Back to Direct Style II:
First-Class Continuations

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

Julia L. Lawall
Department of Computer Science
Indiana University [†]
jll@cs.indiana.edu

Abstract

We continue to investigate the direct-style transformation by extending it to programs requiring `call-with-current-continuation` (a.k.a. `call/cc`). The direct style (DS) and the continuation-passing style (CPS) transformations form a Galois connection. This pair of functions has a place in the programmer's toolbox — yet we are not aware of the existence of any other DS transformer.

Starting from our DS transformer towards pure, call-by-value functional terms (Scheme), we extend it with a counting analysis to detect non-canonical occurrences of a continuation. The declaration of such a continuation is translated into a `call/cc` and its application into the application of the corresponding first-class continuation.

We also present staged versions of the DS and of the CPS transformations, where administrative reductions are separated from the actual translation, and where the actual translations are carried out by local, structure-preserving rewriting rules. These staged transformations are used to prove the Galois connection.

Together, the CPS and the DS transformations enlarge the class of programs that can be manipulated on a semantic basis. We illustrate this point with partial evaluation, by specializing a Scheme program with respect to a static part of its input. The program uses coroutines. This illustration achieves a first: a static coroutine is executed statically and its computational content is inlined in the residual program.

1 Introduction

Functional programming folklore has it that control operators such as `call/cc` are unnecessary because their effect can be simulated by continuation-passing style (CPS) [31]. On the other hand CPS forces one to write programs in an extraordinarily contrived way. Fortunately, the CPS transformation automatically maps programs (with or without control operators) into purely functional programs.

Our goal is to reverse this process, mapping CPS terms back into direct style (DS). However not all CPS expressions correspond to pure functional terms. Those that use the continuation non-canonically require `call/cc`. Thus our goal is to map arbitrary CPS programs into DS programs that can use `call/cc` to account for first-class continuations.

Of course, we assume that the evaluation of DS terms follows the same strategy (*e.g.*, left-to-right call-by-value) as the one assumed by the CPS transformation. Similarly, any CPS term should be compatible with this evaluation strategy.

1.1 Example: a coroutine package in Scheme

Scheme offers first-class continuations via `call/cc`. Because continuations are reified as procedures, they are invoked by procedure application. For clarity, however, in this paper we will consistently use the explicit keyword `throw` to tag such invocations.[1]

Let us consider a coroutine package for Scheme programs. The following procedure swaps coroutines.

```
(define resume
  (lambda (c)
    (call/cc (lambda (k)
               (c (lambda (r) (throw k r)))))))
```

Let us transform this procedure into CPS.

```
(define resume-c
  (lambda (c k)
    (c (lambda (r _) (k r))
       (lambda (v) (k v)))))
```

Two things stand out about this CPS procedure.

- `resume-c` duplicates its continuation. That is, `k` occurs twice in the body of `resume-c`.
- `(lambda (r _) (k r))` does not apply its continuation. That is, `_` does not occur in the body of this λ-abstraction.

These two non-canonical occurrences fundamentally reflect a control operation. Based on this observation, we describe a counting analysis that detects such occurrences in a CPS term. This information tells us where to insert `call/cc` when a CPS term is transformed into DS.

---

[*] Manhattan, Kansas 66506, USA. Part of this work was supported by NSF under grant CCR-9102625.

[†] Bloomington, Indiana 47405, USA. Part of this work was supported by NSF under grant CCR-9000597. This work was initiated during two visits to KSU in September and November 1991.

To appear in the proceedings of the 1992 ACM Symposium on Lisp and Functional Programming, June 1992, San Francisco, California. Also available as Technical Report CIS-92-1, Kansas State University.

[1] This syntactic sugar can be obtained as a Scheme macro [2]:

```
(define-syntax throw
  (syntax-rules ()
    [(throw k v) (k v)]))
```

1.2 Applications

1.2.1 An Interpreter for Scheme 84 (revisited)

In the proceedings of LFP'84 [17, Fig. 1, p. 295], Haynes, Friedman, and Wand present a CPS interpreter for Scheme 84. Our DS transformer maps this interpreter to the natural direct-style specification of Scheme. The result (omitted here for lack of space) uses `call/cc` to implement `call/cc`. Otherwise it looks like any other meta-circular Scheme interpreter. Therefore, the only reason to write the original interpreter in CPS was to specify `call/cc`.

1.2.2 Meta*-continuation passing style

Iterating the CPS transformation over a CPS program yields the "meta-continuation passing style" (MCPS) described by Danvy and Filinski in the proceedings of the LFP'90 [10]. Just as a CPS interpreter can be used to specify `call/cc`, a MCPS interpreter can be used to specify control operators such as `shift` and `reset` while staying in the framework of CPS. The DS transformation maps any pure M^{n+1}CPS into M^{n}CPS.

1.2.3 Semantics-based program manipulation

Semantics-based program manipulation (*e.g.*, compilation or partial evaluation) performs better on CPS programs. The flow analysis of a CPS program by abstract interpretation naturally yields more precise results (Meet Over all Paths instead of a Least Fixed Point solution, even with a non-distributive analysis) [24, 6]. Therefore it is beneficial to first transform the source program into CPS. However, it makes sense to transform the resulting program back to DS, since the user might expect an answer in the form of the original, as in the case of a source to source transformation.

Augmenting the DS transformation algorithm to produce programs with `call/cc` widens the class of higher-order programs that can be manipulated on a semantic basis.

1.3 Overview

The rest of this paper is organized as follows. Section 2 reviews our starting point: the DS transformation of restricted CPS terms into pure call-by-value DS terms. After extending the syntax of CPS terms in Section 3, we describe a counting analysis in Section 4.1. Section 4.3 presents a new DS transformer of CPS terms into call-by-value DS terms. The resulting program may use `call/cc`. In Section 5, we investigate some properties of the DS and the CPS transformations. Section 6 illustrates how our DS transformer can play a rôle in semantics-based program manipulation. After a comparison with related work in Section 7, Section 8 concludes. Finally, in Section 9 we present issues for future work.

2 Pure Direct Style Transformation

This section reviews our starting point: the DS transformation of CPS terms into pure, call-by-value functional terms [9]. Section 2.1 addresses CPS terms that yield pure functional DS terms. Only the current continuation can be applied in such CPS terms. We capture this constraint as an attribute in the BNF. Section 2.2 specifies the BNF of DS terms. Section 2.3 presents the DS transformation as a rewrite system.

2.1 BNF of pure CPS terms

We consider the λ-calculus applied to the usual first-order constants (booleans, 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 list operators (*i.e.*, we do *not* have continuation-passing primitive operators [31]). Procedures are n-ary. Continuation parameters occur last.

Let us give the BNF of the terms produced by the CPS transformation of Scheme programs [11]. Much as Reynolds [29], we distinguish between "serious" and "trivial" expressions. A serious expression s^κ is evaluated in the scope of a continuation κ and a trivial expression t^κ denotes a value that is passed to a continuation κ. Pure CPS may be characterized by the following *second-class constraint:*

Definition 1 (Second-class constraint)

1. *Only the current continuation identifier can be applied.*
2. *The current continuation identifier must occur in every serious subexpression of each expression.*

We index each non-terminal with the current continuation κ as an inherited attribute to ensure that the second-class constraint is satisfied.

A CPS λ-term in an empty context, e, is defined by the following attribute grammar.

$e \in$ CExp — domain of CPS expressions
$s \in$ CSer — domain of serious expressions
$t \in$ CTriv — domain of trivial expressions
$l \in$ CLam — domain of λ-abstractions
$op \in$ Opr — domain of primitive operators
$c \in$ Cst — domain of first-order constant values
$i, v, k \in$ Ide — domain of identifiers

$$\begin{array}{lcll}
e & ::= & @\,(\lambda\, k\,.\, s^k)\,(\lambda\, v\,.\, v) & \\
s^\kappa & ::= & @\, k\, t^\kappa & \textit{where } k = \kappa \\
 & \mid & @\, t_0^\kappa\,(t_1^\kappa,\, ...,\, t_n^\kappa,\, \lambda\, v\,.\, s^\kappa) & \\
 & & & \textit{where } v \neq \kappa \textit{ and } v \textit{ occurs linearly in } s \\
 & \mid & \text{let } k = \lambda\, v\,.\, s_4^\kappa \text{ in cond}(t_1^k,\, s_2^k,\, s_3^k) & \\
 & & & \textit{where } v \neq \kappa \\
 & & & \textit{and } v \textit{ occurs linearly in } s_4 \\
 & & & \textit{and } k = \kappa \vee \kappa \notin FV(\text{cond}(t_2^k,\, s_3^k,\, s_4^k)) \\
 & \mid & \text{let } (i_1,\, ...,\, i_n) = (t_1^\kappa,\, ...,\, t_n^\kappa) \text{ in } s^\kappa & \\
 & & & \textit{where } \kappa \notin \{i_1,\, ...,\, i_n\} \\
 & \mid & \text{letrec } (i_1,\, ...,\, i_n) = (l_1^\kappa,\, ...,\, l_n^\kappa) \text{ in } s^\kappa & \\
 & & & \textit{where } \kappa \notin \{i_1,\, ...,\, i_n\} \\
t^\kappa & ::= & c & \\
 & \mid & i & \textit{where } i \neq \kappa \\
 & \mid & l^\kappa & \\
 & \mid & op\,(t_1^\kappa,\, ...,\, t_m^\kappa) & \\
l^\kappa & ::= & \lambda\,(i_1,\, ...,\, i_n,\, k)\,.\, s^k & \\
 & & & \textit{where } \kappa \notin FV(\lambda\,(i_1,\, ...,\, i_n,\, k)\,.\, s^k)
\end{array}$$

where for any λ-term e, FV(e) denotes the set of variables that are free in e.

Each non-terminal is indexed with an identifier κ denoting the current continuation. The second-class constraint is embodied in the restriction on the production $s^\kappa ::= @\, k\, t^\kappa$ that *only the current continuation can be applied.*

$$\frac{\nu \vdash t}{\nu \vdash @\,k\,t} \qquad \frac{\otimes_{n+2}(\nu \vdash t_0, \quad \dots, \quad \nu \vdash t_n, \quad v \neq \nu \;\;\wedge\;\; \nu \vdash s)}{\nu \vdash @\,t_0\,(t_1, \dots, t_n, \lambda\, v\,.\,s)}$$

$$\frac{\otimes_2(v \neq \nu \;\;\wedge\;\; \nu \vdash s_4, \quad \nu \vdash t_1) \quad \nu \notin \mathrm{FV}(s_2) \quad \nu \notin \mathrm{FV}(s_3)}{\nu \vdash \mathrm{let}\; k = \lambda\, v\,.\,s_4 \;\mathrm{in}\; \mathrm{cond}(t_1, s_2, s_3)}$$

$$\frac{\otimes_{n+1}(\nu \vdash t_1, \quad \dots, \quad \nu \vdash t_n, \quad \nu \notin \{i_1, \dots, i_n\} \;\;\wedge\;\; \nu \vdash s)}{\nu \vdash \mathrm{let}\;(i_1, \dots, i_n) = (t_1, \dots, t_n)\;\mathrm{in}\; s}$$

$$\frac{\nu \notin \{i_1, \dots, i_n\} \quad \forall j \in \{1, \dots, n\},\; \nu \notin \mathrm{FV}(l_j) \quad \nu \vdash s}{\nu \vdash \mathrm{letrec}\;(i_1, \dots, i_n) = (l_1, \dots, l_n)\;\mathrm{in}\; s}$$

$$\frac{v = \nu}{\nu \vdash v} \qquad \frac{\otimes_m(\nu \vdash t_1, \quad \dots, \quad \nu \vdash t_m)}{\nu \vdash op\,(t_1, \dots, t_m)}$$

Figure 1: Linearity conditions over continuations

$$\begin{array}{rcl}
[\![@\,k\,t]\!] & = & [t] \\
[\![@\,t_0\,(t_1, \dots, t_n, \lambda\, v\,.\,s)]\!] & = & @\,(\lambda\, v\,.\,[\![s]\!])\,(@\,[t_0]\,([t_1], \dots, [t_n])) \\
[\![\mathrm{let}\; k' = \lambda\, v\,.\,s_4 \;\mathrm{in}\;\mathrm{cond}(t_1, s_2, s_3)]\!] & = & @\,(\lambda\, v\,.\,[\![s_4]\!])\,(\mathrm{cond}([t_1], [\![s_2]\!], [\![s_3]\!])) \\
[\![\mathrm{let}\;(i_1, \dots, i_n) = (t_1, \dots, t_n)\;\mathrm{in}\; s]\!] & = & \mathrm{let}\;(i_1, \dots, i_n) = ([t_1], \dots, [t_n])\;\mathrm{in}\;[\![s]\!] \\
[\![\mathrm{letrec}\;(i_1, \dots, i_n) = (l_1, \dots, l_n)\;\mathrm{in}\; s]\!] & = & \mathrm{letrec}\;(i_1, \dots, i_n) = ([l_1], \dots, [l_n])\;\mathrm{in}\;[\![s]\!] \\
[c] & = & c \\
[i] & = & i \\
[op\,(t_1, \dots, t_m)] & = & op([t_1], \dots, [t_m]) \\
[\lambda\,(i_1, \dots, i_n, k)\,.\,s] & = & \lambda\,(i_1, \dots, i_n)\,.\,[\![s]\!]
\end{array}$$

Figure 2: Pure direct style transformation

The linearity condition is characterized by the inference rules in Figure 1 (where "$\otimes_n$" stands for "n-ary exclusive or"). A continuation $\lambda\, v\,.\,s$ is linear in its parameter v when the judgment

$$v \vdash s$$

is satisfied. A let expression can be inserted at syntax-analysis time to satisfy this linearity condition.

The attribute κ is used in this BNF to ensure that only the current continuation is applied and to distinguish the identifier representing the current continuation from other identifiers. Once the source program has been properly parsed, it is unnecessary to check this separation. Therefore we omit the κ attribute below.

2.2 BNF of pure DS terms

Our DS transformation maps CPS terms into pure DS terms, as defined by the following BNF. A DS λ_v-term, e, is defined by the following grammar.

$e \in$ DExp — domain of DS expressions

$l \in$ DLam — domain of λ-abstractions

$$\begin{array}{rl}
e ::= & @\,e_0\,(e_1, \dots, e_n) \\
 & \mid \mathrm{cond}(e_1, e_2, e_3) \\
 & \mid \mathrm{let}\;(i_1, \dots, i_n) = (e_1, \dots, e_n)\;\mathrm{in}\; e \\
 & \mid \mathrm{letrec}\;(i_1, \dots, i_n) = (l_1, \dots, l_n)\;\mathrm{in}\; e \\
 & \mid c \\
 & \mid i \\
 & \mid op\,(e_1, \dots, e_m) \\
 & \mid l \\
l ::= & \lambda\,(i_1, \dots, i_n)\,.\,e
\end{array}$$

2.3 Pure DS transformation

As derived in "Back to Direct Style I" [9], the term $@\,(\lambda\, k\,.\,s)(\lambda\, v\,.\,v)$ is rewritten as $[\![s]\!]$, where the rewrite function $[\![\;]\!]$ is defined inductively in Figure 2. $[\![\;]\!]$ translates serious terms. $[\;]$ translates trivial terms.

$$[\![\;]\!]\text{: CSer} \rightarrow \text{DExp}$$

$$[\;]\text{: CTriv} \rightarrow \text{DExp}$$

The redexes introduced for applications and continuation declarations are reduced at translation time. Given our hypothesis that the DS terms will be evaluated using the strategy assumed by the CPS transformation, this administrative reduction is safe because of the linearity on continuation parameters [9].

3 Extending the language of CPS terms

The second-class constraint allows only the current continuation identifier to be applied. In this section, we examine the effect of relaxing this constraint.

3.1 First extension

Let us allow any lexically visible continuation identifier to be applied. Now, while some continuation identifier must appear in every serious subexpression of each expression, it need no longer be the current continuation identifier. The resulting language is characterized by the following *first-class constraint.*

Definition 2 (First-class constraint)

1. *Any lexically visible continuation identifier can be applied.*
2. *Some continuation identifier must occur in every serious subexpression of each expression.*

To allow any lexically visible continuation to be applied, let us extend the BNF of Section 2.1 with a new production

$$s^{\kappa} ::= @\, k\, t \qquad \textit{where } k \neq \kappa$$

Occurrences of this production satisfy the first-class constraint, but not the second-class constraint. A continuation identifier that is used somewhere according to this new production denotes a *first-class continuation.* A continuation identifier that is only used according to the original production

$$s^{\kappa} ::= @\, k\, t \qquad \textit{where } k = \kappa$$

denotes a *second-class continuation.*

In the original BNF, the attribute ensures that the continuation identifier k in $@\, k\, t$ represents the current continuation. We could generalize this attribute to account for all the continuation identifiers visible in the current scope. Instead, let us keep the attribute κ and introduce a new attribute γ denoting the set of lexically visible continuation identifiers other than the current one.

A CPS λ-term, e, is then defined by the following BNF.

$$\begin{array}{lll}
e & ::= @\, (\lambda\, k\, .\, s^{k,\emptyset})\, (\lambda\, v\, .\, v) & \\
s^{\kappa,\gamma} & ::= @\, k\, t^{\gamma'} & \textit{where } \gamma' = \{\kappa\} \cup \gamma \textit{ and } k \in \gamma' \\
 & \mid @\, t_0^{\gamma'}\, (t_1^{\gamma'}, ..., t_n^{\gamma'}, \lambda\, v\, .\, s^{\kappa,\gamma}) & \\
 & & \textit{where } \gamma' = \{\kappa\} \cup \gamma \\
 & & \textit{and } v \notin \gamma' \\
 & & \textit{and } v \textit{ occurs linearly in } s \\
 & \mid \text{let } k = \lambda\, v\, .\, s_4^{\kappa,\gamma} \text{ in cond}(t_1^{\gamma''}, s_2^{k,\gamma'}, s_3^{k,\gamma'}) & \\
 & & \textit{where } \gamma' = \{\kappa\} \cup \gamma \\
 & & \textit{and } \gamma'' = \{k\} \cup \gamma' \\
 & & \textit{and } v \notin \gamma' \\
 & & \textit{and } v \textit{ occurs linearly in } s_4 \\
 & \mid \text{let } (i_1, ..., i_n) = (t_1^{\gamma''}, ..., t_n^{\gamma''}) \text{ in } s^{\kappa,\gamma'} & \\
 & & \textit{where } \kappa \notin \{i_1, ..., i_n\} \\
 & & \textit{and } \gamma' = \gamma \setminus \{i_1, ..., i_n\} \\
 & & \textit{and } \gamma'' = \{\kappa\} \cup \gamma \\
 & \mid \text{letrec } (i_1, ..., i_n) = (l_1^{\gamma''}, ..., l_n^{\gamma''}) \text{ in } s^{\kappa,\gamma'} & \\
 & & \textit{where } \kappa \notin \{i_1, ..., i_n\} \\
 & & \textit{and } \gamma' = \gamma \setminus \{i_1, ..., i_n\} \\
 & & \textit{and } \gamma'' = \{\kappa\} \cup \gamma' \\
t^{\gamma} & ::= c & \\
 & \mid i & \textit{where } i \notin \gamma \\
 & \mid op\, (t_1^{\gamma}, ..., t_m^{\gamma}) & \\
 & \mid l^{\gamma} & \\
l^{\gamma} & ::= \lambda\, (i_1, ..., i_n, k)\, .\, s^{k,\gamma'} & \\
 & & \textit{where } \gamma' = \gamma \setminus \{i_1, ..., i_n, k\}
\end{array}$$

Definition 3 *A continuation identifier k occurs in* first-class position *when it is used by an instance of the production*

$$s^{\kappa} ::= @\, k\, t \qquad \textit{where } k \neq \kappa.$$

Definition 4 *A continuation identifier k occurs in* second-class position *when it is used by an instance of the production*

$$s^{\kappa} ::= @\, k\, t \qquad \textit{where } k = \kappa.$$

3.2 Second extension

We make another change to the syntax independently of the new constraint. According to the extended BNF above, new continuations can only be declared around conditional expressions. Let us extend this further by allowing the declaration of a continuation anywhere as a serious expression. This is captured in the following BNF. For conciseness, we only reproduce what is new. (The two new productions replace the production for conditional expressions.)

$$\begin{array}{lll}
e & ::= ... & \\
s^{\kappa,\gamma} & ::= ... & \\
 & \mid \text{let } k = \lambda\, v\, .\, s_2^{\kappa,\gamma} \text{ in } s_1^{k,\gamma'} & \\
 & & \textit{where } \gamma' = \{\kappa\} \cup \gamma \\
 & & \textit{and } v \notin \gamma' \\
 & & \textit{and } v \textit{ occurs linearly in } s_2 \\
 & \mid \text{cond}(t_1^{\gamma'}, s_2^{\kappa,\gamma}, s_3^{\kappa,\gamma}) & \\
 & & \textit{where } \gamma' = \{\kappa\} \cup \gamma \\
 & \mid ... & \\
t^{\gamma} & ::= ... & \\
l^{\gamma} & ::= ... &
\end{array}$$

Like the attribute κ of the original BNF, γ is needed only to specify the new BNF. Therefore, we generally omit both κ and γ in the rest of this paper.

4 Direct style transformation

We now extend the pure DS transformation to the terms generated by the extended BNF. We want to define the new transformation as a conservative extension. Accordingly, we must determine how the source program varies from the original BNF. In this section we first present a counting analysis to determine whether a continuation identifier occurs in first-class position. We then define the BNF of extended DS terms, and finally we use the counting analysis to extend the DS transformation.

$$\frac{\kappa \notin \mathrm{FV}(t)}{\kappa \vdash t : \bot} \qquad \frac{\kappa \in \mathrm{FV}(t)}{\kappa \vdash t : \top} \qquad \frac{\kappa \vdash t : q}{\kappa \vdash @\, k\ t : q}$$

$$\frac{\kappa \neq k \wedge \kappa \in \mathrm{FV}(s_1) \qquad \kappa \vdash s_2 : q_2}{\kappa \vdash \mathrm{let}\ k = \lambda\, v \,.\, s_2\ \mathrm{in}\ s_1 : \top} \qquad \frac{\kappa = k \vee \kappa \notin \mathrm{FV}(s_1) \qquad \kappa \vdash s_2 : q_2}{\kappa \vdash \mathrm{let}\ k = \lambda\, v \,.\, s_2\ \mathrm{in}\ s_1 : q_2}$$

$$\frac{\kappa \vdash t_0 : q_0 \qquad \ldots \qquad \kappa \vdash t_n : q_n \qquad \kappa \vdash s : q_{n+1}}{\kappa \vdash @\, t_0\ (t_1,\ \ldots,\ t_n,\ \lambda\, v \,.\, s) : \bigsqcup_{j=0}^{n+1} q_j} \qquad \frac{\kappa \vdash t_1 : q_1 \qquad \kappa \vdash s_2 : q_2 \qquad \kappa \vdash s_3 : q_3}{\kappa \vdash \mathrm{cond}(t_1,\ s_2,\ s_3) : \bigsqcup_{j=1}^{3} q_j}$$

$$\frac{\kappa \in \{i_1,\ \ldots,\ i_n\} \qquad \kappa \vdash t_1 : q_1 \qquad \ldots \qquad \kappa \vdash t_n : q_n}{\kappa \vdash \mathrm{let}\ (i_1,\ \ldots,\ i_n) = (t_1,\ \ldots,\ t_n)\ \mathrm{in}\ s : \bigsqcup_{j=1}^{n} q_j} \qquad \frac{\kappa \notin \{i_1,\ \ldots,\ i_n\} \qquad \kappa \vdash t_1 : q_1 \qquad \ldots \qquad \kappa \vdash t_n : q_n \qquad \kappa \vdash s : q_0}{\kappa \vdash \mathrm{let}\ (i_1,\ \ldots,\ i_n) = (t_1,\ \ldots,\ t_n)\ \mathrm{in}\ s : \bigsqcup_{i=0}^{n} q_i}$$

$$\frac{\kappa \in \{i_1,\ \ldots,\ i_n\}}{\kappa \vdash \mathrm{letrec}\ (i_1,\ \ldots,\ i_n) = (l_1,\ \ldots,\ l_n)\ \mathrm{in}\ s : \bot} \qquad \frac{\kappa \notin \{i_1,\ \ldots,\ i_n\} \qquad \kappa \vdash l_1 : q_1 \qquad \ldots \qquad \kappa \vdash l_n : q_n \qquad \kappa \vdash s : q_0}{\kappa \vdash \mathrm{letrec}\ (i_1,\ \ldots,\ i_n) = (l_1,\ \ldots,\ l_n)\ \mathrm{in}\ s : \bigsqcup_{j=0}^{n} q_j}$$

Figure 3: Counting analysis

4.1 Tracking continuations

4.1.1 A domain for counting

We must determine the class of each continuation identifier. Let us define a two point domain

$$\langle \{\bot, \top\}, \sqsubset \rangle$$

where $\bot \sqsubset \top$. We associate $\top$ to continuation identifiers that occur in first-class position, and $\bot$ to the others, *i.e.*, to the identifiers that always occur in second-class position or do not occur at all.

Our counting analysis checks for first-class occurrences. If a continuation identifier k occurs free in the scope of another continuation identifier k', k must occur in first-class position. We want its count then to be $\top$. Otherwise, if all occurrences of k are in second-class positions, then its count should be $\bot$.

Let us also observe that the only trivial term that has a serious term as a subexpression is a λ-expression. But λ-expressions declare new continuation identifiers. Therefore if a continuation identifier occurs free in a trivial term, it denotes a first-class continuation.

Based on these observations we define the analysis as follows. Expressions have two kinds of subexpressions, those that declare a new continuation identifier and those that do not. If the continuation identifier occurs free in a subexpression that declares a new continuation identifier, the count is $\top$. Otherwise the count is the least upper bound of the counts for the other subexpressions.

Because the count for a subexpression declaring a new continuation identifier is determined just by a free-variable test, the algorithm does not explicitly consider the case where κ is not the current continuation identifier. Thus, in particular, the counting analysis of $@\, k\ t$ assumes that either $k = \kappa$ and this expression is an occurrence in second-class position, or that $k \neq \kappa$ and this expression is not an occurrence of κ.

The full counting analysis is displayed in Figure 3. The continuation identifier κ has count q in a serious expression s whenever $\kappa \vdash s : q$, and a count q in a trivial expression t whenever $\kappa \vdash t : q$.

4.1.2 Assessment

We now have a counting analysis that determines whether a continuation parameter satisfies the second-class constraint, or whether it only satisfies the first-class constraint. In our previous work on the DS transformation [9], we fundamentally assumed that all continuation identifiers satisfied the second-class constraint.

Proposition 1 *Let k be a continuation identifier in a pure CPS term as defined by the BNF of Section 2.1. Then*

$$\begin{cases} k \vdash s^k : \bot \\ k \vdash t^k : \bot \end{cases}$$

□

4.2 BNF of extended DS terms

Our DS transformation maps CPS terms into pure DS terms, as defined by the following BNF. A DS λ_v-term, $e^{\emptyset}$, is defined by the following attribute grammar.

$$\begin{array}{rll} e^{\gamma} ::= & \mathrm{call/cc}\ (\lambda\, k \,.\, e^{\gamma'}) & \textit{where } \gamma' = \{k\} \cup \gamma \\ \mid & \mathrm{throw}\ k\ e^{\gamma} & \textit{where } k \in \gamma \\ \mid & @\, e_0^{\gamma}\ (e_1^{\gamma},\ \ldots,\ e_n^{\gamma}) & \\ \mid & \mathrm{cond}(e_1^{\gamma},\ e_2^{\gamma},\ e_3^{\gamma}) & \\ \mid & \mathrm{let}\ (i_1,\ \ldots,\ i_n) = (e_1^{\gamma},\ \ldots,\ e_n^{\gamma})\ \mathrm{in}\ e^{\gamma'} & \\ & & \textit{where } \gamma' = \gamma \setminus \{i_1,\ \ldots,\ i_n\} \\ \mid & \mathrm{letrec}\ (i_1,\ \ldots,\ i_n) = (l_1^{\gamma'},\ \ldots,\ l_n^{\gamma'})\ \mathrm{in}\ e^{\gamma'} & \\ & & \textit{where } \gamma' = \gamma \setminus \{i_1,\ \ldots,\ i_n\} \\ \mid & c & \end{array}$$

$$
\begin{aligned}
\llbracket @\, k\ t \rrbracket\, \kappa &= \begin{cases} [t] & \textit{if } k = \kappa \\ \text{throw}\ k\ [t] & \textit{otherwise} \end{cases} \\
\llbracket @\, t_0\ (t_1,\ \ldots,\ t_n,\ \lambda\, v\, .\, s) \rrbracket\, \kappa &= @\, (\lambda\, v\, .\, \llbracket s \rrbracket\, \kappa)\, (@\, [t_0]\ ([t_1],\ \ldots,\ [t_n])) \\
\llbracket \text{cond}(t_1,\ s_2,\ s_3) \rrbracket\, \kappa &= \text{cond}([t_1],\ \llbracket s_2 \rrbracket\, \kappa,\ \llbracket s_3 \rrbracket\, \kappa) \\
\llbracket \text{let}\ k = \lambda\, v\, .\, s_2\ \text{in}\ s_1 \rrbracket\, \kappa &= \begin{cases} @\, (\lambda\, v\, .\, \llbracket s_2 \rrbracket\, \kappa)\, (\text{call/cc}\ (\lambda\, k\, .\, \llbracket s_1 \rrbracket\, k)) & \textit{if } k \vdash s_1 : \top \\ @\, (\lambda\, v\, .\, \llbracket s_2 \rrbracket\, \kappa)\, (\llbracket s_1 \rrbracket\, k) & \textit{otherwise} \end{cases} \\
\llbracket \text{let}\ (i_1,\ \ldots,\ i_n) = (t_1,\ \ldots,\ t_n)\ \text{in}\ s \rrbracket\, \kappa &= \text{let}\ (i_1,\ \ldots,\ i_n) = ([t_1],\ \ldots,\ [t_m])\ \text{in}\ \llbracket s \rrbracket\, \kappa \\
\llbracket \text{letrec}\ (i_1,\ \ldots,\ i_n) = (l_1,\ \ldots,\ l_n)\ \text{in}\ s \rrbracket\, \kappa &= \text{letrec}\ (i_1,\ \ldots,\ i_n) = ([l_1],\ \ldots,\ [l_m])\ \text{in}\ \llbracket s \rrbracket\, \kappa \\
[c] &= c \\
[i] &= i \\
[op\ (t_1,\ \ldots,\ t_m)] &= op([t_1],\ \ldots,\ [t_m]) \\
[\lambda\, (i_1,\ \ldots,\ i_n,\ k)\, .\, s] &= \begin{cases} \lambda\, (i_1,\ \ldots,\ i_n)\, .\, \text{call/cc}\ (\lambda\, k\, .\, \llbracket s \rrbracket\, k) & \textit{if } k \vdash s : \top \\ \lambda\, (i_1,\ \ldots,\ i_n)\, .\, \llbracket s \rrbracket\, k & \textit{otherwise} \end{cases}
\end{aligned}
$$

Figure 4: New direct style transformation

$$
\begin{array}{lll}
 & \mid i & \textit{where } i \notin \gamma \\
 & \mid op\ (e_1^\gamma,\ \ldots,\ e_m^\gamma) & \\
 & \mid l^\gamma & \\
l^\gamma ::= & \lambda\, (i_1,\ \ldots,\ i_n)\, .\, e^{\gamma'} & \textit{where } \gamma' = \gamma \setminus \{i_1,\ \ldots,\ i_n\}
\end{array}
$$

The attribute γ denotes the set of identifiers declared with call/cc that are lexically visible. Such identifiers can only occur when they are syntactically sugared with throw. (NB: throw can be inserted at syntax-analysis time.)

4.3 The DS transformation

Let us now extend the original DS transformation to handle the new productions of the relaxed language. The translations of the terms

$$\text{let}\ k = \lambda\, v\, .\, s_2\ \text{in}\ s_1$$

and

$$\text{cond}(t_1,\ s_2,\ s_3)$$

can easily be derived from the translation of the pure CPS term

$$\text{let}\ k = \lambda\, v\, .\, s_4\ \text{in}\ \text{cond}(t_1,\ s_2,\ s_3)$$

Thus in this section we concentrate on the production

$$s\ ::=\ @\, k\, t$$

The pure DS transformation of $@\, k\, t$ makes k disappear. If we translate all continuation applications this way, the value of the continuation argument will always be returned to the current continuation and never to any first-class continuation. Thus identifiers denoting first-class continuations must be retained in the translation. If the continuation that is sent a value is not the current continuation, we translate this continuation application with throw. The counting analysis guarantees us that this continuation identifier has a count $\top$.

If a continuation identifier appears in the translated term, it must be declared somewhere. In the CPS program, the value of a continuation identifier represents the continuation current at the point it is declared. In the DS language, call/cc binds an identifier to a representation of the current continuation. Therefore we translate the declaration of a first-class continuation identifier into the declaration of a reified continuation with call/cc, based on the results of the counting analysis (*cf.* Figure 3) that guarantee us that this continuation identifier occurs in first-class position.

The full translation is displayed in Figure 4. The term $@\, (\lambda\, k\, .\, s)\, (\lambda\, v\, .\, v)$ is rewritten as $\llbracket s \rrbracket\, k$.

$$\llbracket\ \rrbracket\text{: CSer} \rightarrow \text{Ide} \rightarrow \text{DExp}$$
$$[\ ]\text{: CTriv} \rightarrow \text{DExp}$$

The redexes introduced for applications and continuation declarations are reduced at translation time. These administrative reductions are safe because of the linearity of continuation parameters, under the hypothesis that the DS terms will be evaluated using the strategy assumed by the CPS transformation.

5 Connecting the DS and the CPS transformations (outline)

In this section we outline the proof that the DS transformation and the CPS transformation form a Galois connection. The complete proof is in the full paper.

5.1 Staging

Ideally, one would symbolically compose the DS and the CPS transformations and simplify them inductively termwise to obtain *e.g.*, the identity transformation. Unfortunately, as they are stated, the two transformations do not allow such a simple proof because the administrative redexes are reduced at translation time. Terms that should be simplified to obtain the identity transformation disappear due to the administrative reductions.

In an earlier work [8], Danvy circumvented this problem by staging the CPS transformation into three steps. In the following, we generalize the method and stage the DS and the CPS transformations to factor the administrative reductions out of *both* transformations. We thus define two intermediate languages: the language of staged DS terms and the language of staged CPS terms.

The language of staged DS terms differs from the language of DS terms only in that *all intermediate values are named in the staged language.* This naming is done by λ-abstraction. A DS term is staged by introducing a β-redex for each intermediate value. A staged DS term is unstaged by reducing just these β-redexes.

The language of staged CPS terms differs from the language of CPS terms only in that *all intermediate continuations are named in the staged language.* This naming is done by λ-abstraction. A CPS term is staged by introducing a β-redex for each intermediate continuation. A staged CPS term is unstaged by reducing just these β-redexes.

By construction, staging and unstaging are inverse transformations up to renaming.

We then define the DS and CPS transformations on these staged terms. In contrast with the two other steps and with the usual CPS transformation, these transformations are carried out by local, structure-preserving rewriting rules. The administrative reductions of our DS transformation or of the usual CPS transformation coincide with unstaging. Correspondingly, staging coincide with the administrative expansions.

By construction, staging a DS term, mapping this staged DS term into a staged CPS term, and unstaging this staged CPS term into a CPS term amounts to transforming the DS term into CPS. Conversely, staging a CPS term, mapping this staged CPS term into a staged DS term, and unstaging this staged DS term into a DS term amounts to transforming the CPS term into DS.

$$\text{DS} \longleftrightarrow \text{staged DS} \longleftrightarrow \text{staged CPS} \longleftrightarrow \text{CPS}$$

Let us illustrate the new CPS and DS transformations with an example.

5.2 An example

Figure 5 displays the usual CPS procedure `product` mapping a list of numbers into the product of these numbers. This procedure uses continuations in an interesting way. If the absorbent element zero occurs in the list, the continuation of the call to `product` is sent zero "in advance", *i.e.*, without multiplying the remaining numbers by zero.

Let us map `product` back to direct style. The transformation, as given in Figure 4, is somewhat unintuitive because of its administrative simplifications that perform non-local transformations. Instead, let us stage this transformation and express `product` first in staged CPS, then in staged DS, and then in DS.

Staging a CPS program amounts to naming non-trivial intermediate continuations. We name them with let expressions. `Product` contains only one non-trivial intermediate continuation:

```
(lambda (v) (k1 (* (car l) v)))
```

This intermediate continuation gets named and the rest of the program remains unchanged (*cf.* Figure 6).

Next we transform `product` into staged DS, naming all intermediate values with let expressions. The transformation is comparable to the DS transformation of Figure 4, but the rewrite rules causing administrative β-redexes in the result now produce let expressions. Let expressions naming continuations are transformed as follows:

$$\frac{k' \vdash s_1 \Rightarrow r_1 \qquad k \vdash s_2 \Rightarrow r_2}{k \vdash \text{let } k' = \lambda\, v\,.\, s_2 \text{ in } s_1 \Rightarrow \text{let } v = r_1 \text{ in } r_2}$$

As can be noticed, `k0` is used in the scope of another continuation, `k1`. Therefore,

$$\texttt{k0} \vdash \texttt{(letrec ...)} : \top$$

Thus `k0` needs to be declared at the top of the program with `call/cc`. Figure 7 displays the resulting staged DS program.

Our final step is to unstage the staged DS version of `product`. Unstaging is simply achieved by unfolding all the linear let expressions. These are precisely the administrative β-reductions performed in the full DS transformation (*cf.* Section 4.3). Figure 8 displays the resulting DS program.

We can reverse each step of the transformation. Staging a DS program amounts to naming the results of all non-trivial sub-expressions. In Figure 8, `product` contains only one non-trivial expression:

```
(traverse (cdr l))
```

We name intermediate values using let. The rest of the program remains unchanged (*cf.* Figure 7). Then we transform `product` into staged CPS, naming all intermediate continuations using let (*cf.* Figure 6). Essentially, the transformation is comparable to the CPS transformation [11], but the rewrite rules causing administrative β-redexes in the result now produce let expressions. As a final step, we unstage the staged CPS version of `product` (*cf.* Figure 5). Unstaging is simply achieved by reducing all the linear let expressions that do not declare first-class continuations (as determined by our counting analysis). These are precisely the administrative β-reductions performed in the full CPS transformation.

As should be noticed, the step from staged DS to staged CPS is at the heart of the CPS transformation, and symmetrically, the step from staged CPS to staged DS is at the heart of the DS transformation. The two other steps (staging and unstaging) only carry out administrative β-expansions and β-reductions. This red tape is confusing because it changes the term globally. In contrast, transforming a staged CPS term into a staged DS term is carried out by local, structure-preserving rewriting rules.

5.3 A Galois connection

At this point we would like to state that the DS transformation and the CPS transformation are inverses, but it is not as simple as that. Instead, we can only state the following weaker property.

Let $\mathcal{D}$ denote the DS transformation and $\mathcal{C}$ denote the CPS transformation. In the rest of this section, d denotes a DS λ_v-term and c denotes a CPS λ-term.

Proposition 2
$$\begin{aligned} \forall d,\quad \mathcal{C}(\mathcal{D}(\mathcal{C}(d))) &= \mathcal{C}(d) \\ \forall c,\quad \mathcal{D}(\mathcal{C}(\mathcal{D}(c))) &= \mathcal{D}(c) \end{aligned}$$

□

Why? Think of a DS term that captures the current continuation but does not use it on a first-class basis. Mapping such a term into CPS and then back to DS yields a term without `call/cc`. Conversely, think of a CPS term explicitly

```
(define product                ;;; List(Num) * [Num -> Ans] -> Ans
  (lambda (l k0)
    (letrec ([traverse         ;;; List(Num) * [Num -> Ans] -> Ans
              (lambda (l k1)
                (cond [(null? l) (k1 1)]
                      [(zero? (car l)) (k0 0)]
                      [else (traverse (cdr l) (lambda (v)
                                                (k1 (* (car l) v))))]))])
      (traverse l (lambda (v)
                    (k0 v))))))
```

Figure 5: CPS program

```
(define product                ;;; List(Num) * [Num -> Ans] -> Ans
  (lambda (l k0)
    (letrec ([traverse         ;;; List(Num) * [Num -> Ans] -> Ans
              (lambda (l k1)
                (cond [(null? l) (k1 1)]
                      [(zero? (car l)) (k0 0)]
                      [else (let ([k2 (lambda (v)
                                        (k1 (* (car l) v)))])
                              (traverse (cdr l) (lambda (v)
                                                  (k2 v))))]))])
      (traverse l (lambda (v)
                    (k0 v))))))
```

Figure 6: Staged CPS program (all intermediate continuations are named)

```
(define product                ;;; List(Num) -> Num
  (lambda (l)
    (call/cc
      (lambda (k0)
        (letrec ([traverse     ;;; List(Num) -> Num
                  (lambda (l)
                    (cond [(null? l) 1]
                          [(zero? (car l)) (throw k0 0)]
                          [else (let ([v (traverse (cdr l))])
                                  (* (car l) v))]))])
          (traverse l))))))
```

Figure 7: Staged DS program (all intermediate values are named)

```
(define product
  (lambda (l)                  ;;; List(Num) -> Num
    (call/cc
      (lambda (k0)
        (letrec ([traverse     ;;; List(Num) -> Num
                  (lambda (l)
                    (cond [(null? l) 1]
                          [(zero? (car l)) (throw k0 0)]
                          [else (* (car l) (traverse (cdr l)))]))])
          (traverse l))))))
```

Figure 8: DS program

naming a continuation whose count is not $\top$. Mapping such a term into DS and then back to CPS yields a term where the continuation is not explicitly named with a let.

In the full version of this paper, we introduce a notion of *normalization.* Normalizing a DS term (resp. a CPS term) roughly amounts to simplifying it in a meaning-preserving way, such that a DS (resp. CPS) term and its normalized form get transformed into the *same* CPS (resp. DS) term (modulo α-conversion).

We define a normalization function $\mathcal{N}$

$$\mathcal{N} : \text{DExp} \rightarrow \text{DExp}$$

as a rewrite system over DS terms. The rewrite system is strongly normalizing and confluent.

Symmetrically, we define a normalization function $\mathcal{M}$

$$\mathcal{M} : \text{CExp} \rightarrow \text{CExp}$$

as a rewrite system over CPS terms. The rewrite system is strongly normalizing and confluent.

$\mathcal{M}$ and $\mathcal{N}$ satisfy the following properties.

Property 1 *The CPS and the DS transformations produce normalized terms:*

$$\begin{array}{rcl} \forall d,\ \mathcal{C}(d) & = & \mathcal{M}(\mathcal{C}(d)) \\ \forall c,\ \mathcal{D}(c) & = & \mathcal{N}(\mathcal{D}(c)) \end{array}$$

Property 2 *The CPS (resp. DS) transformation of a term yields the same result as the transformation of the corresponding normalized term:*

$$\begin{array}{rcl} \forall d,\ \mathcal{C}(d) & = & \mathcal{C}(\mathcal{N}(d)) \\ \forall c,\ \mathcal{D}(c) & = & \mathcal{D}(\mathcal{M}(c)) \end{array}$$

Property 3 *The CPS and the DS transformations are inverses of each other over normalized terms:*

$$\begin{array}{rcl} \forall d,\ \mathcal{D}(\mathcal{C}(\mathcal{N}(d))) & = & \mathcal{N}(d) \\ \forall c,\ \mathcal{C}(\mathcal{D}(\mathcal{M}(c))) & = & \mathcal{M}(c) \end{array}$$

Let us consider the partial order naturally induced by $\mathcal{M}$ and $\mathcal{N}$:

$$\begin{cases} \forall c,\ \mathcal{M}(c) \sqsubseteq c \\ \forall d,\ \mathcal{N}(d) \sqsubseteq d \end{cases}$$

Notice that by Proposition 2,

$$\begin{cases} c \sqsubseteq c' \ \Rightarrow\ \mathcal{C}(c) = \mathcal{C}(c') \\ d \sqsubseteq d' \ \Rightarrow\ \mathcal{D}(d) = \mathcal{D}(d') \end{cases}$$

Then we can define classes of DS and of CPS terms. The characteristic element of a class is the normalized term. It is the smallest of its class. By construction, any element of the DS class (resp. of the CPS class) is mapped to a CPS term (resp. to a DS term) that is mapped back to the characteristic element of the class. This property is characteristic of Galois connections [21].

Proposition 3 *$\mathcal{C}$ and $\mathcal{D}$ form a Galois connection.*

Proof: Based on the three properties above, the equivalence

$$\mathcal{C}(d) \sqsubseteq c \Leftrightarrow d \sqsupseteq \mathcal{D}(c)$$

is simple to prove, for any c and d. □

Of course, the Galois connection suggests to define the normalization functions in term of the CPS and of the DS transformations:

$$\begin{cases} \mathcal{M} = \mathcal{D} \circ \mathcal{C} \\ \mathcal{N} = \mathcal{C} \circ \mathcal{D} \end{cases}$$

5.4 Issues

We believe that a Galois connection offers a nice basis to reason about other kinds of simplifications in the DS world, seen from the CPS world, and in the CPS world, seen from the DS world. Examples include, *e.g.*, compile-time optimizations. Experience shows, though, that one should then be careful that continuations remain linear in the sense of Figure 1. Otherwise, to preserve the validity of administrative reductions, let expressions should be inserted to maintain linearity (our implementation inserts such let expressions automatically) and thus the evaluation order.

The problem is more general, however: our DS transformer maps a CPS term (in which the evaluation order is *explicitely* specified) into the corresponding DS term in which the very same evaluation order is *implicitely* specified. Which changes in the implicit evaluation order or in the explicit evaluation order are possible without getting a semantic mismatch? Our Galois connection could offer a model for characterizing safe changes.

6 An Experiment with Partial Evaluation

6.1 Nature of the experiment

This section describes the automatic specialization of the classical samefringe program for binary trees with respect to one binary tree. We express this program using `call/cc` but without side-effects, based on the detach model of coroutines [7, 17]. The program is first transformed into CPS; it is then specialized. The result is then transformed back into DS.

6.2 Partial evaluation

Partial evaluation is a semantics-based program transformation technique aimed at specializing a "source" program p_{src} with respect to a "static" part s of its input data. Partial evaluation produces a "residual" program p_{res}. The programs p_{src} and p_{res} are related as follows. Running p_{res} on the "dynamic" part d of the input data produces the same result r as running p_{src} on both s and d (but usually running p_{res} is faster). This is captured in the following equations that paraphrase Kleene's S^m_n-theorem.

$$\begin{cases} \text{run } pe\ \langle p_{src}, \langle s, \_\rangle\rangle & = & p_{res} \\ \text{run } p_{res}\langle d\rangle & = & \text{run } p_{src}\langle s, d\rangle \end{cases}$$

In the first equation, *pe* denotes a partial evaluator. Of course, these equations only hold for terminating programs p_{src} and p_{res}, and if partial evaluation terminates.

We do not know of any partial evaluator today that can handle control operators directly. Therefore, if we want to specialize a program involving `call/cc`, it is natural to transform it first into CPS, then to specialize it using a higher-order partial evaluator, and to finally transform the residual program into DS. This is captured in the following expression:

$$\mathcal{D}(\text{run } pe\ \langle \mathcal{C}(p_{src}), \langle s, \_\rangle\rangle)$$

where $\mathcal{C}$ denotes the CPS transformation and $\mathcal{D}$ denotes our DS transformation.

We are using Consel's self-applicable partial evaluator Schism [3, 4, 5]. Through all our experiments, Schism has let residual continuations remain last in every argument list.

```
(defineType 2Tree                                   (defineType Data
  (Pair left right)                                   (Next leaf continuation)
  (Leaf l))                                           (Over))
```

Figure 9: Data structures for the Samefringe program

```
(define (main bt1 bt2)         ;;; 2Tree * 2Tree -> Bool
  (skim (initialize bt1) (initialize bt2)))

(define (skim d1 d2)           ;;; Data * Data -> Bool
  (caseType d1
    [(Next l1 k1) (caseType d2
                    [(Next l2 k2) (and (equal? l1 l2)
                                       (skim (resume k1) (resume k2)))]
                    [(Over) #f])]
    [(Over) (caseType d2
              [(Next l2 k2) #f]
              [(Over) #t])]))

(define (initialize bt)        ;;; 2Tree -> Data
  (call/cc (lambda (k)
             ((defoliate bt (lambda (v) (throw k v))) (Over)))))

(define (resume c)             ;;; Cont -> Ans
  (call/cc (lambda (k)
             (c (lambda (v) (throw k v))))))

(define (defoliate bt k)       ;;; 2Tree * Cont -> Ans
  (caseType bt
    [(Pair left right) (defoliate right (defoliate left k))]
    [(Leaf l) (call/cc (lambda (kp)
                         (k (Next l (lambda (v) (throw kp v))))))]))
```

Figure 10: Control structures for the Samefringe program

```
;;; for all bt2, (main0 bt2) == (main (Pair (Pair (Leaf 0) (Leaf 1)) (Pair (Leaf 2) (Leaf 3))) bt2)

(define (main0 bt2)            ;;; 2Tree -> Bool
  (caseType (initialize bt2)
    [(Next l1 k1) (and (equal? '0 l1)
                       (caseType (resume k1)
                          [(Next l2 k2) (and (equal? '1 l2)
                                             (caseType (resume k2)
                                               [(Next l3 k3) (and (equal? '2 l3)
                                                                  (caseType (resume k3)
                                                                    [(Next l4 k4) (and (equal? '3 l4)
                                                                                       (caseType (resume k4)
                                                                                         [(Next l5 k5) #f]
                                                                                         [(Over) #t]))]
                                                                    [(Over) #f]))]
                                               [(Over) #f]))]
                          [(Over) #f]))]
    [(Over) #f]))
```

Figure 11: Specialized version of the Samefringe program

6.3 The experiment

Figures 9 and 10 display the source program and its data structures.[2] We automatically transform this program into CPS. Schism automatically specializes it. We automatically transform the result into DS. Figure 11 displays a slightly pretty-printed version of the result (local variables have been renamed).

6.4 Assessment

As a whole, the static coroutine has been executed statically. Its computational content has been entirely inlined in the main procedure, yielding an iterative-looking residual program — though in fact, the dynamic binary tree is still traversed recursively. The resulting program uses one separate coroutine to traverse the dynamic binary tree.

One could argue that the resulting program should not use any coroutine at all, but this would require a more radical program manipulation than partial evaluation. Such a transformation would involve a global Eureka step, as in Burstall & Darlington's framework [1]. This step goes beyond mere program specialization. Besides, if we want to specialize an n-ary samefringe program with respect to part of its input, the residual program would naturally be expressed in coroutine style.

In any case, this experiment illustrates a first: the successful specialization of programs involving operations over control.

7 Related Work

Naturally, this paper relies on Fischer's and Plotkin's fundamental work on CPS [15, 25]. The CPS transformation described by Danvy and Filinski [11] is the point of reference for Section 5. In an earlier work [9], Danvy developed the CPS-to-DS transformation described in Section 2. We know of no other work addressing the problem of converting CPS into DS, and none that handles CPS programs whose DS counterpart requires `call/cc`. Sabry and Felleisen's equational reasoning about CPS programs [30] might be related, but we have not yet had the opportunity to read their paper (published in this volume).

The detection of first-class continuations in Section 4.1 is purely syntactic and thus contrasts with Jouvelot and Gifford's or with Deutsch's more ambitious semantic analyses of programs with control effects [12, 19].

Finally, the symmetry between values and continuations encountered in Section 5 is reminiscent of Filinski's dualities [14].

8 Conclusions

We have presented a conservative extension of our DS transformer to handle `call/cc`. This conservative extension relies on a counting analysis that detects non-canonical occurrences of continuations in a CPS term. The DS and the CPS transformations form a Galois connection. Together they widen the class of programs that can be manipulated on a semantic basis. We illustrate this widening with a first: the successful self-applicable partial evaluation of a Scheme program containing first-class continuations.

We have also presented staged versions of both the CPS and the DS transformations. In contrast to the usual presentation of the CPS transformation, the staged transformations distinguish between administrative reductions and actual translations. In addition, the actual translations are carried out by local, structure-preserving rewriting rules that make it considerably easier to understand what is precisely going on in the process of shifting between DS and CPS terms.

9 Issues

We are now extending the DS transformation to cope with sequencing and side-effects in Scheme (on bindings with `set!`, on data structures with `set-car!` *et al.*, and on the outside world with `print` *et al.*).

We are also exploring the connection of the DS transformation with computational monads, following Wadler's insight that CPS and monads offer the same expressive power for structuring functional programs [10, 22, 33].

Based on the Curry-Howard isomorphism, the CPS transformation has been related to transformations on representations of proofs [16, 23]. By the same token, the DS transformation of CPS terms into DS terms with `call/cc` should have an interpretation in proof theory. We leave this aspect for a future work.

We are also investigating the DS transformation towards call-by-name functional terms.

Many new control operators are emerging today [10, 13, 18, 28, 32]. If CPS is to be used as a unifying framework to specify and relate them, it must be possible to shift back and forth between programs using these operators and purely functional programs. Therefore it is important to establish a sound understanding of the DS transformation and its relation to the CPS transformation.

Acknowledgments

To Dan Friedman for his support, to Andrzej Filinski for keenly commenting on earlier versions of this paper, and to the members of the programming language groups of Kansas State University and Indiana University for creating a lively and uplifting atmosphere.

References

[1] Rod M. Burstall and John Darlington. A transformational system for developing recursive programs. *Journal of ACM*, 24(1):44–67, 1977.

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

[3] Charles Consel. Binding time analysis for higher order untyped functional languages. In LFP'90 [20], pages 264–272.

[4] Charles Consel. *The Schism Manual.* Yale University, New Haven, Connecticut, December 1990. Version 1.0.

[5] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In POPL'91 [27], pages 14–24.

---

[2] We use Schism syntactic facilities for declaring and using data types (more precisely: constructor names and their arities). These facilities are not standard in Scheme, but they can be easily defined as macros [2].

[6] Charles Consel and Olivier Danvy. For a better support of static data flow. In *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.

[7] Ole-Johan Dahl and C.A.R. Hoare. Hierarchical program structures. In Ole-Johan Dahl, Edger Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, pages 157–220. Academic Press, 1972.

[8] Olivier Danvy. Three steps for the CPS transformation. Technical Report CIS-92-2, Kansas State University, Manhattan, Kansas, December 1991.

[9] 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.

[10] Olivier Danvy and Andrzej Filinski. Abstracting control. In LFP'90 [20], pages 151–160.

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

[12] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In POPL'90 [26], pages 157–168.

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

[14] Andrzej Filinski. Declarative continuations: An investigation of duality in programming language semantics. In David 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.

[15] 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.

[16] Timothy G. Griffin. A formulae-as-types notion of control. In POPL'90 [26], pages 47–58.

[17] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, Texas, August 1984.

[18] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 128–136, Seattle, Washington, March 1990. SIGPLAN Notices, Vol. 25, No. 3.

[19] Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Languages Design and Implementation*, pages 218–226, Portland, Oregon, June 1989.

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

[21] Austin Melton, David A. Schmidt, and George Strecker. Galois connections and computer science applications. In David H. Pitt et al., editors, *Category Theory and Computer Programming*, number 240 in Lecture Notes in Computer Science, pages 299–312, Guildford, UK, September 1986.

[22] 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.

[23] 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.

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

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

[26] *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 1990. ACM Press.

[27] *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 1991. ACM Press.

[28] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In POPL'91 [27], pages 174–184.

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

[30] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, California, June 1992.

[31] 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.

[32] Carolyn L. Talcott. *The Essence of $\mathcal{R}$um: A Theory of the Intensional and Extensional Aspects of Lisp-type Computation.* PhD thesis, Department of Computer Science, Stanford University, Stanford, California, August 1985.

[33] Philip Wadler. The essence of functional programming. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.