On the Transformation Between Direct and Continuation Semantics *

Olivier Danvy and John Hatcliff

Aarhus University ** and Kansas State University ***

Abstract. Proving the congruence between a direct semantics and a continuation semantics is often surprisingly complicated considering that direct-style λ-terms can be transformed into continuation style automatically. However, transforming the representation of a direct-style semantics into continuation style usually does *not* yield the expected representation of a continuation-style semantics (*i.e.*, one written by hand).

The goal of our work is to automate the transformation between textual representations of direct semantics and of continuation semantics. Essentially, we identify properties of a direct-style representation (*e.g.*, totality), and we generalize the transformation into continuation style accordingly. As a result, we can produce the expected representation of a continuation semantics, automatically.

It is important to understand the transformation between representations of direct and of continuation semantics because it is these representations that get processed in any kind of semantics-based program manipulation (*e.g.*, compiling, compiler generation, and partial evaluation). A tool producing a variety of continuation-style representations is a valuable new one in a programming-language workbench.

* 9th Conference on Mathematical Foundations of Programming Semantics. New Orleans, Louisiana, April 1993.

** Department of Computer Science, Ny Munkegade, 8000 Aarhus C, Denmark. E-mail: danvy@daimi.aau.dk — This work was initiated at Kansas State University, continued at Carnegie Mellon University, and completed at Aarhus University. It was partly supported by NSF under grant CCR-9102625.

*** Department of Computing and Information Sciences, Manhattan, Kansas 66506, USA. E-mail: hatcliff@cis.ksu.edu

1 Introduction

Proving the congruence between a denotational-semantics specification in direct style and a denotational-semantics specification in continuation style is not trivial [26, 28, 31]. Yet,

- both direct-style and continuation-style specifications can be represented as typed λ-terms: semantic domains are represented with types, and valuation functions with λ-terms [22, 28];
- typed λ-terms can be transformed into continuation style *automatically* using Plotkin's continuation-passing-style (CPS) transformation [9, 15, 24].

We have transformed the representation of several direct-style specifications into continuation style. Since the meta-language of denotational semantics obeys normal order [28], we have used the call-by-name CPS transformation. The result is *not* the expected representation of a continuation-style semantics (*i.e.*, one written by hand).

1.1 An example

It is sufficient to look at types to see where a mismatch occurs.

$$\mathcal{C}_d[\![\cdot]\!] : Env_d \to Com_d \quad \text{where} \quad Com_d = Store \to Store$$

$$\mathcal{C}_c[\![\cdot]\!] : Env_c \to Com_c \quad \text{where} \quad Com_c = Store \to (Store \to Ans) \to Ans$$

Fig. 1. Types of valuation functions for a simple imperative language

Figure 1 gives the types of two valuation functions for a simple imperative language. $\mathcal{C}_d[\![\cdot]\!]$ is a direct-style valuation function and $\mathcal{C}_c[\![\cdot]\!]$ is a continuation-style valuation function.

Figure 2 displays Plotkin's call-by-name CPS transformation $\mathcal{C}_{\mathrm{n}}$ for typed terms [9, 24]. ι represents a base type.

Transforming the types of the direct-style valuation function $\mathcal{C}_d[\![\cdot]\!]$ does not yield the types of the continuation-style valuation function $\mathcal{C}_c[\![\cdot]\!]$. For example, the transformation of the function space $Env_d \to Com_d$ yields

$$((\mathcal{C}_{\mathrm{n}}\langle Env_d\rangle \to Ans) \to Ans) \to (\mathcal{C}_{\mathrm{n}}\langle Com_d\rangle \to Ans) \to Ans$$

which does not match the type of the corresponding function space

$$Env_c \to Com_c$$

in $\mathcal{C}_c[\![\cdot]\!]$. Essentially, $\mathcal{C}_{\mathrm{n}}$ introduces too many continuations.

$$
\begin{aligned}
\mathcal{C}_{\mathrm{n}}[\![i]\!] &= i \\
\mathcal{C}_{\mathrm{n}}[\![\lambda i : t.\epsilon]\!] &= \lambda\kappa.\kappa\,(\lambda i : \mathcal{C}_{\mathrm{n}}[\![t]\!].\mathcal{C}_{\mathrm{n}}[\![\epsilon]\!]) \\
\mathcal{C}_{\mathrm{n}}[\![\epsilon_0\ \epsilon_1]\!] &= \lambda\kappa.\mathcal{C}_{\mathrm{n}}[\![\epsilon_0]\!]\,(\lambda v_0.(v_0\ \mathcal{C}_{\mathrm{n}}[\![\epsilon_1]\!])\,\kappa)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_{\mathrm{n}}\langle\iota\rangle &= \iota \\
\mathcal{C}_{\mathrm{n}}\langle t_0 \rightarrow t_1\rangle &= \mathcal{C}_{\mathrm{n}}[\![t_0]\!] \rightarrow \mathcal{C}_{\mathrm{n}}[\![t_1]\!] \\
\mathcal{C}_{\mathrm{n}}[\![t]\!] &= (\mathcal{C}_{\mathrm{n}}\langle t\rangle \rightarrow Ans) \rightarrow Ans
\end{aligned}
$$

Fig. 2. Transformation of call-by-name λ-terms into continuation style

This mismatch is significant because it shows that a continuation semantics is not just a direct semantics with continuations. For another example, in a continuation semantics, environments (represented as functions) usually are expressed in direct style, *i.e.*, they are not passed any continuation [28, 31].

1.2 A choice

At this point we have a choice:

- We could establish the relationship between the result of CPS-transforming [the representation of] a direct-style semantics and [the representation of] a continuation-style semantics that one would write by hand, and maybe map one into the other.
- We could devise a new CPS transformation that would transform [the representation of] a direct-style semantics into [the representation of] a realistic continuation-style semantics. By a "realistic" continuation-style semantics, we mean "one that a professional denotational-semanticist would write".

We choose the latter option.

1.3 On the transformation between direct and continuation semantics

The goal of our work is to automate the transformation between textual representations of direct semantics and of continuation semantics. Essentially, we identify properties of a direct-style representation (*e.g.*, totality), and we generalize the call-by-name CPS transformation accordingly. As a result, we can produce the expected representation of a realistic continuation semantics, automatically.

It is important to understand the transformation between representations of direct and of continuation semantics for at least three reasons.

1. It is these representations that get processed in any kind of semantics-based program manipulation (*e.g.*, compiling, compiler generation, and partial evaluation).

2. The properties of [the representation of] the direct-style semantics should give precious insights to establishing the congruence relation between the direct semantics and the continuation semantics.
3. The properties of the transformation should give guidelines for proving the congruence between the direct semantics and the continuation semantics.

1.4 Issues

The tools used in this paper are interesting in their own right.

1. The generalized call-by-name CPS transformation is based on a system of annotations capturing reduction properties such as partiality and totality. Using these annotations, we extend Reynolds's classification of trivial and serious λ-*terms*[4] to serious and trivial *functions*. The extended classification gives a finer scheme for describing termination properties of terms — unlike Reynolds's original scheme, it allows us to state that *e.g.*, some applications are actually trivial.
2. The annotations are obtained by an automatic control-flow analysis that extends Mycroft's $\flat$ termination analysis to higher-order programs [20, 21]. This tool has applications in other areas such as compiling and partial evaluation.
3. Retaining Reynolds's method of introducing continuations in serious terms yields a transformation that introduces continuations only when necessary to achieve evaluation-order independence. Thus, this new transformation generalizes the call-by-name CPS transformation (should all functions be serious) and the identity transformation (should all functions be trivial). In an earlier work, we reported a CPS transformation after strictness analysis that generalizes the call-by-value CPS transformation (should all constructs be strict) and the call-by-name CPS transformation (should all constructs be non-strict) [5]. Tools producing a variety of continuation-style representations are valuable new ones in a programming-language workbench.

1.5 Organization

The rest of this paper is organized as follows. Section 2 presents an example language and two semantic definitions, one in direct style and one in continuation style. Section 3 describes how these semantic definitions can be represented as typed λ-terms. In Section 4, we generalize Reynolds's notion of trivial and serious terms. In Section 5, we extend the transformation into continuation style to handle terms with annotations describing trivial and serious properties. In Section 6, we examine the properties of the representation of the direct-style semantics of Section 2 and we annotate this representation. In Section 7, we

[4] Reducing a trivial λ-term always terminates whereas reducing a serious λ-term may not terminate [25]. Reynolds's notion of trivial λ-term coincides with Plotkin's notion of value [24].

$z \in$ Program $\quad l \in$ Location
$c \in$ Command $\quad m \in$ Ident[num]
$e \in$ Expression $\quad p \in$ Ident[proc]
$n \in$ Numeral $\quad f \in$ Ident[fun]

$z ::= \text{proc}\, p\,(m) = c \,\text{in}\, z \mid \text{fun}\, f\,(m) = e \,\text{in}\, z \mid c.$
$c ::= \text{skip} \mid c_1\,;\, c_2 \mid l := e \mid \text{if}\, e \,\text{then}\, c_1 \,\text{else}\, c_2 \mid \text{while}\, e \,\text{do}\, c \mid \text{call}\, p\,(e)$
$e ::= n \mid m \mid \text{succ}\, e \mid \text{pred}\, e \mid \text{deref}\, l \mid \text{apply}\, f\,(e)$

Fig. 3. Abstract syntax of the simple imperative language

transform this annotated representation into continuation style and we obtain the expected representation of a continuation semantics. Finally, Section 8 concludes and puts this work into perspective.

2 Example Denotational Definitions

Figure 3 presents the abstract syntax of a simple imperative language with global and non-recursive first-order procedures. Figures 4 and 5 give a direct semantics and a continuation semantics for the simple imperative language. The functionality of the semantic algebras for stores, environments, and natural numbers are the usual ones and the specifications are omitted.

Proposition 1. *The semantics of Figures 4 and 5 define the same language, that is, they are congruent [31, page 340].* □

3 Representing Denotational Definitions as Typed λ-terms

Denotational definitions are usually implemented by *treating the semantic notation as a "machine language"* [28, Section 10.1]. A common notation of denotational semantics is the λ-calculus. Thus, domains are mapped into types and domain constructors into type constructors, valuation functions are mapped into λ-expressions, and semantic-algebra operations into δ-rules. Figure 6 presents the syntax of an extended λ-calculus used to represent denotational definitions as typed terms. The typing rules are the usual ones and are omitted. When e has type t under type assumptions π we write $\pi \vdash e : t$. Each element of the set of type assumptions π is of the form $i : t$. To simplify substitution, and without loss of generality, we assume that all identifiers are unique.

Let us summarize how the example denotational definitions of Figure 4 and 5 are represented by the typed terms of Figure 6.

Valuation Functions:

$$\begin{aligned}
\mathcal{Z}[\![\text{Program}]\!] &: \mathit{Env} \to \mathit{Com} \\
\mathcal{C}[\![\text{Command}]\!] &: \mathit{Env} \to \mathit{Com} \\
\mathcal{E}[\![\text{Expression}]\!] &: \mathit{Env} \to \mathit{Exp} \\
\mathcal{N}[\![\text{Numeral}]\!] &: \mathit{Nat} \\
\mathcal{L}[\![\text{Location}]\!] &: \mathit{Loc}
\end{aligned}$$

Semantic Domains:

$$\begin{aligned}
\mathit{Com} &= \mathit{Store} \to \mathit{Store}_\perp \\
\mathit{Exp} &= \mathit{Store} \to \mathit{Nat} \\
\mathit{Proc} &= \mathit{Nat} \to \mathit{Com} \\
\mathit{Fun} &= \mathit{Nat} \to \mathit{Exp}
\end{aligned}$$

Programs:

$$\begin{aligned}
\mathcal{Z}[\![\text{proc } p\ (m) = c \text{ in } z]\!] &= \lambda\rho.\lambda\sigma.\mathcal{Z}[\![z]\!]\,(\mathsf{ext}\ \rho\ p\ (\lambda i.\mathcal{C}[\![c]\!]\,(\mathsf{ext}\ \rho\ m\ i)))\,\sigma \\
\mathcal{Z}[\![\text{fun } f\ (m) = e \text{ in } p]\!] &= \lambda\rho.\lambda\sigma.\mathcal{Z}[\![z]\!]\,(\mathsf{ext}\ \rho\ f\ (\lambda i.\mathcal{E}[\![e]\!]\,(\mathsf{ext}\ \rho\ m\ i)))\,\sigma \\
\mathcal{Z}[\![c.]\!] &= \lambda\rho.\lambda\sigma.\mathcal{C}[\![c]\!]\,\rho\,\sigma
\end{aligned}$$

Commands:

$$\begin{aligned}
\mathcal{C}[\![\text{skip}]\!] &= \lambda\rho.\lambda\sigma.\sigma \\
\mathcal{C}[\![c_1\ ;\ c_2]\!] &= \lambda\rho.\lambda\sigma.\mathit{let}\ \sigma' = \mathcal{C}[\![c_1]\!]\,\rho\,\sigma\ \mathit{in}\ \mathcal{C}[\![c_2]\!]\,\rho\,\sigma' \\
\mathcal{C}[\![l := e]\!] &= \lambda\rho.\lambda\sigma.\mathsf{upd}\ \sigma\ \mathcal{L}[\![l]\!]\,(\mathcal{E}[\![e]\!]\,\rho\,\sigma) \\
\mathcal{C}[\![\text{if } e \text{ then } c_1 \text{ else } c_2]\!] &= \lambda\rho.\lambda\sigma.\mathit{if}\ \mathsf{iszero?}\ (\mathcal{E}[\![e]\!]\,\rho\,\sigma)\ \mathit{then}\ (\mathcal{C}[\![c_1]\!]\,\rho\,\sigma)\ \mathit{else}\ (\mathcal{C}[\![c_2]\!]\,\rho\,\sigma) \\
\mathcal{C}[\![\text{while } e \text{ do } c]\!] &= \lambda\rho.\lambda\sigma.\ \mathit{letrec}\ w = \lambda\sigma.\ \mathit{if}\ \ \mathsf{iszero?}\ (\mathcal{E}[\![e]\!]\,\rho\,\sigma) \\
&\qquad\qquad \mathit{then}\ \mathit{let}\ \sigma' = \mathcal{C}[\![c]\!]\,\rho\,\sigma\ \mathit{in}\ w\,\sigma' \\
&\qquad\qquad \mathit{else}\ \ \sigma \\
&\qquad \mathit{in}\ w\,\sigma \\
\mathcal{C}[\![\text{call } p\ (e)]\!] &= \lambda\rho.\lambda\sigma.(\mathsf{lookup}\ \rho\ p)\,(\mathcal{E}[\![e]\!]\,\rho\,\sigma)\,\sigma
\end{aligned}$$

Expressions:

$$\begin{aligned}
\mathcal{E}[\![n]\!] &= \lambda\rho.\lambda\sigma.\mathcal{N}[\![n]\!] \\
\mathcal{E}[\![m]\!] &= \lambda\rho.\lambda\sigma.\mathsf{lookup}\ \rho\ m \\
\mathcal{E}[\![\text{succ } e]\!] &= \lambda\rho.\lambda\sigma.\mathsf{succ}\ (\mathcal{E}[\![e]\!]\,\rho\,\sigma) \\
\mathcal{E}[\![\text{pred } e]\!] &= \lambda\rho.\lambda\sigma.\mathsf{pred}\ (\mathcal{E}[\![e]\!]\,\rho\,\sigma) \\
\mathcal{E}[\![\text{deref } l]\!] &= \lambda\rho.\lambda\sigma.\mathsf{fetch}\ \sigma\ \mathcal{L}[\![l]\!] \\
\mathcal{E}[\![\text{apply } f\ (e)]\!] &= \lambda\rho.\lambda\sigma.(\mathsf{lookup}\ \rho\ f)\,(\mathcal{E}[\![e]\!]\,\rho\,\sigma)\,\sigma
\end{aligned}$$

Fig. 4. Direct semantics of the simple imperative language

Valuation Functions:

$$\begin{aligned}
\mathcal{Z}[\![\text{Program}]\!] &: \mathit{Env} \to \mathit{Com} \\
\mathcal{C}[\![\text{Command}]\!] &: \mathit{Env} \to \mathit{Com} \\
\mathcal{E}[\![\text{Expression}]\!] &: \mathit{Env} \to \mathit{Exp} \\
\mathcal{N}[\![\text{Numeral}]\!] &: \mathit{Nat} \\
\mathcal{L}[\![\text{Location}]\!] &: \mathit{Loc}
\end{aligned}$$

Semantic Domains:

$$\begin{aligned}
\mathit{Com} &= \mathit{Store} \to (\mathit{Store} \to \mathit{Ans}) \to \mathit{Ans} \\
\mathit{Exp} &= \mathit{Store} \to \mathit{Nat} \\
\mathit{Proc} &= \mathit{Nat} \to \mathit{Com} \\
\mathit{Fun} &= \mathit{Nat} \to \mathit{Exp}
\end{aligned}$$

Programs:

$$\begin{aligned}
\mathcal{Z}[\![\text{proc } p\ (m) = c \text{ in } z]\!] &= \lambda\rho.\lambda\sigma.\lambda\kappa.\mathcal{Z}[\![z]\!]\,(\mathsf{ext}\ \rho\ p\ (\lambda i.\mathcal{C}[\![c]\!]\,(\mathsf{ext}\ \rho\ m\ i)))\,\sigma\,\kappa \\
\mathcal{Z}[\![\text{fun } f\ (m) = e \text{ in } z]\!] &= \lambda\rho.\lambda\sigma.\lambda\kappa.\mathcal{Z}[\![z]\!]\,(\mathsf{ext}\ \rho\ f\ (\lambda i.\mathcal{E}[\![e]\!]\,(\mathsf{ext}\ \rho\ m\ i)))\,\sigma\,\kappa \\
\mathcal{Z}[\![c.]\!] &= \lambda\rho.\lambda\sigma.\lambda\kappa.\mathcal{C}[\![c]\!]\,\rho\,\sigma\,\kappa
\end{aligned}$$

Commands:

$$\begin{aligned}
\mathcal{C}[\![\text{skip}]\!] &= \lambda\rho.\lambda\sigma.\lambda\kappa.\kappa\,\sigma \\
\mathcal{C}[\![c_1\ ;\ c_2]\!] &= \lambda\rho.\lambda\sigma.\lambda\kappa.\mathcal{C}[\![c_1]\!]\,\rho\,\sigma\,(\lambda\sigma'.\mathcal{C}[\![c_2]\!]\,\rho\,\sigma'\,\kappa) \\
\mathcal{C}[\![l := e]\!] &= \lambda\rho.\lambda\sigma.\lambda\kappa.\kappa\,(\mathsf{upd}\ \sigma\ \mathcal{L}[\![l]\!]\,(\mathcal{E}[\![e]\!]\,\rho\,\sigma)) \\
\mathcal{C}[\![\text{if } e \text{ then } c_1 \text{ else } c_2]\!] &= \lambda\rho.\lambda\sigma.\lambda\kappa.\ \mathit{if}\ \mathsf{iszero?}\ (\mathcal{E}[\![e]\!]\,\rho\,\sigma)\ \mathit{then}\ (\mathcal{C}[\![c_1]\!]\,\rho\,\sigma\,\kappa) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else}\ (\mathcal{C}[\![c_2]\!]\,\rho\,\sigma\,\kappa) \\
\mathcal{C}[\![\text{while } e \text{ do } c]\!] &= \lambda\rho.\lambda\sigma.\lambda\kappa.\ \mathit{letrec}\ w = \lambda\sigma.\lambda\kappa'.\ \mathit{if}\quad \mathsf{iszero?}\ (\mathcal{E}[\![e]\!]\,\rho\,\sigma) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{then}\ \mathcal{C}[\![c]\!]\,\rho\,\sigma\,(\lambda\sigma'.w\,\sigma'\,\kappa') \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{else}\ \ \kappa'\,\sigma \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{in}\ w\,\sigma\,\kappa \\
\mathcal{C}[\![\text{call } p\ (e)]\!] &= \lambda\rho.\lambda\sigma.\lambda\kappa.(\mathsf{lookup}\ \rho\ p)\,(\mathcal{E}[\![e]\!]\,\rho\,\sigma)\,\sigma\,\kappa
\end{aligned}$$

Expressions:

$$\begin{aligned}
\mathcal{E}[\![n]\!] &= \lambda\rho.\lambda\sigma.\mathcal{N}[\![n]\!] \\
\mathcal{E}[\![m]\!] &= \lambda\rho.\lambda\sigma.\mathsf{lookup}\ \rho\ m \\
\mathcal{E}[\![\text{succ } e]\!] &= \lambda\rho.\lambda\sigma.\mathsf{succ}\ (\mathcal{E}[\![e]\!]\,\rho\,\sigma) \\
\mathcal{E}[\![\text{pred } e]\!] &= \lambda\rho.\lambda\sigma.\mathsf{pred}\ (\mathcal{E}[\![e]\!]\,\rho\,\sigma) \\
\mathcal{E}[\![\text{deref } l]\!] &= \lambda\rho.\lambda\sigma.\mathsf{fetch}\ \sigma\ \mathcal{L}[\![l]\!] \\
\mathcal{E}[\![\text{apply } f\ (e)]\!] &= \lambda\rho.\lambda\sigma.(\mathsf{lookup}\ \rho\ f)\,(\mathcal{E}[\![e]\!]\,\rho\,\sigma)\,\sigma
\end{aligned}$$

Fig. 5. Continuation semantics of the simple imperative language

$$\epsilon \in \lambda\text{-}exp \qquad a \in \lambda\text{-}abs \qquad r \in \delta\text{-}rule$$

$$\epsilon ::= i \mid a \mid \epsilon_0\,\epsilon_1 \mid \mathit{if}\ \epsilon_1\ \mathit{then}\ \epsilon_2\ \mathit{else}\ \epsilon_3 \mid \mathit{let}\ i = \epsilon_0\ \mathit{in}\ \epsilon_1 \mid \mathit{letrec}\ i = a\ \mathit{in}\ \epsilon \mid r$$
$$a ::= \lambda i : t.\epsilon$$
$$r ::= \mathsf{op}_0 \mid \mathsf{op}_1\,\epsilon \mid \mathsf{op}_2\,\epsilon_1\,\epsilon_2 \mid \ldots$$

$$t \in \mathit{Type}$$
$$t ::= \iota \mid t_0 \rightarrow t_1$$

Fig. 6. Abstract syntax for an extended typed λ-calculus

The primitive domains *Store*, *Env*, and *Ide* form the base types and the semantic-algebra operations such as upd and fetch become δ-rules.

The valuation functions become typed λ-terms. A key point in this step is that operational notions such as non-termination and recursion (represented explicitly in the denotational semantics by the special element $\perp$ and least fixed-point operations over *cpo*'s) must be captured implicitly in the reduction properties of the λ-terms.

Following Schmidt [28], *let* expressions used in the direct semantics of Figure 4 include a strictness check over some lifted domain $A_\perp$. They are defined as follows.

$$\mathit{let}\ i = e_0\ \mathit{in}\ e_1 \;=\; \begin{cases} \perp & \mathit{if}\ e_0 = \perp \\ (\lambda i.e_1)\,e_0 & \mathit{otherwise} \end{cases}$$

So each *let* expression is represented with an eager binding construct. The operational behavior of the binding construct (*i.e.*, call-by-value) captures the appropriate termination properties [24].

Similarly, *letrec* is defined by the usual desugaring into the fixed-point operator. So each *letrec* expression is represented with the usual recursive binding construct. Its operational behavior approximates the computation of the least fixed-point of a function.

4 Analyzing the Representation of a Direct-Style Definition

As pointed out in Section 1.1, transforming the λ-representation of a direct semantics into continuation style using a call-by-name transformation does *not* yield the λ-representation of a realistic continuation semantics. Essentially, the transformation introduces more continuations than are needed.[5] In this section,

[5] For example, in Figure 5, $\mathcal{E}$ is expressed in direct style even though it is part of a continuation semantics. We aim to clarify why $\mathcal{E}$ does not need any continuation, and to establish conditions that allow one to transform the text of Figure 4 into the text of Figure 5, automatically.

we go back to the source [25] and investigate where continuations are really necessary.

4.1 Reynolds's notion of trivial and serious terms

Originally, Reynolds distinguished between "trivial" terms (whose evaluation never diverges) and "serious" terms (whose evaluation might diverge) [25]. Trivial terms correspond to Plotkin's notion of "value" [24]. Since introducing continuations aims at obtaining evaluation-order independence, only serious terms need to be transformed into continuation style. As an approximation, Reynolds decided that all applications are serious terms and thus they all need a continuation — forcing each function to be passed a continuation.

Considering the particular case of denotational semantics, this approximation often is too coarse. For example, valuation functions are usually curried. Most of the time, the result of applying a valuation function to an abstract-syntax tree is a λ-abstraction. In fact, this is the case for $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{E}$ in Figure 4. Since a λ-abstraction is a trivial term, applying a valuation function does not yield a serious term. Thus it is too conservative to approximate all applications as serious terms.[6]

4.2 Trivial and serious functions

In a denotational-semantics specification, a function is defined textually as a λ-abstraction.

- If the body of this λ-abstraction is trivial, the function is obviously *total.* Since evaluating the body does not require a continuation, the function does not need a continuation either.
- Conversely, if the body of a λ-abstraction is serious, the corresponding function may be *partial.* Since evaluating the body requires a continuation, the function needs to be passed this continuation.

We refer to such λ-abstractions as "trivial functions" and "serious functions", respectively.

Let us now turn to the arguments of these functions. Denotational specifications are customarily higher-order, so it is not obvious which expression occurs as the argument of which λ-abstraction. However following Reynolds again [25], we can enumerate the λ-abstractions that may occur in each higher-order application. This enumeration is achieved by control-flow analysis (a.k.a. closure analysis) [29, 30].

---

[6] This is probably why Reynolds's definitional interpreters are uncurried [25].

4.3 Call-by-value and call-by-name functions

A λ-abstraction can be applied to a trivial argument or to a serious one. Again, trivial arguments do not need to be computed with a continuation. Conversely, serious arguments need to be computed with a continuation. We approximate this situation by stating that if a λ-abstraction is always applied to trivial arguments, we can pass the arguments as they are, and that if a function may be applied to a serious argument, then all arguments are computed with a continuation. By analogy with the fact that evaluating a trivial expression must yield a value, we refer to the former λ-abstractions as "call-by-value functions" and to the latter as "call-by-name functions". So let us consider the four cases of λ-abstractions:

1. trivial call-by-value functions (*i.e.*, λ-abstractions whose bodies are trivial and that are applied to trivial arguments);
2. trivial call-by-name functions (*i.e.*, λ-abstractions whose bodies are trivial and that are applied to serious arguments);
3. serious call-by-value functions (*i.e.*, λ-abstractions whose bodies are serious and that are applied to trivial arguments);
4. serious call-by-name functions (*i.e.*, λ-abstraction whose bodies are serious and that are applied to serious arguments).

Correspondingly, a variable declared in a call-by-value (resp. call-by-name) function is a trivial (resp. serious) expression.

In the following section, we describe how to annotate λ-abstractions and applications to account for their triviality and their seriousness, and for their mode of parameter passing.

5 Annotating the Representation of a Direct-Style Definition

Figure 7 presents the syntax of the annotated λ-calculus. Essentially, we introduce the explicit infix notation "@" in applications, and we tag the constructs and types that depart from standard call-by-name.[7]

Constructs and types associated with trivial functions are annotated with "t". Constructs and types associated with call-by-value are annotated with "v". Constructs and types associated with both are annotated with "tv". For example, $\lambda_{tv}x.e$ denotes a call-by-value trivial λ-abstraction. y is a variable declared in a call-by-name λ-abstraction. z_v is a variable declared in a call-by-value λ-abstraction. $e_0 \mathbin{@} e_1$ denotes the application of a call-by-name serious function to an argument. $e'_0 \mathbin{@_t} e'_1$ denotes the application of a call-by-name trivial function to an argument.

We also use the annotations *trivial* and *serious* to tag trivial and serious expressions. The annotation tags form a partially ordered set $(AnnTag, \sqsubseteq)$ where

[7] This follows the spirit of the diacritical convention: only the terms whose meaning is farthest to the original meaning (*i.e.*, call-by-name) are annotated [16, 31].

$\epsilon \in \mathit{Ann}\text{-}\lambda\text{-}\mathit{exp}$ $\quad a \in \mathit{Ann}\text{-}\lambda\text{-}\mathit{abs}$ $\quad r \in \delta\text{-}\mathit{rule}$

$$\epsilon ::= i_v \mid i \mid a \mid \epsilon_0 @_w \epsilon_1 \mid \epsilon_0 @_v \epsilon_1 \mid \epsilon_0 @_t \epsilon_1 \mid \epsilon_0 @ \epsilon_1$$
$$\mid \mathit{if}\ \epsilon_1\ \mathit{then}\ \epsilon_2\ \mathit{else}\ \epsilon_3 \mid \mathit{let}\ i = \epsilon_0\ \mathit{in}\ \epsilon_1 \mid \mathit{letrec}\ i = a\ \mathit{in}\ \epsilon \mid r$$
$$a ::= \lambda_w i : \tau.\epsilon \mid \lambda_v i : \tau.\epsilon \mid \lambda_t i : \tau.\epsilon \mid \lambda i : \tau.\epsilon$$
$$r ::= \mathsf{op}_0 \mid \mathsf{op}_1\,\epsilon \mid \mathsf{op}_2\,\epsilon_1\,\epsilon_2 \mid \ldots$$

$\tau \in \mathit{AnnType}$

$$\tau ::= \iota \mid \tau_0 \rightarrow_w \tau_1 \mid \tau_0 \rightarrow_v \tau_1 \mid \tau_0 \rightarrow_t \tau_1 \mid \tau_0 \rightarrow \tau_1$$

$\alpha \in \mathit{AnnTag}$

$$\alpha ::= \mathit{trivial} \mid \mathit{serious}$$

Fig. 7. Abstract syntax for the annotated typed λ-calculus

$\mathit{trivial} \sqsubseteq \mathit{serious}$. They will contribute to characterizing reduction properties of individual terms.

Figure 8 presents type-annotation rules for the annotated λ-calculus. Each term is associated with a pair (τ, α). The first component $\tau \in \mathit{AnnType}$ is an annotated type. The second component $\alpha \in \mathit{AnnTag}$ indicates whether the term is trivial or serious. Γ is a set of type assumptions where each element is of the form $i : \tau$. For simplicity, we assume that all identifier names are unique, and that the algebraic operators cannot diverge.

The other binding constructs also warrant explanation. In the *let* construct, the actual parameter e_1 may be either trivial or serious. However, due to the eager evaluation of e_1, i always binds to a value and thus is annotated as trivial. Of course, binding may not occur at all due to the diverging evaluation of a serious e_1. This is captured by the fact that a serious e_1 causes the entire construct to be classified as serious. In the *letrec* construct, the declared identifier f always binds to a λ-abstraction and is thus annotated as trivial.

Note that there is redundancy in the given annotation scheme. In particular, annotation pairs (τ, α) are sufficient for our purposes.[8] Annotations on terms have been included to simplify the presentation of the transformation into continuation style in Section 5.2.

5.1 Correct assignment of annotations

To formalize the correctness of the annotation rules, let us introduce the following notation. $\Downarrow_n$ and $\Downarrow_v$ respectively denote the relations defined by a call-by-name

[8] The annotation scheme can also be phrased more elegantly in terms of Moggi's computational metalanguage [17] — *serious* terms are typed as *computations*, *trivial* terms are typed as *values*. This point is developed elsewhere [10, 11].

Identifiers:

$$\Gamma \cup \{i : \tau\} \vdash_a i : (\tau\,,\ serious) \qquad\qquad \Gamma \cup \{i_v : \tau\} \vdash_a i_v : (\tau\,,\ trivial)$$

Primitive Operators:

$$\Gamma \vdash_a \mathsf{op}_0 : (\iota\,,\ trivial) \qquad \frac{\Gamma \vdash_a e_1 : (\iota\,,\ \alpha)}{\Gamma \vdash_a \mathsf{op}_1\, e_1 : (\iota\,,\ \alpha)} \qquad \frac{\Gamma \vdash_a e_1 : (\iota\,,\ \alpha_1) \quad \Gamma \vdash_a e_2 : (\iota\,,\ \alpha_2)}{\Gamma \vdash_a \mathsf{op}_2\, e_1\, e_2 : (\iota\,,\ \alpha_1 \sqcup \alpha_2)}$$

Conditional:

$$\frac{\Gamma \vdash_a e_1 : (\iota\,,\ \alpha_1) \quad \Gamma \vdash_a e_2 : (\tau\,,\ \alpha_2) \quad \Gamma \vdash_a e_3 : (\tau\,,\ \alpha_3)}{\Gamma \vdash_a \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : (\tau\,,\ \alpha_1 \sqcup \alpha_2 \sqcup \alpha_3)}$$

Eager Binding:

$$\frac{\Gamma \vdash_a e_0 : (\tau_0\,,\ \alpha_0) \quad \Gamma \cup \{i_v : \tau_0\} \vdash_a e_1 : (\tau_1\,,\ \alpha_1)}{\Gamma \vdash_a \mathit{let}\ i = e_0\ \mathit{in}\ e_1 : (\tau_1\,,\ \alpha_0 \sqcup \alpha_1)}$$

Recursive Binding:

$$\frac{\Gamma \cup \{i_v : \tau_0\} \vdash_a a : (\tau_0\,,\ trivial) \quad \Gamma \cup \{i_v : \tau_0\} \vdash_a e : (\tau_1\,,\ \alpha)}{\Gamma \vdash_a \mathit{letrec}\ i = a\ \mathit{in}\ e : (\tau_1\,,\ \alpha)}$$

Abstractions:

$$\frac{\Gamma \cup \{i : \tau_0\} \vdash_a e : (\tau_1\,,\ trivial)}{\Gamma \vdash_a \lambda_t i : \tau_0.e : (\tau_0 \to_t \tau_1\,,\ trivial)} \qquad \frac{\Gamma \cup \{i : \tau_0\} \vdash_a e : (\tau_1\,,\ serious)}{\Gamma \vdash_a \lambda i : \tau_0.e : (\tau_0 \to \tau_1\,,\ trivial)}$$

$$\frac{\Gamma \cup \{i_v : \tau_0\} \vdash_a e : (\tau_1\,,\ trivial)}{\Gamma \vdash_a \lambda_{tv} i : \tau_0.e : (\tau_0 \to_{tv} \tau_1\,,\ trivial)} \qquad \frac{\Gamma \cup \{i_v : \tau_0\} \vdash_a e : (\tau_1\,,\ serious)}{\Gamma \vdash_a \lambda_v i : \tau_0.e : (\tau_0 \to_v \tau_1\,,\ trivial)}$$

Applications:

$$\frac{\Gamma \vdash_a e_0 : (\tau_0 \to_{tv} \tau_1\,,\ \alpha) \quad \Gamma \vdash_a e_1 : (\tau_0\,,\ trivial)}{\Gamma \vdash_a e_0\ @_{tv}\ e_1 : (\tau_1\,,\ \alpha)}$$

$$\frac{\Gamma \vdash_a e_0 : (\tau_0 \to_t \tau_1\,,\ \alpha) \quad \Gamma \vdash_a e_1 : (\tau_0\,,\ serious)}{\Gamma \vdash_a e_0\ @_t\ e_1 : (\tau_1\,,\ \alpha)}$$

$$\frac{\Gamma \vdash_a e_0 : (\tau_0 \to_v \tau_1\,,\ \alpha) \quad \Gamma \vdash_a e_1 : (\tau_0\,,\ trivial)}{\Gamma \vdash_a e_0\ @_v\ e_1 : (\tau_1\,,\ serious)}$$

$$\frac{\Gamma \vdash_a e_0 : (\tau_0 \to \tau_1\,,\ \alpha) \quad \Gamma \vdash_a e_1 : (\tau_0\,,\ serious)}{\Gamma \vdash_a e_0\ @\ e_1 : (\tau_1\,,\ serious)}$$

Generalization:

$$\frac{\Gamma \vdash_a e : (\tau\,,\ trivial)}{\Gamma \vdash_a e : (\tau\,,\ serious)}$$

Fig. 8. Type-checking rules for the annotated λ-calculus

and call-by-value operational semantics for the non-annotated λ-calculus (λ-*exp*) of Figure 6. For either reduction relation, $e \Downarrow v$ (read "e halts at value v") denotes the reduction of some type-correct closed term $e \in \lambda\text{-}exp$ to a value v (*i.e.*, a constant or a λ-abstraction). Similarly, $e \Downarrow$ (read "e halts") denotes the reduction of some type-correct closed term e to an unspecified value.

Let $\mathcal{A} : \lambda\text{-}exp \rightarrow Ann\text{-}\lambda\text{-}exp$ denote an annotation-assigning function. $\mathcal{A}$ is considered to be correct if and only if it satisfies both of the following properties.

Property 1 (Soundness) *An annotation function $\mathcal{A}$ is* sound *iff for all type-correct closed terms $e \in \lambda\text{-}exp$, $\mathcal{A}[\![e]\!] = e' : (\tau\,,\, trivial)$ implies $e \Downarrow_n$ — that is, the evaluation of e terminates under call-by-name reduction.*

Property 2 (Consistency) *An annotation function $\mathcal{A}$ is* consistent *iff for all type-correct closed terms $e \in \lambda\text{-}exp$, $\mathcal{A}[\![e]\!] = e' : (\tau\,,\, \alpha)$ implies $\vdash_a e' : (\tau\,,\, \alpha)$.*

The process of assigning annotations can be automated using the techniques of abstract interpretation or of type inference. The abstract interpretation approach is summarized as follows. As a first step, the application sites of each abstraction are enumerated using a control-flow analysis [30]. The enumeration of application sites allows a straightforward generalization of Mycroft's ♭ termination analysis to our higher-order language [20]. The correctness of the termination analysis establishes the required soundness property (Property 1). Based on the results of the termination analysis, terms are assigned annotations via our type-annotation rules of Figure 8. At this step, the *Generalization* rule of Figure 8 needs to be used to establish the consistency requirement (Property 2). For example, if an abstraction is applied to both trivial and serious arguments, all trivial arguments are generalized to serious terms.

Note that the *Generalization* rule may lead to more than one correct assignment of annotations to a particular term. However, no semantic ambiguity results since the transformation $\mathcal{C}_a$ is correct for all correct annotation assignments $\mathcal{A}$ (see Proposition 3).

5.2 A transformation for the annotated λ-calculus

Figure 9 displays the extended transformation into continuation style. The transformation $\mathcal{C}_a[\![\cdot]\!]$ is used over both serious and trivial terms and dispatches to either $\mathcal{C}_a\langle\cdot\rangle$ or $\mathcal{C}_a\langle\!|\cdot|\!\rangle$.

- $\mathcal{C}_a\langle\cdot\rangle$ transforms trivial terms. No continuations are introduced in the transformed terms.
- $\mathcal{C}_a\langle\!|\cdot|\!\rangle$ transforms serious terms. Continuations are introduced in each transformed term.

Figure 9 displays the transformation $\mathcal{C}_a$ on types as well. $\mathcal{C}_a$ is extended to type assumptions by defining

$$\mathcal{C}_a[\![\{..., i : \tau, ..., i'_v : \tau', ...\}]\!] = \{..., i : \mathcal{C}_a\langle\!|\tau|\!\rangle, ..., i' : \mathcal{C}_a\langle\tau'\rangle, ...\}$$

The following proposition states the relationship between the types of annotated terms and the types of terms in the image of $\mathcal{C}_a$.

General Transformation:

$$\mathcal{C}_a[\![e : (\tau\,,\,\alpha)]\!] \;:\; \mathcal{C}_a[\![\tau]\!]$$
$$\mathcal{C}_a[\![e : (\tau\,,\,\mathit{trivial})]\!] = \lambda\kappa.\kappa\,\mathcal{C}_a\langle e\rangle$$
$$\mathcal{C}_a[\![e : (\tau\,,\,\mathit{serious})]\!] = \mathcal{C}_a(\!|e|\!)$$

Trivial Terms:

$$\mathcal{C}_a\langle e : (\tau\,,\,\mathit{trivial})\rangle \;:\; \mathcal{C}_a\langle\tau\rangle$$
$$\mathcal{C}_a\langle i_v\rangle = i$$
$$\mathcal{C}_a\langle \mathsf{op}_0\rangle = \mathsf{op}_0$$
$$\mathcal{C}_a\langle \mathsf{op}_1\, e_1\rangle = \mathsf{op}_1\,\mathcal{C}_a\langle e_1\rangle$$
$$\mathcal{C}_a\langle \mathsf{op}_2\, e_1\, e_2\rangle = \mathsf{op}_2\,\mathcal{C}_a\langle e_1\rangle\,\mathcal{C}_a\langle e_2\rangle$$
$$\mathcal{C}_a\langle \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2\rangle = \mathit{if}\ \mathcal{C}_a\langle e_0\rangle\ \mathit{then}\ \mathcal{C}_a\langle e_1\rangle\ \mathit{else}\ \mathcal{C}_a\langle e_2\rangle$$
$$\mathcal{C}_a\langle \mathit{let}\ i = e_0\ \mathit{in}\ e_1\rangle = \mathit{let}\ i = \mathcal{C}_a\langle e_0\rangle\ \mathit{in}\ \mathcal{C}_a\langle e_1\rangle$$
$$\mathcal{C}_a\langle \mathit{letrec}\ i = a\ \mathit{in}\ e\rangle = \mathit{letrec}\ i = \mathcal{C}_a\langle a\rangle\ \mathit{in}\ \mathcal{C}_a\langle e\rangle$$
$$\mathcal{C}_a\langle \lambda_w i : \tau.e\rangle = \lambda i : \mathcal{C}_a\langle\tau\rangle.\mathcal{C}_a\langle e\rangle$$
$$\mathcal{C}_a\langle \lambda_t i : \tau.e\rangle = \lambda i : \mathcal{C}_a(\!|\tau|\!).\mathcal{C}_a\langle e\rangle$$
$$\mathcal{C}_a\langle \lambda_v i : \tau.e\rangle = \lambda i : \mathcal{C}_a\langle\tau\rangle.\mathcal{C}_a(\!|e|\!)$$
$$\mathcal{C}_a\langle \lambda i : \tau.e\rangle = \lambda i : \mathcal{C}_a(\!|\tau|\!).\mathcal{C}_a(\!|e|\!)$$
$$\mathcal{C}_a\langle e_0 \mathbin{@_w} e_1\rangle = \mathcal{C}_a\langle e_0\rangle\,\mathcal{C}_a\langle e_1\rangle$$
$$\mathcal{C}_a\langle e_0 \mathbin{@_t} e_1\rangle = \mathcal{C}_a\langle e_0\rangle\,\mathcal{C}_a(\!|e_1|\!)$$

Serious Terms:

$$\mathcal{C}_a(\!|e : (\tau\,,\,\mathit{serious})|\!) \;:\; \mathcal{C}_a(\!|\tau|\!)$$
$$\mathcal{C}_a(\!|i|\!) = i$$
$$\mathcal{C}_a(\!|\mathsf{op}_1\, e_1|\!) = \lambda\kappa.\mathcal{C}_a[\![e_1]\!]\,(\lambda v_1.\kappa\,(\mathsf{op}_1\, v_1))$$
$$\mathcal{C}_a(\!|\mathsf{op}_2\, e_1\, e_2|\!) = \lambda\kappa.\mathcal{C}_a[\![e_1]\!]\,(\lambda v_1.\mathcal{C}_a[\![e_2]\!]\,(\lambda v_2.\kappa\,(\mathsf{op}_2\, v_1\, v_2)))$$
$$\mathcal{C}_a(\!|\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2|\!) = \lambda\kappa.\mathcal{C}_a[\![e_0]\!]\,(\lambda v_0.\mathit{if}\ v_0\ \mathit{then}\ \mathcal{C}_a[\![e_1]\!]\,\kappa\ \mathit{else}\ \mathcal{C}_a[\![e_2]\!]\,\kappa)$$
$$\mathcal{C}_a(\!|\mathit{let}\ i = e_0\ \mathit{in}\ e_1|\!) = \lambda\kappa.\mathcal{C}_a[\![e_0]\!]\,(\lambda i.\mathcal{C}_a[\![e_1]\!]\,\kappa)$$
$$\mathcal{C}_a(\!|\mathit{letrec}\ i = a\ \mathit{in}\ e|\!) = \lambda\kappa.\mathit{letrec}\ i = \mathcal{C}_a\langle a\rangle\ \mathit{in}\ \mathcal{C}_a[\![e]\!]\kappa$$
$$\mathcal{C}_a(\!|e_0 \mathbin{@_w} e_1|\!) = \lambda\kappa.\mathcal{C}_a[\![e_0]\!]\,(\lambda v_0.\kappa\,(v_0\,\mathcal{C}_a\langle e_1\rangle))$$
$$\mathcal{C}_a(\!|e_0 \mathbin{@_t} e_1|\!) = \lambda\kappa.\mathcal{C}_a[\![e_0]\!]\,(\lambda v_0.\kappa\,(v_0\,\mathcal{C}_a(\!|e_1|\!)))$$
$$\mathcal{C}_a(\!|e_0 \mathbin{@_v} e_1|\!) = \lambda\kappa.\mathcal{C}_a[\![e_0]\!]\,(\lambda v_0.(v_0\,\mathcal{C}_a\langle e_1\rangle)\,\kappa)$$
$$\mathcal{C}_a(\!|e_0 \mathbin{@} e_1|\!) = \lambda\kappa.\mathcal{C}_a[\![e_0]\!]\,(\lambda v_0.(v_0\,\mathcal{C}_a(\!|e_1|\!))\,\kappa)$$

Types:

$$\mathcal{C}_a[\![\tau]\!] = \mathcal{C}_a(\!|\tau|\!)$$
$$\mathcal{C}_a(\!|\tau|\!) = (\mathcal{C}_a\langle\tau\rangle \to \mathit{Ans}) \to \mathit{Ans}$$
$$\mathcal{C}_a\langle\iota\rangle = \iota$$

$$\mathcal{C}_a\langle\tau_0 \to_w \tau_1\rangle = \mathcal{C}_a\langle\tau_0\rangle \to \mathcal{C}_a\langle\tau_1\rangle$$
$$\mathcal{C}_a\langle\tau_0 \to_v \tau_1\rangle = \mathcal{C}_a\langle\tau_0\rangle \to \mathcal{C}_a(\!|\tau_1|\!)$$
$$\mathcal{C}_a\langle\tau_0 \to_t \tau_1\rangle = \mathcal{C}_a(\!|\tau_0|\!) \to \mathcal{C}_a\langle\tau_1\rangle$$
$$\mathcal{C}_a\langle\tau_0 \to \tau_1\rangle = \mathcal{C}_a(\!|\tau_0|\!) \to \mathcal{C}_a(\!|\tau_1|\!)$$

Fig. 9. Transformation of annotated λ-terms into continuation style

Proposition 2.

- *If* $\Gamma \vdash_a e : (\tau, \alpha)$ *then* $\mathcal{C}_a[\![\Gamma]\!] \vdash \mathcal{C}_a[\![e]\!] : \mathcal{C}_a[\![\tau]\!]$.
- *If* $\Gamma \vdash_a e : (\tau, \mathit{serious})$ *then* $\mathcal{C}_a[\![\Gamma]\!] \vdash \mathcal{C}_a(\![e]\!) : \mathcal{C}_a(\![\tau]\!)$.
- *If* $\Gamma \vdash_a e : (\tau, \mathit{trivial})$ *then* $\mathcal{C}_a[\![\Gamma]\!] \vdash \mathcal{C}_a\langle e \rangle : \mathcal{C}_a\langle \tau \rangle$.

The correctness of $\mathcal{C}_a$ is stated as follows. (The notation "$e \Downarrow_n r$" is defined in Section 5.1.)

Proposition 3. *For all type-correct closed terms e of base type and for all correct annotation-assigning functions* $\mathcal{A}$,

$$e \Downarrow_n r \iff (\mathcal{C}_a \circ \mathcal{A}[\![e]\!])(\lambda i : \iota.i) \Downarrow_n r \iff (\mathcal{C}_a \circ \mathcal{A}[\![e]\!])(\lambda i : \iota.i) \Downarrow_v r$$

Proof. See [10].

5.3 Assessment

Restricting the new transformation $\mathcal{C}_a$ (see Figure 9) to call-by-name serious λ-terms yields the call-by-name transformation into continuation style (see Figure 2). Conversely, restricting $\mathcal{C}_a$ to call-by-value trivial λ-terms yields the identity transformation — no continuations are needed at all (*e.g.*, the denotational semantics of a strongly-normalizing language or of the language of Figure 3 without the "while" statement). Thus, $\mathcal{C}_a$ generalizes both the call-by-name transformation into continuation style and the identity transformation.

6 Some Properties of the Direct-Style Definition of the Simple Imperative Language (Figure 4)

Property 3 $\mathcal{Z}[\![\text{Program}]\!]$, $\mathcal{C}[\![\text{Command}]\!]$, *and* $\mathcal{E}[\![\text{Expression}]\!]$ *are trivial and call-by-value.*

Proof. Each is call-by-value because it is not possible for an argument expression of type *Env* to diverge. Each is trivial because a λ-abstraction (a value) is always returned.

Property 4 *The function type Com is call-by-value and serious.*

Proof. Due to the eager binding of the *let* construct, each command is passed a reduced store value. Therefore, the function type can be classified as call-by-value. Commands are serious because looping may occur in the *while* construct (this was accounted for by the lifting of the codomain of *Com*).

Property 5 *The function type Exp is call-by-value and trivial.*

Proof. Due to call-by-value property of commands, each expression is passed a reduced store value. Therefore, the function type can be classified as call-by-value. No expression contains components which may loop. Therefore expressions are trivial.

Property 6 *Proc and Fun are call-by-value and trivial.*

Proof. The arguments to procedures and functions originate from the evaluation of expressions which can never loop. Therefore, the function spaces are classified as call-by-value. They are trivial because they both return λ-abstractions.

Figure 10 presents an annotated representation of the direct semantics of Figure 4.[9] Any reasonable implementation of $\mathcal{A}$ as outlined in Section 5.1 would assign such annotations automatically. Let us now transform the annotated terms into continuation style.

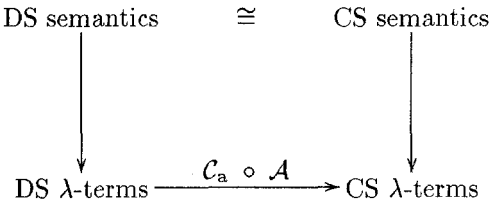
7 Transforming the Representation of a Direct-Style Definition

Fact 1 *Transforming the annotated λ-representation of the direct semantics in Figure 10 into continuation style does yield the λ-representation of the continuation semantics in Figure 5, after administrative reductions.*

Further, we now have the ability to specify *any kind* of continuation semantics. For example, we could classify *Exp* to be serious (this would happen if recursive functions were allowed in the simple imperative language). This classification suffices to construct a continuation semantics where the valuation functions for both commands and expressions are in continuation style, automatically.

8 Conclusion, Issues, and Future Work

We have tried to contribute to the study of the relation between direct and continuation semantics [26] by connecting it to the transformation of λ-terms into continuation style. To this end, we have described how to construct the representation of a realistic continuation semantics automatically, given the representation of a direct semantics. (Again, by "realistic", we mean "that could have been written by hand".) The situation is summarized by the diagram below.



[9] Note that non-termination — represented explicitly via lifting (*e.g.*, $Store_\perp$) in the semantics definition of Figure 4 — is captured implicitly in the reduction properties of terms in Figure 10. This was accounted for in Property 4: *Com* has a serious function type.

Valuation Functions:

$$\begin{aligned}
\mathcal{Z}[\![\text{Program}]\!] &: \mathit{Env} \to_{tv} \mathit{Com} \\
\mathcal{C}[\![\text{Command}]\!] &: \mathit{Env} \to_{tv} \mathit{Com} \\
\mathcal{E}[\![\text{Expression}]\!] &: \mathit{Env} \to_{tv} \mathit{Exp} \\
\mathcal{N}[\![\text{Numeral}]\!] &: \mathit{Nat} \\
\mathcal{L}[\![\text{Location}]\!] &: \mathit{Loc}
\end{aligned}$$

Semantic Domains:

$$\begin{aligned}
\mathit{Com} &= \mathit{Store} \to_{v} \mathit{Store} \\
\mathit{Exp} &= \mathit{Store} \to_{tv} \mathit{Nat} \\
\mathit{Proc} &= \mathit{Nat} \to_{tv} \mathit{Com} \\
\mathit{Fun} &= \mathit{Nat} \to_{tv} \mathit{Exp}
\end{aligned}$$

Programs:

$$\mathcal{Z}[\![\text{proc } p\ (m) = c \text{ in } z]\!] = \lambda_{tv}\rho.\lambda_v\sigma.\mathcal{Z}[\![z]\!]\,@_{tv}\,(\mathsf{ext}\ \rho_v\ p\ (\lambda_{tv} i.\mathcal{C}[\![c]\!]\,@_{tv}\,(\mathsf{ext}\ \rho_v\ m\ i_v)))\,@_v\,\sigma_v$$

$$\mathcal{Z}[\![\text{fun } f\ (m) = e \text{ in } z]\!] = \lambda_{tv}\rho.\lambda_v\sigma.\mathcal{Z}[\![z]\!]\,@_{tv}\,(\mathsf{ext}\ \rho_v\ f\ (\lambda_{tv} i.\mathcal{E}[\![e]\!]\,@_{tv}\,(\mathsf{ext}\ \rho_v\ m\ i_v)))\,@_v\,\sigma_v$$

$$\mathcal{Z}[\![c.]\!] = \lambda_{tv}\rho.\lambda_v\sigma.\mathcal{C}[\![c]\!]\,@_{tv}\,\rho_v\,@_v\,\sigma_v$$

Commands:

$$\mathcal{C}[\![\text{skip}]\!] = \lambda_{tv}\rho.\lambda_v\sigma.\sigma_v$$

$$\mathcal{C}[\![c_1\ ;\ c_2]\!] = \lambda_{tv}\rho.\lambda_v\sigma.\mathit{let}_v\ \sigma' = \mathcal{C}[\![c_1]\!]\,@_{tv}\,\rho_v\,@_v\,\sigma_v\ \mathit{in}\ \mathcal{C}[\![c_2]\!]\,@_{tv}\,\rho_v\,@_v\,\sigma'_v$$

$$\mathcal{C}[\![l := e]\!] = \lambda_{tv}\rho.\lambda_v\sigma.\mathsf{upd}\ \sigma_v\ \mathcal{L}[\![l]\!]\ (\mathcal{E}[\![e]\!]\,@_{tv}\,\rho_v\,@_{tv}\,\sigma_v)$$

$$\begin{aligned}
\mathcal{C}[\![\text{if } e \text{ then } c_1 \text{ else } c_2]\!] = \lambda_{tv}\rho.\lambda_v\sigma.\ &\mathit{if}\quad \mathsf{iszero?}\ (\mathcal{E}[\![e]\!]\,@_{tv}\,\rho_v\,@_{tv}\,\sigma_v) \\
&\mathit{then}\ (\mathcal{C}[\![c_1]\!]\,@_{tv}\,\rho_v\,@_v\,\sigma_v) \\
&\mathit{else}\ \ (\mathcal{C}[\![c_2]\!]\,@_{tv}\,\rho_v\,@_v\,\sigma_v)
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\![\text{while } e \text{ do } c]\!] = \lambda_{tv}\rho.\lambda_v\sigma.\ &\mathit{letrec}_v\ w = \lambda_v\sigma.\ \mathit{if}\quad \mathsf{iszero?}\ (\mathcal{E}[\![e]\!]\,@_{tv}\,\rho_v\,@_{tv}\,\sigma_v) \\
&\qquad\qquad \mathit{then}\ \ \mathit{let}_v\ \sigma' = \mathcal{C}[\![c]\!]\,@_{tv}\,\rho_v\,@_v\,\sigma_v \\
&\qquad\qquad\qquad \mathit{in}\ w_v\,@_v\,\sigma'_v \\
&\qquad\qquad \mathit{else}\ \ \sigma_v \\
&\mathit{in}\ w_v\,@_v\,\sigma_v
\end{aligned}$$

$$\mathcal{C}[\![\text{call } p\ (e)]\!] = \lambda_{tv}\rho.\lambda_v\sigma.(\mathsf{lookup}\ \rho_v\ p)\,@_{tv}\,(\mathcal{E}[\![e]\!]\,@_{tv}\,\rho_v\,@_{tv}\,\sigma_v)\,@_v\,\sigma_v$$

Expressions:

$$\mathcal{E}[\![n]\!] = \lambda_{tv}\rho.\lambda_{tv}\sigma.\mathcal{N}[\![n]\!]$$

$$\mathcal{E}[\![m]\!] = \lambda_{tv}\rho.\lambda_{tv}\sigma.\mathsf{lookup}\ \rho_v\ m$$

$$\mathcal{E}[\![\text{succ } e]\!] = \lambda_{tv}\rho.\lambda_{tv}\sigma.\mathsf{succ}\ (\mathcal{E}[\![e]\!]\,@_{tv}\,\rho_v\,@_{tv}\,\sigma_v)$$

$$\mathcal{E}[\![\text{pred } e]\!] = \lambda_{tv}\rho.\lambda_{tv}\sigma.\mathsf{pred}\ (\mathcal{E}[\![e]\!]\,@_{tv}\,\rho_v\,@_{tv}\,\sigma_v)$$

$$\mathcal{E}[\![\text{deref } l]\!] = \lambda_{tv}\rho.\lambda_{tv}\sigma.\mathsf{fetch}\ \sigma_v\ \mathcal{L}[\![l]\!]$$

$$\mathcal{E}[\![\text{apply } f\ (e)]\!] = \lambda_{tv}\rho.\lambda_{tv}\sigma.(\mathsf{lookup}\ \rho_v\ f)\,@_{tv}\,(\mathcal{E}[\![e]\!]\,@_{tv}\,\rho_v\,@_{tv}\,\sigma_v)\,@_{tv}\,\sigma_v$$

Fig. 10. Annotated representation of the direct semantics in Figure 4

Based on the annotations produced by $\mathcal{A}$, the transformation $\mathcal{C}_a$ introduces just enough continuations to preserve call-by-name meaning under both call-by-name and call-by-value reduction. $\mathcal{C}_a$ generalizes both the call-by-name continuation transformation (should all terms be serious) and the identity transformation (should all terms be trivial).

8.1 A shortcoming?

One might criticize one shortcoming of this approach: it only produces the representation of a continuation semantics, not the continuation semantics itself. One answer to this criticism goes as follows.

Why would one want a continuation semantics when one already has a brave and honest direct semantics?[10] Not for the love of mathematics alone, but for implementation purposes [16, 18]! But then one does not need the continuation semantics, but its representation — which is precisely what our new transformation produces automatically. Therefore our approach enables the language developer to stay with one mathematical model — the direct semantics — and to derive the continuation semantics as part of the implementation work.

8.2 An alternative?

One could transform the direct semantics into continuation style and then simplify the result into a manageable continuation semantics. We believe that our approach is more natural since most of the work operates over the original direct semantics and the rest is automatic. (A worthwhile property considering how counter-intuitive continuation-style specifications may look.)

8.3 Applications

Semantics-directed compiler-generation systems often work on continuation semantics [13, 14, 16, 18], thus forcing one to write a continuation semantics and to prove its congruence with the direct one. Our new transformation allows one to produce the representation of a continuation semantics automatically.

Partial evaluators work better on continuation-passing programs, but again not all continuations are always necessary [2, 3]. Our extended transformation into continuation style makes it possible to reduce the occurrences of continuations in a source program. In addition, it also enables partial evaluation of call-by-name programs (after evaluation-order analysis — be it for strictness or termination) with a regular partial evaluator for call-by-value programs.

[10] Of course, the situation is different if the source language includes some form of jump. But then one has no direct semantics and thus *starts* with a continuation semantics.

8.4 Variations

Denotational definitions are written in various fashions. We briefly mention how the present work can be adapted to other fashions.

Partial functions are often used in place of total functions and lifted domains when modeling non-terminating computations [23, 34]. Our explanation of totality and partiality in terms of trivial and serious functions naturally applies to denotational specifications based on partial functions.

Strict functions are often used to model the strictness properties associated with eager (*i.e.*, call-by-value) functions [23, 28]. For simplicity of presentation, we have expressed strictness properties using *let* constructs only. Just as *let* expressions are represented using eager binding constructs, strict functions are represented using eager applications. Thus, a transformation of an annotated language including both eager and normal-order application generalizes both the call-by-value and the call-by-name transformation into continuation style. We have presented a formalization of such a mixed transformation elsewhere [5].

Continuation semantics of imperative languages often express the meaning of commands as "continuation transformers" [28, 34]. Specifically, the functionality of *Com* is given as

$$(Store \to Ans) \to Store \to Ans.$$

It is very simple to specify a transformation into continuation style that "puts continuations first", as in Fischer's original transformation [7, 27]. Such a transformation would naturally yield the functionality above.

Finally, our work has relied on denotational definitions being stated using a simply-typed meta-language. This meta-language is sufficient for defining simple imperative languages and simply-typed languages such as Algol 60, Pascal, and PCF. We are currently investigating how the results presented here can be extended to a meta-language with recursive types. This would be necessary for defining untyped languages such as Scheme.

8.5 Generalization

The work presented here can be generalized to other styles than continuation style. Alternatively, one could define a core meta-language and parameterize it with the style of the interpretation. This approach is reminiscent of Mosses and Watt's Action Semantics [19, 35], of the Nielsons's two-level meta-language [23], and of Moggi's computational λ-calculus [17]. We investigate it elsewhere [11].

8.6 Transforming the representation of a continuation semantics into direct style

The transformation from continuation style to direct style has been investigated recently [4, 6, 12, 27], and enables one to transform the representation of a continuation semantics into direct style. Of course, we can only produce the representation of a direct semantics from a continuation semantics where continuations

are second-class [31] — for example, we could not produce a direct semantics for a language with jumps [32], not without adding some kind of control operator to the λ-calculus [6]. Syntactic conditions over a continuation-passing λ-term to ensure that continuations are second-class can be found elsewhere [4]. Overall we leave this transformation for future work.

8.7 Continuation style and evaluation-order independence

It is interesting to compare the structure of the terms produced by our transformation $\mathcal{C}_a$ with the structure of the terms produced by *e.g.*, Plotkin's continuation-style transformations [24]. In addition to satisfying his *Simulation*, *Indifference*, and *Translation* theorems, Plotkin's continuation-style terms have two additional properties that are often utilized for implementation purposes:

- all function calls are in "tail" position;[11]
- all intermediate values are given names.

In contrast, $\mathcal{C}_a$ inserts just enough continuations to preserve call-by-name meaning under both call-by-name and call-by-value reduction. Thus, $\mathcal{C}_a$ satisfies Plotkin's *Simulation*, *Indifference*, and *Translation* theorems [10], but the two additional properties above are lost because some applications may be trivial.

- Trivial applications may occur as function arguments — not a "tail" position.
- Trivial applications yield intermediate values that are not named — since trivial functions are not passed any continuation.

In other words, our transformation $\mathcal{C}_a$ does not produce "Continuation-Passing Style" terms (!) but it does produce terms that are independent of the evaluation order.

Acknowledgements

We are grateful to Andrzej Filinski, Julia Lawall, Karoline Malmkjær, David Schmidt, and the referees for their comments, to Frank Pfenning and Kirsten Solberg for their interest and time, and to Hanne Riis Nielson for a key comment about our style of annotation. Thanks are also due to Patrick Cousot for connecting our termination analysis with Alan Mycroft's $\sharp$ and $\flat$ analyses. The diagram of Section 8 was drawn using Kristoffer Rose's XY-pic package.

References

1. William Clinger, editor. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, San Francisco, California, June 1992. ACM Press.

[11] As characterized by Meyer and Wand [15], "[Continuation-style] terms are tail-recursive: no argument is an application."

2. Charles Consel and Olivier Danvy. For a better support of static data flow. In John Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 496–519, Cambridge, Massachusetts, August 1991.
3. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Graham [8], pages 493–501.
4. Olivier Danvy. Back to direct style. In Bernd Krieg-Brückner, editor, *Proceedings of the Fourth European Symposium on Programming*, number 582 in Lecture Notes in Computer Science, pages 130–150, Rennes, France, February 1992. Extended version to appear in Science of Computer Programming.
5. Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.
6. Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In Clinger [1], pages 299–310.
7. Michael J. Fischer. Lambda calculus schemata. In Talcott [33]. An earlier version appeared in an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.
8. Susan L. Graham, editor. *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993. ACM Press.
9. Bob Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Talcott [33].
10. John Hatcliff. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, USA, March 1994. Forthcoming.
11. John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press. To appear.
12. Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In Graham [8], pages 124–136.
13. Peter Lee. *Realistic Compiler Generation*. MIT Press, 1989.
14. Peter Lee and Uwe Pleban. On the use of LISP in implementing denotational semantics. In William L. Scherlis and John H. Williams, editors, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 233–248, Cambridge, Massachusetts, August 1986.
15. Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, June 1985.
16. Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, and John Wiley, New York, 1976.
17. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
18. Margaret Montenyohl and Mitchell Wand. Correct flow analysis in continuation semantics. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 204–218, San Diego, California, January 1988.
19. Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

20. Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In Bernard Robinet, editor, *Proceedings of the Fourth International Symposium on Programming*, number 83 in Lecture Notes in Computer Science, pages 269–281, Paris, France, April 1980.
21. Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1981.
22. Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988. Special issue on ESOP'86, the First European Symposium on Programming, Saarbrücken, March 17-19, 1986.
23. Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.
24. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
25. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, 1972.
26. John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *2nd Colloquium on Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages 141–156, Saarbrücken, West Germany, July 1974.
27. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Clinger [1], pages 288–298.
28. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.
29. Peter Sestoft. Replacing function parameters by global variables. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 39–53, London, England, September 1989. ACM Press.
30. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, CMU, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.
31. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.
32. 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.
33. Carolyn L. Talcott, editor. *Special issue on continuations*, LISP and Symbolic Computation, Vol. 6, Nos. 3/4. Kluwer Academic Publishers, 1993.
34. Robert D. Tennent. *Semantics of Programming Languages.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1991.
35. David Watt. *Programming Languages Concepts and Paradigms.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.