuation to receive not only a value but also a single-threaded resource, *e.g.*, for monitoring [35, 22].
- CPS programs are single-threaded in their continuation and therefore their control is inherently sequential. Could the DS transformer be used as a tool for parallelization? We are thinking of a programming style where DS sub-terms would be evaluated in parallel and CPS terms would be evaluated sequentially.

- The CPS transformation corresponds to other transformations in constructive mathematics [17, 26]. Can the DS transformation have a similar equivalent?

- Finally the DS transformer can contribute to derive program analyzers for CPS code that are at least as good as program analyzers for DS code. Here is the idea.

  Let $\mathcal{C}$ and $\mathcal{D}$ be inverse CPS and DS transformers, respectively; and let $\mathcal{A}_d$ and $\mathcal{A}_c$ be program analyzers for DS and for CPS programs, respectively, such that

  $$\mathcal{A}_c \circ \mathcal{C} \sqsupseteq \mathcal{A}_d$$

  In other terms, analyzing a DS program should yield a result which is at least as good as analyzing the CPS counterpart of this program (*cf.* Section 5). We can isolate $\mathcal{A}_c$ by composing $\mathcal{C}$ on the right

  $$\mathcal{A}_c \circ \mathcal{C} \circ \mathcal{D} \sqsupseteq \mathcal{A}_d \circ \mathcal{C}$$

  and by simplifying (composition is associative, and $\mathcal{C}$ and $\mathcal{D}$ are inverses of each other)

  $$\mathcal{A}_c \sqsupseteq \mathcal{A}_d \circ \mathcal{C}$$

  There are two ways to read this equation.

  1. Trivial way: "To analyze a CPS program, first map it back to DS and then analyze it by conventional means. The result is guaranteed not to get worse."
  2. Insightful way: "To derive an analyzer of CPS terms, symbolically compose (and simplify!) an analyzer of DS terms and the DS transformer."

  The latter way offers a practical insight to build program analyzers for CPS programs that are at least as good as existing program analyzers for DS programs. Such a class of new program analyzers appears to be needed in modern compilers for strict functional languages (Scheme, ML). Tarditi is working on this class of new program analyzers at Carnegie-Mellon University.

Acknowledgements

This work benefited from Karoline Malmkjær's patient and sharp-witted comments and from David Schmidt's interest and rigor. Thanks are also due to Andrzej Filinski, Charles Consel, Jim des Rivières, Peter Sestoft, and Julia Lawall.

A Denotational Semantics of the λ_v-Calculus

This appendix addresses the λ_v-calculus applied to the usual first-order constants (boolean, numbers, *etc.*) and extended with conditional expressions, recursive definitions, and primitive operations. Primitive operators either map first-order arguments to first-order results or are data structure constructors and destructors such as in list operations.

Abstract Syntax

$e \in$ Exp	— domain of expressions
$l \in$ Lam	— domain of λ-abstractions
$op \in$ Opr	— domain of primitive operators
$c \in$ Cst	— domain of first-order constant values

$i \in \text{Ide}$ — domain of identifiers

$$e ::= c \mid i \mid l \mid @\, e_0\,(e_1, ..., e_n) \mid op\,(e_1, ..., e_m) \mid$$
$$e_1 \rightarrow e_2, e_3 \mid \text{let}\,(i_1, ..., i_n) = (e_1, ..., e_n)\,\text{in}\,e_0 \mid \text{letrec}\,(i_1, ..., i_n) = (l_1, ..., l_n)\,\text{in}\,e_0$$
$$l ::= \lambda\,(i_1, ..., i_n)\,.\,e$$

NB: The symbol @ denotes an application.

Semantic Domains

$$\begin{array}{rcl} \mathit{Val} & = & (\mathit{Cst} + \mathit{Fun})_\perp \\ \mathit{Env} & = & \mathit{Var} \rightarrow \mathit{Val} \\ \mathit{Fun} & = & \mathit{Val}^* \rightarrow \mathit{Val} \end{array} \qquad \begin{array}{rcl} \mathcal{E} & : & \text{Exp} \rightarrow \mathit{Env} \rightarrow \mathit{Val} \\ \mathcal{L} & : & \text{Lam} \rightarrow \mathit{Env} \rightarrow \mathit{Fun} \\ \mathcal{C} & : & \text{Cst} \rightarrow \mathit{Val} \\ \mathcal{I} & : & \text{Ide} \rightarrow \mathit{Var} \\ \mathcal{O} & : & \text{Opr} \rightarrow \mathit{Fun} \end{array}$$

For simplicity, we identify the syntactic domain of identifiers Ide and the semantic domain of variables *Var*. Literally speaking, a syntactic identifier i is mapped into a semantic variable $\mathcal{I}[\![i]\!]$, but we will refer to this variable as i. Identifying identifiers and variables allows to refer to them uniformly. This makes it easier to read the following equations.

Valuation functions

We assume the semantic **let** construct to be strict. This ensures the call-by-value nature of the defined language. We also leave out the injection and projection of summands, for simplicity.

$$\begin{array}{rcl}
\mathcal{E}[\![c]\!]\rho & = & \mathcal{C}[\![c]\!] \\
\mathcal{E}[\![i]\!]\rho & = & \rho\, i \\
\mathcal{E}[\![l]\!]\rho & = & \mathcal{L}[\![l]\!]\rho \\
\mathcal{E}[\![@\, e_0\,(e_1, ..., e_n)]\!]\rho & = & \textbf{let}\ v_0 = \mathcal{E}[\![e_0]\!]\rho,\ v_1 = \mathcal{E}[\![e_1]\!]\rho,\ ...,\ v_n = \mathcal{E}[\![e_n]\!]\rho \\
 & & \textbf{in}\ v_0(v_1, ..., v_n) \\
\mathcal{E}[\![op\,(e_1, ..., e_m)]\!]\rho & = & \textbf{let}\ v_1 = \mathcal{E}[\![e_1]\!]\rho,\ ...,\ v_m = \mathcal{E}[\![e_m]\!]\rho \\
 & & \textbf{in}\ \mathcal{O}[\![op]\!]\,(v_1, ..., v_m) \\
\mathcal{E}[\![e_1 \rightarrow e_2, e_3]\!]\rho & = & \textbf{let}\ b = \mathcal{E}[\![e_0]\!]\rho\ \textbf{in}\ b \rightarrow \mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![\text{let}\,(i_1, ..., i_n) = (e_1, ..., e_n)\,\text{in}\,e_0]\!]\rho & = & \textbf{let}\ v_1 = \mathcal{E}[\![e_1]\!]\rho,\ ...,\ v_n = \mathcal{E}[\![e_n]\!]\rho \\
 & & \textbf{in}\ \mathcal{E}[\![e_0]\!]\rho[i_1 \mapsto v_1, ..., i_n \mapsto v_n] \\
\mathcal{E}[\![\text{letrec}\,(i_1, ..., i_n) = (l_1, ..., l_n)\,\text{in}\,e_0]\!]\rho & = & \textbf{letrec}\,(f_1, ..., f_n) = (\mathcal{L}[\![l_1]\!]\rho[i_1 \mapsto f_1, ..., i_n \mapsto f_n], \\
 & & \qquad ..., \\
 & & \qquad \mathcal{L}[\![l_n]\!]\rho[i_1 \mapsto f_1, ..., i_n \mapsto f_n]) \\
 & & \textbf{in}\ \mathcal{E}[\![e_0]\!]\rho[i_1 \mapsto v_1, ..., i_n \mapsto v_n] \\
\mathcal{L}[\![\lambda\,(i_1, ..., i_n)\,.\,e]\!]\rho & = & \lambda(v_1, ..., v_n).\mathcal{E}[\![e]\!]\rho[i_1 \mapsto v_1, ..., i_n \mapsto v_n]
\end{array}$$

The meaning of a term $[\![e]\!]$ is given by $\mathcal{E}[\![e]\!]\,\rho_{init}$ where ρ_{init} denotes the initial environment.

References

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

[2] Anders Bondorf. Automatic autoprojection of higher-order recursive equations. *Science of Computer Programming*, 1991. To appear.

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

[4] Charles Consel and Olivier Danvy. For a better support of static data flow. In *Proceedings of the 1991 Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 496–519, Cambridge, Massachusetts, August 1991. Springer-Verlag.

[5] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.

[6] Olivier Danvy. Three steps for the CPS transformation. Technical Report CIS-92-2, Kansas State University, Manhattan, Kansas, 1992.

[7] Olivier Danvy and Andrzej Filinski. Abstracting control. In LFP'90 [23], pages 151–160.

[8] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. Technical Report CIS-91-2, Kansas State University, Manhattan, Kansas, 1991.

[9] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. Technical Report CIS-92-1, Kansas State University, Manhattan, Kansas, 1992.

[10] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[11] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report Rice COMP TR89-100, Department of Computer Science, Rice University, Houston, Texas, June 1989. To appear in Theoretical Computer Science.

[12] Andrzej Filinski. Declarative continuations: An investigation of duality in programming language semantics. In D.H. Pitt et al., editors, *Category Theory and Computer Science*, number 389 in Lecture Notes in Computer Science, pages 224–249, Manchester, UK, September 1989.

[13] Michael J. Fischer. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, pages 104–109. SIGPLAN Notices, Vol. 7, No 1 and SIGACT News, No 14, January 1972.

[14] Pascal Fradet and Daniel Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13:21–51, 1991.

[15] Daniel P. Friedman. Applications of continuations. Report 237, Computer Science Department, Indiana University, Bloomington, Indiana, January 1988. Tutorial of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California.

[16] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* MIT Press and McGraw-Hill, 1991.

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

[18] Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 296–306, January 1986.

[19] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis (chapter in preparation). In *The Handbook of Logic in Computer Science.* North-Holland, 1991.

[20] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[21] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 281–292, Austin, Texas, January 1989.

[22] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Languages Design and Implementation*, pages 338–352, Toronto, Ontario, June 1991.

[23] *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[24] Antoni W. Mazurkiewicz. Proving algorithms by tail functions. *Information and Control*, 18:220–226, 1971.

[25] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE.

[26] Chetan R. Murthy. An evaluation semantics for classical proofs. In *Proceedings of the Sixth Symposium on Logic in Computer Science*, Amsterdam, The Netherlands, July 1991. IEEE.

[27] Flemming Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[28] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988.

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

[30] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, 1972.

[31] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[32] 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.

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

[34] Adriaan van Wijngaarden. Recursive definition of syntax and semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming*, pages 13–24. North-Holland, 1966.

[35] Philip Wadler. Comprehending monads. In LFP'90 [23], pages 61–78.

[36] Mitchell Wand. Semantics-directed machine architecture. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 234–241, January 1982.

[37] Mitchell Wand. A short proof of the lexical addressing algorithm. *Information Processing Letters*, 35:1–5, 1990.

```
(lambda (f l)       ; [A -> B] * List(A) -> List(B)
  (letrec ([loop (lambda (l)
                   (if (null? l)
                       '()
                       (cons (f (car l)) (loop (cdr l)))))])
    (loop l)))

(lambda (k)
  (k (lambda (f l k)    ; [A * [B -> Ans] -> Ans] * List(B) * [List(B) -> Ans] -> Ans
       (letrec ([loop (lambda (l k)
                        (if (null? l)
                            (k '())
                            (f (car l) (lambda (v)
                                         (loop (cdr l) (lambda (vs)
                                                         (k (cons v vs))))))))])
         (loop l k)))))
```

Figure 5: Interconvertible DS and CPS definitions of the map procedure in Scheme
As can be noticed, the CPS transformation commits the order of evaluation of sub-expressions in an application, which is not in the true spirit of Scheme [3].

```
(lambda (x) x)                          (lambda (k) (k (lambda (x k) (k x))))
```

Figure 6: Interconvertible DS and CPS definitions of the identity procedure in Scheme