Separating Stages in the Continuation-Passing Style Transformation

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

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

Abstract

The continuation-passing style (CPS) transformation is powerful but complex. Our thesis is that this transformation is in fact *compound*, and we set out to *stage* it. We factor the CPS transformation into several steps, separating aspects in each step:

1. Intermediate values are named.
2. Continuations are introduced.
3. Sequencing order is decided and administrative reductions are performed.

Step 1 determines the evaluation order (*e.g.*, call-by-name or call-by-value). Step 2 isolates the introduction of continuations and is expressed with local, structure-preserving rewrite rules — a novel aspect standing in sharp contrast with the usual CPS transformations. Step 3 determines the ordering of continuations (*e.g.*, left-to-right or right-to-left evaluation) and leads to the familiar-looking continuation-passing terms.

Step 2 is completely reversible and Steps 1 and 3 form Galois connections. Together they leading to the direct style (DS) transformation of our earlier work (including first-class continuations):

1. Intermediate continuations are named and sequencing order is abstracted.
2. Second-class continuations are eliminated.
3. Administrative reductions are performed.

A subset of these transformations can leverage program manipulation systems: CPS-based compilers can modify sequencing to improve *e.g.*, register allocation; static program analyzers can yield more precise results; and overspecified CPS programs can be rescheduled. Separating aspects of the CPS transformation also enables a new programming style, with applications to nondeterministic programming. As a byproduct, our work also suggests a new continuation semantics for unspecified sequencing orders in programming languages (*e.g.*, Scheme).

1 Introduction

Continuation-passing style (CPS) terms enjoy a number of useful properties: They offer a good format for compiling and optimization [1, pages 4–6] and they enable non-trivial improvements for semantics-based program manipulation [6, 28]. CPS terms are independent of their evaluation order [29, 30] and CPS is at the basis of several mathematical semantics of programming languages [12, 36]. On top of that, the CPS transformation occurs in several areas of theoretical Computer Science, including Category Theory [13, 26, 37] and Logic [17, 27]. In short, there is something truly fundamental in the CPS transformation, even though it can be expressed in only three lines for the pure λ-calculus [29]:

$$\begin{array}{rcl} [\![x]\!]\,\kappa & = & \kappa\,x \\ [\![\lambda\,x\,.\,e]\!]\,\kappa & = & \kappa\,(\lambda\,x\,.\,\lambda\,k\,.\,[\![e]\!]\,k) \\ [\![e_0\,e_1]\!]\,\kappa & = & [\![e_0]\!]\,(\lambda\,v_0\,.\,[\![e_1]\!]\,(\lambda\,v_1\,.\,v_0\,v_1\,\kappa)) \end{array}$$

Figure 1: Plotkin's call-by-value, left-to-right CPS transformation

The simplicity of this transformation is, however, deceiving. At least three things happen in the CPS transformation: Intermediate values are named, continuations are introduced forcing terms into tail-recursive form, and terms are sequenced [1, 9, 14, 16, 29, 33]. Our thesis is that the CPS transformation is in fact a *compound* transformation. We propose to *stage* the mapping between the DS world and the CPS world.

We stage the mapping between DS and CPS terms as follows. First we restrict the DS and CPS languages so that continuations can be introduced and eliminated without disturbing the structure of the source term. The restricted languages are called CoreDS and CoreCPS, respectively. Then we define a transformation between ordinary DS and CoreDS, and another transformation between ordinary CPS and CoreCPS. Thus the CPS and DS transformations are staged as follows:

$$\begin{tikzcd} \mathrm{DS} \arrow[r, shift left] & \mathrm{CoreDS} \arrow[l, shift left] \arrow[r, shift left] & \mathrm{CoreCPS} \arrow[l, shift left] \arrow[r, shift left] & \mathrm{CPS} \arrow[l, shift left] \end{tikzcd}$$

Our staged transformation shows a number of useful properties:

---

[*] Bloomington, Indiana 47405, USA. Part of this work was supported by NSF under grant CCR-9000597.

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

- The *sequencing order* (*i.e.*, the order in which independent sub-expressions should be evaluated) is isolated from the introduction and elimination of continuations.
- The introduction and elimination of continuations is carried out with *local*, *structure-preserving*, and *reversible* rewrite rules.
- The staging isolates the fundamental CPS idea of *naming intermediate values* and identifies it with the *administrative reductions* of the DS transformation. Symmetrically, it isolates the DS idea of naming intermediate continuations and identifies it with the administrative reductions of the CPS transformation [29].

The rest of this paper is organized as follows. Section 2 stages the CPS transformation of pure λ-terms. This staging is extended to a Scheme-like language in Section 3. Section 4 addresses naming and unnaming DS terms while Section 5 addresses unnaming and naming CPS terms. In particular, Section 5.2 specifies how to extract tree-like context information out of a string-like continuation. The mapping between the DS and the CPS worlds scales up to handle first-class continuations, as described in Section 6. We illustrate the full transformation with an example in Section 7. Finally we conclude, raise some new issues, and present applications.

2 CPS transformation with local and structure-preserving rewrite rules

We first derive a local and structure-preserving CPS transformation for the λ-calculus. To this end we first restrict the DS λ-calculus to a subset on which the traditional CPS algorithm can be re-expressed with local and structure-preserving rewrite rules. We call this restriction of the DS language, Core Direct Style (CoreDS). The image of the CPS transformation on CoreDS terms is a restriction of the CPS language, called Core Continuation-Passing Style (CoreCPS). To restore expressiveness, we then introduce a new special form to CoreDS language, and its counterpart to CoreCPS. The resulting languages continue to be interconvertible with local and structure-preserving rewrite rules.

2.1 Restricting the DS language

A practical transformation based on Plotkin's CPS-transformation (*cf.* Figure 1) operates in two steps: first a DS term is rewritten into CPS, and second the result is simplified using "administrative reductions". As a consequence, residual CPS-terms bear little resemblance to source DS-terms. The two steps of the CPS-transformation can be merged into one [1, 9, 38], but then the transformation is not expressed with local and structure-preserving rewrite rules.[1] The transformation still uses static reductions, even though they are merged with the construction of the result. These static reductions hide the net effect of the transformation.

The administrative reductions essentially amount to unfolding the applications of the κ's in Figure 1. This has the effect of flattening the components of sub-terms of an application. For example, under left-to-right sequencing-order, a DS term such as

$$(f\,x)(g\,x)$$

yields the term

$$\begin{array}{l}
\lambda\,k\,.\\
\quad(\lambda\,v_4\,.\\
\qquad(\lambda\,v_5\,.\,v_4\;v_5\;\lambda\,v_0\,.\\
\qquad\qquad\qquad\qquad(\lambda\,v_2\,.\\
\qquad\qquad\qquad\qquad\quad(\lambda\,v_3\,.\,v_2\;v_3\;\lambda\,v_1\,.\,v_0\;v_1\;k)\\
\qquad\qquad\qquad\qquad\quad\;x)\\
\qquad\qquad\qquad\qquad\;g)\\
\qquad\;x)\\
\quad\;f
\end{array}$$

that gets simplified into the CPS term

$$\lambda\,k\,.\,f\;x\;(\lambda\,v\,.\,g\;x\;(\lambda\,w\,.\,v\;w\;k))$$

On the other hand, if every sub-term of an application is either a variable or a λ-abstraction, the application is already flat. For example, a term such as

$$f\;(\lambda\,x\,.\,x)$$

yields the term

$$\lambda\,k\,.\,(\lambda\,v_0\,.\,(\lambda\,v_1\,.\,v_0\;v_1\;k)\;\lambda\,x\,.\,\lambda\,k\,.\,k\;x)\;f$$

that gets simplified into the CPS term

$$\lambda\,k\,.\,f\;(\lambda\,x\,.\,\lambda\,k\,.\,k\;x)\;k$$

having the same structure as the original. This observation suggests we restrict the DS language to a language where all sub-terms are either variables or lambda-abstractions.

Following Reynolds [30], we refer to variables and λ-abstractions, whose evaluation cannot loop, cause a side-effect, raise an exception, or provoke an error, as *trivial* terms.[2] All other terms are *serious*. This distinction between trivial and serious terms is reflected in Plotkin's CPS transformation. The continuation is applied to the result of transforming a trivial term, whereas the transformation of a serious term is applied to the continuation.

We thus define the following restriction of the DS language. We refer to this language as the CoreDS language for the lambda-calculus. An expression e in CoreDS is either trivial or serious, and sub-terms of a serious expression are trivial:

$$\begin{array}{rcl}
e & ::= & t \mid s\\
t & ::= & x \mid \lambda\,x\,.\,e\\
s & ::= & t_0\;t_1
\end{array}$$

On this language we can re-express Plotkin's CPS-transformation (including its administrative reductions) with a function $[.]$ for trivial terms and a function $\llbracket . \rrbracket$ for serious terms. The CPS counterpart of a DS term e is given by $\lambda\,k\,.\,\langle\![e]\!\rangle\;k$.

$$\begin{array}{rcl}
\langle\![t]\!\rangle\;k & = & k\;[t]\\
\langle\![s]\!\rangle\;k & = & \llbracket s \rrbracket\;k\\
[x] & = & x\\
[\lambda\,x\,.\,e] & = & \lambda\,x\,.\,\lambda\,k\,.\,\langle\![e]\!\rangle\;k\\
\llbracket t_0\;t_1 \rrbracket\;k & = & [t_0]\;[t_1]\;k
\end{array}$$

This transformation consists only of local and structure-preserving rewrite rules. In addition, it is stand-alone, since

---

[1] An observation due to Gordon Plotkin (personal communication to the second author, Stanford, California, July 1991).

[2] Trivial terms correspond to Plotkin's notion of *value* [29].

$$\frac{\vdash \acute{t} \rightarrow \grave{t}}{k \vdash \acute{t} \rightharpoonup k\,\grave{t}} \qquad \frac{k \vdash \acute{s} \Rightarrow \grave{s}}{k \vdash \acute{s} \rightharpoonup \grave{s}}$$

$$\vdash x \rightarrow x \qquad \frac{k \vdash \acute{e} \rightharpoonup \grave{e}}{\vdash \lambda\, x\,.\,\acute{e} \rightarrow \lambda\, x\,.\,\lambda\, k\,.\,\grave{e}}$$

$$\frac{\vdash \acute{t}_0 \rightarrow \grave{t}_0 \qquad \vdash \acute{t}_1 \rightarrow \grave{t}_1}{k \vdash \acute{t}_0\,\acute{t}_1 \Rightarrow \grave{t}_0\,\grave{t}_1\,k}$$

Figure 2: Restricted CPS transformation

$$\frac{\vdash \acute{t} \leftarrow \grave{t}}{k \vdash \acute{t} \leftharpoondown k\,\grave{t}} \qquad \frac{k \vdash \acute{s} \Leftarrow \grave{s}}{k \vdash \acute{s} \leftharpoondown \grave{s}}$$

$$\vdash x \leftarrow x \qquad \frac{k \vdash \acute{e} \leftharpoondown \grave{e}}{\vdash \lambda\, x\,.\,\acute{e} \leftarrow \lambda\, x\,.\,\lambda\, k\,.\,\grave{e}}$$

$$\frac{\vdash \acute{t}_0 \leftarrow \grave{t}_0 \qquad \vdash \acute{t}_1 \leftarrow \grave{t}_1}{k \vdash \acute{t}_0\,\acute{t}_1 \Leftarrow \grave{t}_0\,\grave{t}_1\,k}$$

Figure 3: Restricted DS transformation

it does not require any administrative reductions. So it is both simpler to use and simpler to understand.

For example, our earlier example

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

directly yields the CPS term

$$\lambda\, k\,.\,f\ (\lambda\, x\,.\,\lambda\, k\,.\,k\ x)\ k$$

without any administrative reductions.

Here is the BNF of the restricted CPS terms $\lambda\, k\,.\,e$ produced by this transformation:

$$\begin{array}{lcl} e & ::= & k\ t \mid s \\ t & ::= & x \mid \lambda\, x\,.\,\lambda\, k\,.\,e \\ s & ::= & t_0\ t_1\ k \end{array}$$

where k is a meta-variable representing the set of all continuation identifiers.[3] We call this language the CoreCPS language for the λ-calculus.

The transformation can be rewritten more expressively in a natural semantics-style, as displayed in Figure 2. We use Stoy's diacritical convention [35] of overlining DS terms with an acute accent ´ and CPS terms with a grave accent `. Given a continuation k, a CoreDS term $\acute{e}$ is translated into a CoreCPS term $\grave{e}$ whenever $k \vdash \acute{e} \rightharpoonup \grave{e}$. Correspondingly, a trivial DS term $\acute{t}$ is translated into a CPS term $\grave{t}$ whenever $\vdash \acute{t} \rightarrow \grave{t}$, and, given a continuation k, a serious DS term $\acute{s}$ is translated into a CPS term $\grave{s}$ whenever $k \vdash \acute{s} \Rightarrow \grave{s}$.

The resulting transformation is not only expressed with local and structure-preserving rewrite rules — it is also reversible, as shown by Figure 3. Given a continuation k, a CoreCPS term $\grave{e}$ is translated into a CoreDS term $\acute{e}$ whenever $k \vdash \acute{e} \leftharpoondown \grave{e}$. Correspondingly, a trivial CoreCPS term $\grave{t}$ is translated into a CoreDS term $\acute{t}$ whenever $\vdash \acute{t} \leftarrow \grave{t}$ and, given a continuation k, a CoreCPS term $\grave{s}$ is translated into a serious CoreDS term $\acute{s}$ whenever $k \vdash \acute{s} \Leftarrow \grave{s}$.

We capture this symmetry by merging the contents of Figures 2 and 3 in Figure 4.

$$\frac{\vdash \acute{t} \leftrightarrow \grave{t}}{k \vdash \acute{t} \rightleftharpoons k\,\grave{t}} \qquad \frac{k \vdash \acute{s} \Leftrightarrow \grave{s}}{k \vdash \acute{s} \rightleftharpoons \grave{s}}$$

$$\vdash x \leftrightarrow x \qquad \frac{k \vdash \acute{e} \rightleftharpoons \grave{e}}{\vdash \lambda\, x\,.\,\acute{e} \leftrightarrow \lambda\, x\,.\,\lambda\, k\,.\,\grave{e}}$$

$$\frac{\vdash \acute{t}_0 \leftrightarrow \grave{t}_0 \qquad \vdash \acute{t}_1 \leftrightarrow \grave{t}_1}{k \vdash \acute{t}_0\,\acute{t}_1 \Leftrightarrow \grave{t}_0\,\grave{t}_1\,k}$$

Figure 4: Restricted DS and CPS inter-transformations

2.2 Restoring expressiveness

The CoreDS and CoreCPS languages do allow inverse CPS- and DS-transformations with local and structure-preserving rewrite rules, but these languages cannot directly express intermediate results, such as nested function calls. Let us now restore the expressiveness of the λ-calculus. Essentially, the CPS-transformation *names* intermediate results [1, Section 1.1]. Observing that a name (*i.e.*, a variable) is a trivial term, we only need a special form for naming intermediate results, with the constraint that it can be transformed into CPS with a local and structure-preserving rewrite rule.

We choose to extend the BNF of serious CoreDS terms with a bind-expression:

$$s\ ::=\ \ldots \mid \mathrm{bind}\ (x_1,\ \ldots,\ x_n) = (s_1,\ \ldots,\ s_n)\ \mathrm{in}\ e$$

A bind-expression provides a name x_i for the result of evaluating a non-trivial term s_i that may occur as an immediate sub-term in e. For example, a DS term such as

$$(f\ x)(g\ x)$$

yields the term

$$\mathrm{bind}\ (v,\ w) = (f\ x,\ g\ x)\ \mathrm{in}\ v\ w$$

Symmetrically, unfolding all the bind-expressions yields a DS term.

NB: In the rest of the paper, we use bind-expressions interchangeably with their higher-order abstract-syntax counterpart,

$$\mathrm{combine}\ (\lambda\, (x_1,\ \ldots,\ x_n)\,.\,e)\ (s_1,\ \ldots,\ s_n)$$

Correspondingly, we extend the BNF of serious CoreCPS terms with a schedule-expression:

---

[3] In the absence of control operators such as `call/cc`, one variable k is enough [8, 11, 31].

$$\begin{array}{lcl} s & ::= & \ldots \mid \text{schedule}\,(\lambda\,k\,.\,s_1,\,\ldots,\,\lambda\,k\,.\,s_n) \\ & & \qquad\qquad\quad (\lambda\,(x_1,\,\ldots,\,x_n)\,.\,e) \end{array}$$

A schedule-expression provides a name x_i for the result of performing a computation $\lambda\,k\,.\,s_i$. For example, a term such as

$$\text{bind}\,(v,\,w) = (f\,x,\,g\,x)\,\text{in}\;v\,w$$

yields the term

$$\begin{array}{l} \lambda\,k\,.\,\text{schedule}\,(\lambda\,k\,.\,f\,x\,k,\,\lambda\,k\,.\,g\,x\,k) \\ \qquad\qquad\quad (\lambda\,(v,\,w)\,.\,v\,w\,k) \end{array}$$

Choosing, *e.g.*, left-to-right sequencing-order then yields the CPS term

$$\lambda\,k\,.\,f\,x\,(\lambda\,v\,.\,g\,x\,(\lambda\,w\,.\,v\,w\,k))$$

Accordingly, we extend the inter-transformations of Figure 4 with the rule displayed in Figure 5. This rewrite rule is local and structure-preserving.

3 A Scheme-like language

We now generalize the above transformations to a more realistic DS language, by adding Scheme's constants, primitive operations, conditional expressions, and let- and letrec-expressions [4]. We also consider n-ary functions.

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

Constants are trivial. For simplicity we classify all the other new syntactic constructs as serious. We use continuation-passing primitive operators in the CPS world.

The CPS transformation is displayed in Figure 6. Here is the BNF of the CPS terms $\lambda\,k\,.\,e$ this transformation produces:

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

where v occurs once in $\lambda\,v\,.\,e$.

NB: Originally, we derived the BNF of CPS terms by induction over DS terms and over the possible values of κ, in Figure 6. Since then, we have verified it using Malmkjær's analysis that takes a two-level λ-term and produces the BNF of its result, based on abstract interpretation [22, 23]. Applying this analysis to the CPS transformer of Figure 6 yields the same BNF as the one derived by hand. We use Malmkjær's analysis again in the following section.

3.1 CoreDS and CoreCPS

Along the lines of Section 2.2, we now define CoreDS for the Scheme-like language. In CoreDS again, all sub-terms but the actual parameters of bind-expressions are trivial. A CoreDS term e is defined by the following BNF:

$$\begin{array}{lcl} e & ::= & t \mid s \\ t & ::= & c \mid i \mid l \\ s & ::= & t_0\,(t_1,\,\ldots,\,t_n) \mid op\,(t_1,\,\ldots,\,t_m) \\ & \mid & \text{cond}(t_1,\,e_2,\,e_3) \\ & \mid & \text{let}\,(i_1,\,\ldots,\,i_n) = (t_1,\,\ldots,\,t_n)\,\text{in}\;e \\ & \mid & \text{letrec}\,(i_1,\,\ldots,\,i_n) = (l_1,\,\ldots,\,l_n)\,\text{in}\;e \\ & \mid & \text{combine}\,(\lambda\,(x_1,\,\ldots,\,x_n)\,.\,e)\,(s_1,\,\ldots,\,s_n) \\ l & ::= & \lambda\,(i_1,\,\ldots,\,i_n)\,.\,e \end{array}$$

Symmetrically, a CoreCPS term for the Scheme-like language has the form $\lambda\,k\,.\,e$ and is defined by the following BNF:

$$\begin{array}{lcl} e & ::= & k\,t \mid s \\ t & ::= & c \mid i \mid l \\ s & ::= & t_0\,(t_1,\,\ldots,\,t_n,\,k) \mid op\,(t_1,\,\ldots,\,t_m,k) \\ & \mid & \text{cond}(t_1,\,e_2,\,e_3) \\ & \mid & \text{let}\,(i_1,\,\ldots,\,i_n) = (t_1,\,\ldots,\,t_n)\,\text{in}\;e \\ & \mid & \text{letrec}\,(i_1,\,\ldots,\,i_n) = (l_1,\,\ldots,\,l_n)\,\text{in}\;e \\ & \mid & \text{schedule}\,(\lambda\,k\,.\,s_1,\,\ldots,\,\lambda\,k\,.\,s_n)\,(\lambda\,(x_1,\,\ldots,\,x_n)\,.\,e) \\ l & ::= & \lambda\,(i_1,\,\ldots,\,i_n,\,k)\,.\,e \end{array}$$

As in Section 2.2, we can transform CoreDS terms into CoreCPS terms, using local and structure-preserving rewrite rules. These two transformations are exact inverses. They generalize Figure 5 and are presented in Figure 7.

NB: These two transformations (minus the rule for combine- and schedule-expressions) can be derived using Consel's partial evaluator [5]. Specializing the CPS transformation with respect to CoreDS yields one half of the transformation of Figure 7. Dually, specializing the DS transformation with respect to CoreDS yields the other half of the transformation of Figure 7.

3.2 Conclusion

To summarize, we have defined a CoreDS language as expressive as the ordinary DS language, for both the λ-calculus and the Scheme-like language. In CoreDS, the result of all serious subcomputations are named. Similarly, CoreCPS is as expressive as CPS. In CoreCPS, all intermediate continuations are named (by the parameters of expressions in the tuple argument to schedule). We have derived local, structure-preserving, and reversible transformations between the two languages.

Let us now turn to the transformations between DS and CoreDS, and between CPS and CoreCPS. These transformations are applicable to both the λ-calculus and the Scheme-like language. They are described in the next two sections.

4 DS and CoreDS

In CoreDS, serious subterms must be named, whereas in DS they may occur anywhere. Thus the transformations between these two languages simply amount to introducing or eliminating names for serious expressions. Both transformations preserve the implicit sequencing order of DS. This section is written in the spirit of the programming language Scheme, where sequencing order is unspecified [4]. Further refinements may be possible in a language that specifies a sequencing order, as, *e.g.*, in Standard ML [25].

$$\begin{tikzcd} \text{DS} \arrow[r, shift left, "\mathcal{N}_d"] & \text{CoreDS} \arrow[l, shift left, "\mathcal{U}_d"] \end{tikzcd}$$

$$\frac{k \vdash \acute{e} \rightleftharpoons \grave{e} \qquad k \vdash \acute{s}_1 \Leftrightarrow \grave{s}_1 \quad \ldots \quad k \vdash \acute{s}_n \Leftrightarrow \grave{s}_n}{k \vdash \text{combine}\,(\lambda\,(v_1,\,\ldots,\,v_n)\,.\,\acute{e})\,(\acute{s}_1,\,\ldots,\,\acute{s}_n) \Leftrightarrow \text{schedule}\,(\lambda\,k\,.\,\grave{s}_1,\,\ldots,\,\lambda\,k\,.\,\grave{s}_n)\,(\lambda\,(v_1,\,\ldots,\,v_n)\,.\,\grave{e})}$$

Figure 5: Extended DS and CPS inter-transformations

$$\begin{array}{rcl}
\llbracket c \rrbracket\,\kappa & = & \kappa\,c \\
\llbracket i \rrbracket\,\kappa & = & \kappa\,i \\
\llbracket \lambda\,(i_1,\,\ldots,\,i_n)\,.\,e \rrbracket\,\kappa & = & \kappa\,(\lambda\,(i_1,\,\ldots,\,i_n,k)\,.\,\llbracket e \rrbracket\,k) \\
\llbracket e_0\,(e_1,\,\ldots,\,e_n) \rrbracket\,\kappa & = & \llbracket e_0 \rrbracket\,(\lambda\,v_0\,.\,\llbracket e_1 \rrbracket\,(\lambda\,v_1\,.\,\ldots\llbracket e_n \rrbracket\,(\lambda\,v_n\,.\,v_0\,(v_1,\,\ldots,\,v_n,\kappa)))) \\
\llbracket \acute{o}p\,(e_1,\,\ldots,\,e_m) \rrbracket\,\kappa & = & \llbracket e_1 \rrbracket\,(\lambda\,v_1\,.\,\ldots\llbracket e_m \rrbracket\,(\lambda\,v_m\,.\,(\grave{o}p\,(v_1,\,\ldots,\,v_m,\kappa)))) \\
\llbracket \text{cond}(e_0,\,e_1,\,e_2) \rrbracket\,\kappa & = & \llbracket e_0 \rrbracket\,(\lambda\,v_0\,.\,\text{let}\ k = \kappa\ \text{in}\ \text{cond}(v_0,\,\llbracket e_1 \rrbracket\,k,\,\llbracket e_2 \rrbracket\,k) \\
\llbracket \text{let}\ (i_1,\,\ldots,\,i_n) = (e_1,\,\ldots,\,e_n)\ \text{in}\ e \rrbracket\,\kappa & = & \llbracket e_1 \rrbracket\,(\lambda\,v_1\,.\,\ldots\llbracket e_n \rrbracket\,(\lambda\,v_n\,.\,\text{let}\ (i_1,\,\ldots,\,i_n) = (v_1,\,\ldots,\,v_n)\ \text{in}\ \llbracket e \rrbracket\,\kappa)) \\
\llbracket \text{letrec}\ (f_1,\,\ldots) = (\lambda\,(i_1,\,\ldots,\,i_m)\,.\,e_1,\,\ldots)\ \text{in}\ e \rrbracket\,\kappa & = & \text{letrec}\ (f_1,\,\ldots) = (\lambda\,(i_1,\,\ldots,\,i_m,k)\,.\,\llbracket e_1 \rrbracket\,k,\,\ldots)\ \text{in}\ \llbracket e \rrbracket\,\kappa
\end{array}$$

where k is a fresh variable. A DS term e is transformed into CPS as $\lambda\,k\,.\,\llbracket e \rrbracket\,k$.

Figure 6: The call-by-value CPS transformation

$$\frac{\vdash \acute{t} \leftrightarrow \grave{t}}{k \vdash \acute{t} \rightleftharpoons k\,\grave{t}} \qquad \frac{k \vdash \acute{s} \Leftrightarrow \grave{s}}{k \vdash \acute{s} \rightleftharpoons \grave{s}}$$

$$\vdash \acute{c} \leftrightarrow \grave{c} \qquad \vdash \acute{i} \leftrightarrow \grave{i} \qquad \frac{k \vdash \acute{e} \rightleftharpoons \grave{e}}{\vdash \lambda\,(i_1,\,\ldots,\,i_n)\,.\,\acute{e} \leftrightarrow \lambda\,(i_1,\,\ldots,\,i_n,k)\,.\,\grave{e}}$$

$$\frac{\vdash \acute{t}_0 \leftrightarrow \grave{t}_0 \quad \ldots \quad \vdash \acute{t}_n \leftrightarrow \grave{t}_n}{k \vdash \acute{t}_0\,(\acute{t}_1,\,\ldots,\,\acute{t}_n) \Leftrightarrow \grave{t}_0\,(\grave{t}_1,\,\ldots,\,\grave{t}_n,\,k)} \qquad \frac{\vdash \acute{t}_0 \leftrightarrow \grave{t}_0 \quad \ldots \quad \vdash \acute{t}_m \leftrightarrow \grave{t}_m}{k \vdash \acute{o}p\,(\acute{t}_1,\,\ldots,\,\acute{t}_m) \Leftrightarrow \grave{o}p\,(\grave{t}_1,\,\ldots,\,\grave{t}_m,k)}$$

$$\frac{\vdash \acute{t}_1 \leftrightarrow \grave{t}_1 \qquad k \vdash \acute{e}_2 \rightleftharpoons \grave{e}_2 \qquad k \vdash \acute{e}_3 \rightleftharpoons \grave{e}_3}{k \vdash \text{cond}(\acute{t}_1,\,\acute{e}_2,\,\acute{e}_3) \Leftrightarrow \text{cond}(\grave{t}_1,\,\grave{e}_2,\,\grave{e}_3)}$$

$$\frac{\vdash \acute{t}_0 \leftrightarrow \grave{t}_0 \quad \ldots \quad \vdash \acute{t}_n \leftrightarrow \grave{t}_n \qquad k \vdash \acute{e} \rightleftharpoons \grave{e}}{k \vdash \text{let}\ (i_1,\,\ldots,\,i_n) = (\acute{t}_1,\,\ldots,\,\acute{t}_n)\ \text{in}\ \acute{e} \Leftrightarrow \text{let}\ (i_1,\,\ldots,\,i_n) = (\grave{t}_1,\,\ldots,\,\grave{t}_n)\ \text{in}\ \grave{e}}$$

$$\frac{\vdash \acute{l}_0 \leftrightarrow \grave{l}_0 \quad \ldots \quad \vdash \acute{l}_n \leftrightarrow \grave{l}_n \qquad k \vdash \acute{e} \rightleftharpoons \grave{e}}{k \vdash \text{letrec}\ (i_1,\,\ldots,\,i_n) = (\acute{l}_1,\,\ldots,\,\acute{l}_n)\ \text{in}\ \acute{e} \Leftrightarrow \text{letrec}\ (i_1,\,\ldots,\,i_n) = (\grave{l}_1,\,\ldots,\,\grave{l}_n)\ \text{in}\ \grave{e}}$$

$$\frac{k \vdash \acute{e} \rightleftharpoons \grave{e} \qquad k \vdash \acute{s}_1 \Leftrightarrow \grave{s}_1 \quad \ldots \quad k \vdash \acute{s}_n \Leftrightarrow \grave{s}_n}{k \vdash \text{combine}\,(\lambda\,(v_1,\,\ldots,\,v_n)\,.\,\acute{e})\,(\acute{s}_1,\,\ldots,\,\acute{s}_n) \Leftrightarrow \text{schedule}\,(\lambda\,k\,.\,\grave{s}_1,\,\ldots,\,\lambda\,k\,.\,\grave{s}_n)\,(\lambda\,(v_1,\,\ldots,\,v_n)\,.\,\grave{e})}$$

Figure 7: Inter-translation between CoreDS and CoreCPS

4.1 Naming DS terms

The transformation from DS to CoreDS, $\mathcal{N}_d$, maps a compound term in which some immediate subterms are serious into a bind-expression naming all these serious subterms. Replacing the serious terms by their names yields the body of the bind-expression. For example,

$$\begin{aligned}\mathcal{N}_d((f\ x)\ (g\ x)) &= \text{bind}\ (v,\ w) = (\mathcal{N}_d(f\ x),\ \mathcal{N}_d(g\ x)) \\ &\ \text{in}\ v\ w \\ &\equiv \text{combine}\ (\lambda\,(v,\ w)\,.\,v\ w) \\ &\qquad\qquad (\mathcal{N}_d(f\ x),\ \mathcal{N}_d(g\ x))\end{aligned}$$

Naming all the serious subterms in a single bind-expression preserves the sequencing order of the original term. The formals of the bind-expressions introduced by $\mathcal{N}_d$ are fresh names, which makes it difficult to compare CoreDS terms. For simplicity we define equality on CoreDS terms to be equality up to α-equivalence of the formal parameters of bind-expressions. This definition of equality of CoreDS terms makes $\mathcal{N}_d$ into a function. In practice, CoreDS terms can be compared by simultaneously substituting fresh variables for all the formal parameters of bind-expressions [34].

4.2 Unnaming CoreDS terms

Unnaming a CoreDS term with $\mathcal{U}_d$ corresponds to performing the administrative reductions of the DS transformation. For each bind-expression, we either unfold all the bindings, substituting each named term for its name, or we residualize the entire bind-expression as a let-expression. Our strategy is similar to that used in partial evaluation [3].

We first consider the image of $\mathcal{N}_d$. Because there are no bind-expressions in DS, $\mathcal{N}_d$ never produces a bind-expression where the body is another bind-expression. Additionally, each name bound by a bind-expression occurs exactly once, specifically as one of the t's in the BNF production of the body. If an expression denotes a CoreDS term in the image of $\mathcal{N}_d$, mapping it back to DS amounts to unfolding every bind-expression. On such terms, naming and unnaming are inverses.

In addition to the terms in the image of $\mathcal{N}_d$, $\mathcal{U}_d$ must accommodate terms that result from transforming CPS terms into CoreDS. Thus we extend it to the full BNF of CoreDS terms. To maintain the sequencing order, we simply residualize as a let-expression any bind-expression that is not in the image of the transformation from DS to CoreDS. For example,

$$\text{bind}\ v = f\ x\ \text{in}\ g\ x$$

is transformed into the DS term

$$\text{let}\ v = f\ x\ \text{in}\ g\ x$$

More precisely, $\mathcal{U}_d$ unfolds a bind-expression whenever

1. The body is an application, a primitive operation, a conditional expression, a let-expression, or, degenerately, an identifier and,
2. Each of the names bound by the bind-expression occurs exactly once and as one of the trivial subterms of the body.

Otherwise the bind-expression is residualized into a let-expression.

4.3 A Galois connection

We now examine the relationship between $\mathcal{N}_d$ and $\mathcal{U}_d$. While the transformations are not inverses on all terms, there are some terms, called *normalized* terms on which they are inverses. For each language we give a semantics-preserving mapping from terms to normalized terms. Then, we show that the partial orderings generated by these functions are related as a Galois connection.

Naming and then unnaming a DS term produces the original DS term, but the converse is not true. As described in the previous section, a bind-expression in the CoreDS term may be residualized into a let-expression in the DS term. These let-expressions remain let-expressions when translating back to CoreDS. These problematic CoreDS terms are not in the image of $\mathcal{N}_d$.

$$\begin{tikzcd} \text{DS} \arrow[r, shift left, "\mathcal{N}_d"] & \text{CoreDS} \arrow[l, shift left, "\mathcal{U}_d"] \end{tikzcd}$$

More formally, $\mathcal{N}_d$ and $\mathcal{U}_d$ are related as follows:

$$\begin{cases} \mathcal{U}_d \circ \mathcal{N}_d = \mathit{Identity}_{\text{DS}} \\ \mathcal{U}_d \circ \mathcal{N}_d \circ \mathcal{U}_d = \mathcal{U}_d \end{cases}$$

We can trivially define a mapping $\mathcal{S}_{\text{DS}}$ from DS terms to normalized DS terms, because the transformations are inverses on all DS terms. Hence we define

$$\mathcal{S}_{\text{DS}}[\![e]\!] = e$$

Although $\mathcal{U}_d$ and $\mathcal{N}_d$ are not inverses on all CoreDS terms, the following equation

$$\mathcal{N}_d \circ \mathcal{U}_d \circ \mathcal{N}_d \circ \mathcal{U}_d = \mathcal{N}_d \circ \mathcal{U}_d$$

which can be easily verified using the above equations, shows that they are inverses on terms that have been unnamed and then named again. Thus we can define a mapping $\mathcal{S}_{\text{CoreDS}}$ from all CoreDS terms into the set of normalized CoreDS terms, as follows

$$\mathcal{S}_{\text{CoreDS}}[\![e]\!] = \mathcal{N}_d(\mathcal{U}_d(e))$$

This mapping can be described inductively over CoreDS terms. We first give the rules for a bind-expression:

1. Whenever $\mathcal{U}_d$ residualizes bind $(x_1,\ ...) = (s_1,\ ...)$ in e as a let-expression,

$$\begin{aligned}&\mathcal{S}_{\text{CoreDS}}[\![\text{bind}\ (x_1,\ ...) = (s_1,\ ...)\ \text{in}\ e]\!] = \\ &\quad \text{bind}\ (y_1,\ ...) = (\mathcal{S}_{\text{CoreDS}}[\![s_1]\!],\ ...) \\ &\quad \text{in let}\ (x_1,\ ...) = (y_1,\ ...)\ \text{in}\ \mathcal{S}_{\text{CoreDS}}[\![e]\!]\end{aligned}$$

2. Whenever $\mathcal{U}_d$ unfolds bind $(x_1,\ ...) = (s_1,\ ...)$ in e, and e is not an identifier,

$$\begin{aligned}&\mathcal{S}_{\text{CoreDS}}[\![\text{bind}\ (x_1,\ ...) = (s_1,\ ...)\ \text{in}\ e]\!] = \\ &\quad \text{bind}\ (x_1,\ ...) = (\mathcal{S}_{\text{CoreDS}}[\![s_1]\!],\ ...)\ \text{in}\ \mathcal{S}_{\text{CoreDS}}[\![e]\!]\end{aligned}$$

3. Finally,

$$\mathcal{S}_{\text{CoreDS}}[\![\text{bind}\ x = s\ \text{in}\ x]\!] = \mathcal{S}_{\text{CoreDS}}[\![s]\!]$$

The rules for other terms are defined by straightforward induction over the structure of the term, as in the second case above.

Using the mappings $\mathcal{S}_{\mathrm{DS}}$ and $\mathcal{S}_{\mathrm{CoreDS}}$ we may define the partial orders $\sqsubseteq_{\mathrm{DS}}$ and $\sqsubseteq_{\mathrm{CoreDS}}$ on DS and CoreDS terms respectively. These partial orders relate terms to semantically-equivalent normalized terms, and are defined as follows. Again we use Stoy's diacritical convention of overlining DS terms with an acute accent ´ and CoreDS terms with a grave accent `.

$$\begin{cases} \acute{e}_1 \sqsubseteq_{\mathrm{DS}} \acute{e}_2 & \Leftrightarrow \quad \acute{e}_2 = \acute{e}_1 \\ \grave{e}_1 \sqsubseteq_{\mathrm{CoreDS}} \grave{e}_2 & \Leftrightarrow \quad \grave{e}_2 = \mathcal{N}_d(\mathcal{U}_d(\grave{e}_1)) \end{cases}$$

Together these two equations imply a Galois connection [24]:

$$\acute{e}_1 \sqsubseteq_{\mathrm{DS}} \mathcal{U}_d(\grave{e}_2) \quad \Leftrightarrow \quad \mathcal{N}_d(\acute{e}_1) \sqsupseteq_{\mathrm{CoreDS}} \grave{e}_2$$

4.4 Conclusion

CoreDS was introduced as an intermediate language where the result of every serious expression is explicitly named, on the motivation that, given a CoreDS term, continuations can be introduced with local and structure-preserving rewrite rules. In this section, we have described the mappings $\mathcal{N}_d$ and $\mathcal{U}_d$ between plain DS terms and CoreDS terms. Several CoreDS terms may encode a single DS term. We have characterized this flexibility with a Galois connection, and have specified the corresponding partial orders on DS and on CoreDS terms.

5 CoreCPS and CPS

In CoreCPS, intermediate continuations must be named, whereas in CPS they may occur anonymously. Thus the transformations between these two languages simply amount to introducing or reducing redexes that name continuations. Because the sequencing order of CPS is explicit, while that of CoreCPS is implicit in the definition of schedule, some care must be taken in the transformation from CPS to CoreCPS to preserve the intent of the CPS term.

$$\begin{tikzcd} \text{CoreCPS} \arrow[r, shift left, "\mathcal{U}_c"] & \text{CPS} \arrow[l, shift left, "\mathcal{N}_c"] \end{tikzcd}$$

5.1 Unnaming Core CPS terms

The transformation $\mathcal{U}_c$ unnames CoreCPS terms by (1) choosing a sequencing order and simplifying the schedule-expressions, and (2) by performing administrative reductions over the resulting CPS term [29].

Choosing a sequencing order amounts to selecting a definition of schedule. The following function schedule$_2$ sequences two computations from left to right:

$$\mathrm{schedule}_2 \;=\; \lambda\,(c_1,\, c_2)\,.\,\lambda\,\kappa\,.\,c_1\,(\lambda\,v_1\,.\,c_2\,(\lambda\,v_2\,.\,\kappa\,(v_1,\, v_2)))$$

Simplifying a schedule-expression amounts to moving its semantics into the syntax of the CPS term.

5.2 Naming CPS terms

The primary task of the DS transformation is to encode a flat, tail-recursive term into a tree-like (in general non-tail recursive) term. This mapping from a flat, string-like term into a tree is reminiscent of parsing a concrete-syntax string into an abstract-syntax tree. Indeed, the analogy holds for CPS terms that encode the sequencing order of the DS language, *e.g.*, left-to-right as in SML. For example, the CPS term

$$\lambda\,k\,.\,f\,(x,\,\lambda\,v_0\,.\,g\,(x,\,\lambda\,v_1\,.\,v_0\,(v_1,\,\lambda\,v_2\,.\,h\,(x,\,\lambda\,v_3\,.\,v_2\,(v_3,\,k)))))$$

straightforwardly yields

$$((f\ x)(g\ x))\ (h\ x)$$

Terms that do not encode the sequencing order or that may contain side-effects or control-effects require a more detailed analysis, particularly if the DS language is Scheme, where the sequencing order of subexpressions is unspecified. For example, the term

$$\lambda\,k\,.\,f\,(x,\,\lambda\,v_0\,.\,h\,(x,\,\lambda\,v_3\,.\,g\,(x,\,\lambda\,v_1\,.\,v_0\,(v_1,\,\lambda\,v_2\,.\,v_2\,(v_3,\,k)))))$$

should yield

$$\lambda\,x\,.\,\mathrm{let}\ v_0 = f\ x\ \mathrm{in}\ \mathrm{let}\ v_3 = h\ x\ \mathrm{in}\ (v_0(g\ x))\ v_3$$

We conjecture, however, that the sequencing order of most CPS terms is overspecified, and thus for most terms the straightforward tree-building algorithm applies. In any case, an effect analysis would come in handy here.

5.2.1 Canonically-sequentialized CPS terms

The task in naming a CPS term is to reconstruct the tree structure of the original DS term, hiding the sequencing order in the definition of the schedule operator. A CPS term essentially encodes a postfix traversal of this tree [7]. Each expression is evaluated tail-recursively and yields a value that is denoted by the actual parameter of the continuation. Once a parameter is used in a term to yield a new value, it is never used again. Thus the problem of constructing a tree from a CPS expression is analogous to the problem of parsing a program written in a postfix language. In particular, independent subexpressions in a CPS term can be naturally detected using a stack.

The transformation $\mathcal{N}_c$ traverses the continuation, pushing the temporaries (*i.e.*, the continuation parameters) on a stack. At consumption points (*i.e.*, where the continuation parameters occur), we pop the stack and match the temporaries according to the sequencing order of the target DS language. For example, if the target language is SML where the sequencing order is left-to-right, then a prefix of the stack must equal the sequence of the temporaries of the expression. On the other hand, if the target language is Scheme where the sequencing order is unspecified, then we only require that the temporaries in the expression be equal to the set of temporaries on top of the stack.

In a CPS term that encodes the sequencing order of the DS language canonically, the temporaries on the top of the stack are always the ones required. The evaluation of the corresponding expressions does not need to be explicitly sequenced.

5.2.2 Other CPS terms

A CPS term is not, however, constrained to reflect precisely a postfix traversal of some tree. Temporaries need not be declared in a last-in, first-out fashion. If a subexpression is evaluated early, then the CPS term reflects a sequencing constraint that should be preserved in the CoreCPS term. In this case the temporaries at the top of the stack are not the temporaries required for the current expression, since one denotes the subexpression evaluated early. In that situation the expressions denoted by all the variables lower on the stack must be explicitly sequenced in the CoreCPS result.

5.2.3 Computational effects

Side effects and control effects do not raise particular problems in a language where sequencing order is fixed. In a language such as Scheme, however, some kind of effect analysis is needed to avoid assuming that every expression has a computational effect — otherwise we would have to translate a term such as

$$\lambda\,(x,\,k)\,.\,f\;(x,\,\lambda\,v_0\,.\,g\;(x,\,\lambda\,v_1\,.\,v_0\;(v_1,\,k)))$$

into

$$\lambda\,x\,.\,\mathrm{let}\;v_0 = f\;x\;\mathrm{in}\;v_0\;(g\;x)$$

instead of

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

Given safe effect information, a CPS term can be staged using the above algorithm, modified to sequentialize subterms that may contain effects.

5.3 A Galois connection

We now examine to what extent $\mathcal{N}_c$ and $\mathcal{U}_c$ are inverses. For each language we give a semantics-preserving mapping from terms to normalized terms. Then, we show that the partial orderings generated by these functions are related as a Galois connection.

As in the case of the naming and unnaming of DS terms (*cf.* Section 4.3), the naming and unnaming of CPS terms are not inverse transformations. In fact the transformations are not inverses on either CoreCPS or CPS terms.

$$\begin{tikzcd}
\mathrm{CoreCPS} \arrow[r, shift left, "\mathcal{U}_c"] & \mathrm{CPS} \arrow[l, shift left, "\mathcal{N}_c"]
\end{tikzcd}$$

Naming and unnaming are not inverse transformations on CPS terms because let-expressions may name continuations that are used only once. $\mathcal{N}_c$ transforms all continuation-naming let-expressions into schedule-expressions. When the term is transformed back into CoreCPS a continuation-naming let-expression is introduced only if the continuation identifier occurs more than once. Thus the transformations are not inverses on CPS terms containing useless let-expressions. Because $\mathcal{U}_c$ does not produce these useless let-expressions, the transformations are inverses on the result of unnaming CoreCPS terms.

The transformations are not inverses on CoreCPS terms because $\mathcal{U}_c$ moves nested let- and letrec-expressions outward. Because $\mathcal{N}_c$ does not create nested let- and letrec-expressions, the transformations are inverses on the result of naming CPS terms.

More formally, $\mathcal{N}_c$ and $\mathcal{U}_c$ are related as follows:

$$\begin{cases} \mathcal{N}_c \circ \mathcal{U}_c \circ \mathcal{N}_c & = & \mathcal{N}_c \\ \mathcal{U}_c \circ \mathcal{N}_c \circ \mathcal{U}_c & = & \mathcal{U}_c \end{cases}$$

These equations imply the following:

$$\begin{cases} \mathcal{U}_c \circ \mathcal{N}_c \circ \mathcal{U}_c \circ \mathcal{N}_c & = & \mathcal{U}_c \circ \mathcal{N}_c \\ \mathcal{N}_c \circ \mathcal{U}_c \circ \mathcal{N}_c \circ \mathcal{U}_c & = & \mathcal{N}_c \circ \mathcal{U}_c \end{cases}$$

which say that the transformations are inverses on terms that have been named and unnamed once. We can thus define the functions $\mathcal{S}_{\mathrm{CPS}}$ and $\mathcal{S}_{\mathrm{CoreCPS}}$, which normalize CPS and CoreCPS terms respectively.

$$\begin{cases} \mathcal{S}_{\mathrm{CPS}}[\![e]\!] & = & \mathcal{U}_c(\mathcal{N}_c(e)) \\ \mathcal{S}_{\mathrm{CoreCPS}}[\![e]\!] & = & \mathcal{N}_c(\mathcal{U}_c(e)) \end{cases}$$

$\mathcal{S}_{\mathrm{CPS}}$ and $\mathcal{S}_{\mathrm{CoreCPS}}$ can be defined inductively over source terms using the observations outlined above. $\mathcal{S}_{\mathrm{CoreCPS}}$ must be defined relative to a fixed sequencing order.

Using the mappings $\mathcal{S}_{\mathrm{CPS}}$ and $\mathcal{S}_{\mathrm{CoreCPS}}$ we may define the partial orders $\sqsubseteq_{\mathrm{CPS}}$ and $\sqsubseteq_{\mathrm{CoreCPS}}$ on CPS and CoreCPS terms respectively. These partial orders relate terms to semantically-equivalent normalized terms, and are defined as follows. Again we use Stoy's diacritical convention of overlining CPS terms with an acute accent ´ and CoreCPS terms with a grave accent `.

$$\begin{cases} \acute{e}_1 \sqsubseteq_{\mathrm{CPS}} \acute{e}_2 & \Leftrightarrow & \acute{e}_2 = \mathcal{U}_c(\mathcal{N}_c(\acute{e}_1)) \\ \grave{e}_1 \sqsubseteq_{\mathrm{CoreCPS}} \grave{e}_2 & \Leftrightarrow & \grave{e}_2 = \mathcal{N}_c(\mathcal{U}_c(\grave{e}_1)) \end{cases}$$

Together these two equations imply a Galois connection:

$$\acute{e}_1 \sqsubseteq_{\mathrm{CPS}} \mathcal{U}_d(\grave{e}_2) \;\Leftrightarrow\; \mathcal{N}_d(\acute{e}_1) \sqsupseteq_{\mathrm{CoreCPS}} \grave{e}_2$$

5.4 Conclusion

The administrative reductions of the full CPS transformation make it difficult to predict the shape of the result of transforming a DS expression. We have isolated these administrative reductions in the transformation from CoreCPS to CPS. The definition of the partial order on CoreCPS terms shows how the administrative reductions affect the shape of a term.

The partial order on CPS terms reflects a mismatch between how contexts may be specified in CPS and how they are specified in DS. Thus there is some flexibility in the relationship between CoreCPS and CPS terms. We have characterized this flexibility as a Galois connection.

5.5 Caveat

This particular partial order may not be the most appropriate one, though, because the introduction and elimination of continuations is not order-preserving. For example,

$$\begin{array}{lcl} \mathrm{bind}\;v = f\;x & \sqsubseteq_{\mathrm{CoreDS}} & \mathrm{bind}\;v' = f\;x \\ \mathrm{in\;bind}\;w = g\;x & & \quad \mathrm{in\;let}\;v = v' \\ \quad \mathrm{in}\;v\;w & & \qquad \mathrm{in\;bind}\;w = g\;x \\ & & \qquad\quad \mathrm{in}\;v\;w \end{array}$$

but introducing continuations in these two terms as specified by Figure 7 does not yield two terms that are related by the partial order $\sqsubseteq_{\mathrm{CoreCPS}}$. We are currently looking for partial orders that would be preserved by the introduction and elimination of continuations.

6 First-Class Continuations (outline)

In an earlier work [11], we describe how to handle first-class continuations in the DS transformation. The technique scales up to the staged transformation. Briefly stated, we instrument the BNF of CPS terms with the set of all the continuation identifiers that are lexically visible, and we relax the constraint that only the current continuation be applied. Instead we let any continuation identifier be applied. Then we analyze CPS terms to determine whether a continuation k is used exceptionally in a term e. We then need to distinguish terms where continuations are declared. Any declaration of a continuation that is used exceptionally is translated into a `call/cc`. Conversely, any application of a continuation that is not the current one is translated into a `throw`. Correspondingly, we instrument the BNF of DS terms with the set of all first-class continuation identifiers that are lexically visible, and we enforce that only a continuation identifier can be thrown values.

7 A complete example

We consider the binary tree specified by the data type in Figure 8. This declaration and the corresponding control structure `caseType` are inspired by SML and used in Consel's partial evaluator Schism [5]. We are using them for readability. Also for readability, we have maintained direct-style primitive operators in Figures 11 and 12.

Let us determine whether a given tree of positive numbers "weighs" more than a given number. Rather than first computing the weight of the tree and then comparing this weight with the number, we combine the comparison with computation of the weight, using `call/cc` to abort the computation as soon as the weight exceeds the number. This leads to more comparisons on average but to a potentially shorter tree traversal. The corresponding direct-style program is given in Figure 9.

Let us transform this program into CoreDS by naming all intermediate (*i.e.*, temporary) values. The result is given in Figure 10.

We now turn this program into CoreCPS. Thanks to bind, the evaluation steps are explicit and we can express them as continuation abstractions, using a schedule expression to sequentialize these abstractions. The resulting program is given in Figure 11.

Finally, we can choose *e.g.*, left-to-right sequencing order to transform the CoreCPS program into CPS. The resulting CPS program is given in Figure 12.

Each step is reversible. First, continuations are named and continuation extensions (and thus sequencing order) are abstracted, thereby going from the CPS program of Figure 12 to the CoreCPS program of Figure 11. Second, continuation abstractions are turned into evaluations of expressions, thus going from the CoreCPS program of Figure 11 to the CoreDS program of Figure 10. Third, expressions are inlined (provided the order of their evaluation does not conflict), leading from the CoreDS program of Figure 10 to the DS program of Figure 9.

8 Conclusion

CPS is generally agreed to be useful but it overcommits a program to a particular sequentialization. We have factored out this commitment by separating stages in the CPS transformation:

Proposition 1 (CPS transformation) *Naming serious subterms in a DS term, introducing continuations in the resulting CoreDS term, choosing a sequencing order, and unnaming the resulting CoreCPS term amount to transforming the DS term into CPS with respect to the same sequencing order.*

Proof: By composition. Unnaming the CoreCPS term amounts to performing the administrative reductions of the CPS transformation. □

Proposition 2 (DS transformation) *Naming continuations in a CPS term, eliminating continuations in the resulting CoreCPS term, and unnaming the resulting CoreDS term amount to transforming the CPS term into DS, respecting the sequencing order.*

Proof: By composition. Unnaming the CoreDS term amounts to performing the administrative reductions of the DS transformation. □

Fundamentally, we have isolated the heart of the CPS and of the DS transformations — *i.e.*, the introduction and elimination of continuations — with local, structure-preserving, and reversible rewrite rules.

The CPS and the DS transformations looked complicated because of red tape — the administrative reductions [29]. We have isolated this red tape and characterized it with two Galois connections: naming and unnaming of DS and CPS terms.

$$\begin{tikzcd}
\text{DS} \arrow[r, shift left, "\text{naming values}"] & \text{Core DS} \arrow[l, shift left] \arrow[r, shift left, "\text{introduction of continuations}"] & \text{Core CPS} \arrow[l, shift left, "\text{elimination of continuations}"] \arrow[r, shift right] & \text{CPS} \arrow[l, shift right, "\text{naming continuations}"'] \\
{} \arrow[r, phantom, "\text{Galois connection}"] & {} \arrow[r, phantom, "\text{bijection}"] & {} \arrow[r, phantom, "\text{Galois connection}"] & {}
\end{tikzcd}$$

This staging scales up to the call-by-name CPS transformation [29] because this transformation can be staged into (1) thunk introduction and (2) call-by-value CPS transformation [10]. Thus the call-by-value CPS transformation is a more basic CPS transformation, in some sense.

9 Issues and Future Work

The CPS transformation must offer useful properties because it is used in many places: for flow analysis [32], for parallelization [19], for compiling [1, 33], and for partial evaluation [2, 6], to name a few. Yet it is not easy to pinpoint which properties CPS offers that are not already there in DS, besides the obvious: CPS terms are a subset of DS terms and thus they can be modeled and processed in a simpler way. A staged CPS-transformation such as the one presented here enables one to examine each stage to determine which one triggers which property.

To this end, we need to refine the partial order over CoreDS and CoreCPS terms because the introduction and elimination of continuations is not order-preserving, currently (*cf.* Section 5.5). Better partial-orders would make it possible to consider the transformation over a least term.

```
(defineType Tree
  (Leaf value)
  (Node left right))
```

Figure 8: Binary tree datatype declaration

```
(define overweight?
  (lambda (x t)
    (call/cc
      (lambda (c)
        (letrec ([traverse
                  (lambda (t)
                    (let ([w (caseType t
                               [(Leaf value) value]
                               [(Node left right)
                                (+ (traverse left) (traverse right))])])
                      (if (> w x)
                          (throw c #t)
                          w)))])
          (let ([_ (traverse t)])
            #f))))))
```

Figure 9: DS program

```
(define overweight?
  (lambda (x t)
    (call/cc
      (lambda (c)
        (letrec ([traverse
                  (lambda (t)
                    (let ([w (caseType t
                               [(Leaf value) value]
                               [(Node left right)
                                (bind ([l (traverse left)]
                                       [r (traverse right)])
                                  (+ l r))])])
                      (if (> w x)
                          (throw c #t)
                          w)))])
          (bind ([v (traverse t)])
            (let ([_ v])
              #f)))))))
```

Figure 10: Staged DS program (all intermediate values are named)

```
(define overweight?
  (lambda (x t c)
    (letrec ([traverse
             (lambda (t k)
               (let ([k (lambda (v)
                          (let ([w v])
                            (if (> w x)
                                (c #t)
                                (k w))))])
                 (caseType t
                   [(Leaf value) (k value)]
                   [(Node left right)
                    (schedule (list (lambda (k) (traverse left k))
                                    (lambda (k) (traverse right k)))
                      (lambda (l r) (k (+ l r))))]))))])
      (schedule (list (lambda (k) (traverse t k)))
        (lambda (v)
          (let ([_ v])
            (c #f)))))))
```

Figure 11: Staged CPS program (all intermediate continuations are abstracted)

```
(define overweight?
  (lambda (x t c)
    (letrec ([traverse
              (lambda (t k)
                (let ([k (lambda (v)
                           (let ([w v])
                             (if (> w x)
                                 (c #t)
                                 (k w))))])
                  (caseType t
                    [(Leaf value) (k value)]
                    [(Node left right)
                     (traverse left (lambda (l)
                                      (traverse right (lambda (r)
                                                        (k (+ l r))))))]))))])
      (traverse t (lambda (v)
                    (let ([_ v])
                      (c #f)))))))
```

Figure 12: CPS program with left-to-right sequencing order

$$\begin{aligned}\mathcal{E}\,[\![\texttt{(E}_0\ \texttt{E}_1\ \ldots\ \texttt{E}_n\texttt{)}]\!]\,\rho\,\kappa \;&=\; \mathit{aplis}\,(\mathit{permute}\,\langle\mathcal{E}\,[\![\texttt{E}_0]\!]\,\rho,\ \ldots,\ \mathcal{E}\,[\![\texttt{E}_n]\!]\,\rho\rangle)\\ &\qquad\qquad ((\lambda\,\langle\epsilon_0,\ \epsilon_1,\ \ldots,\ \epsilon_n\rangle\,.\,\mathit{applicate}\,\epsilon_0\,\langle\epsilon_1,\ \ldots,\ \epsilon_n\rangle\,\kappa)\circ\mathit{unpermute})\end{aligned}$$

where

$$\begin{aligned}\texttt{K} \;&=\; \texttt{E}^* \to \texttt{C}\\ \mathit{permute} \;&:\; (\texttt{K} \to \texttt{C})^* \to (\texttt{K} \to \texttt{C})^*\\ \mathit{unpermute} \;&:\; \texttt{E}^* \to \texttt{E}^*\\ \mathit{applicate} \;&:\; \texttt{E} \to \texttt{E}^* \to \texttt{K} \to \texttt{C}\\ \mathit{single} \;&:\; (\texttt{E} \to \texttt{C}) \to \texttt{K}\\ \mathit{aplis} \;&:\; (\texttt{K} \to \texttt{C})^* \to \texttt{K} \to \texttt{C}\\ \mathit{aplis} \;&=\; \lambda\,\langle f_0,\ \ldots,\ f_n\rangle\,.\,\lambda\,\kappa\,.\,f_0\,(\mathit{single}(\lambda\,\epsilon_0\,.\,\ldots f_n\,(\mathit{single}(\lambda\,\epsilon_n\,.\,\kappa\,\langle\epsilon_0,\ \ldots,\ \epsilon_n\rangle)))\ldots))\end{aligned}$$

and *aplis* is the only new auxiliary function in the formal semantics of Scheme.

Figure 13: Denotational semantics of applications, in Scheme (fragment)

$$\begin{array}{rcl} \mathcal{E}[\![(\mathtt{E}_0\ \mathtt{E}_1\ \ldots\ \mathtt{E}_n)]\!]\,\rho\,\kappa & = & \mathit{schedule}\,\langle \mathcal{E}[\![\mathtt{E}_0]\!]\,\rho, \ldots, \mathcal{E}[\![\mathtt{E}_n]\!]\,\rho\rangle \\ & & \qquad (\lambda\,\langle\epsilon_0, \epsilon_1, \ldots, \epsilon_n\rangle\,.\,\mathit{applicate}\,\epsilon_0\,\langle\epsilon_1, \ldots, \epsilon_n\rangle\,\kappa) \end{array}$$

where

$$\begin{array}{rcl} \mathtt{K} & = & \mathtt{E}^* \rightarrow \mathtt{C} \\ \mathit{schedule} & : & (\mathtt{K} \rightarrow \mathtt{C})^* \rightarrow \mathtt{K} \rightarrow \mathtt{C} \\ \mathit{applicate} & : & \mathtt{E} \rightarrow \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C} \end{array}$$

and *schedule* is the only new auxiliary function in the formal semantics of Scheme.

Figure 14: Denotational semantics of applications, in Scheme (revised fragment)

Then we could determine to which extent it would be sufficient to simplify a term within the same style (*i.e.*, DS, CoreDS, CoreCPS, or CPS) rather than transforming it into another style.

10 Applications

10.1 The Scheme programming language

A few years ago [20], Jones and Muchnick proposed a programming language design based on *binding times* — including lexical-analysis time, syntax-analysis time, macro-expansion time, semantics-analysis time, code-generation time, link-time, and run-time. They also proposed that the compiler for such an ideal programming language should be structured in phases corresponding to each of the binding times, and that no reference should be made to an earlier binding time. Our staged CPS- and DS-transformations follow this spirit by separating evaluation order from sequencing order. They can also be applied to the Scheme programming language as follows.

According to the semantics of Scheme [4], sub-expressions in an application are evaluated in an unspecified order (a controversial issue among Scheme programmers). Our CPS transformation remains uncommitted with respect to the sequencing order and thus it suggests we rephrase a part of the Scheme semantics as follows.

The formal semantics of Scheme [4, Sections 7.2.3 & 7.2.4] uses two inverse functions *permute* and *unpermute* to ensure the sequencing-order independence of sub-expressions in an application. These functions map a piece of abstract syntax into a new piece of abstract syntax, hiding the fact that this semantics is actually compositional [35]. From the point of view of binding-times, this specification freezes the sequencing order at syntax-analysis time. Our uncommitted CPS transformation enables one to delay this permutation until a later binding-time, by permuting the *computations* of these sub-expressions. Figure 13 presents the new fragment specifying the semantics of applications, and Figure 14 re-expresses it in terms of schedule.

In addition, unscheduled CPS introduces a new style of CPS programming, without oversequentialization, *e.g.*, to simulate nondeterminism.

10.2 Compile-time analyses

Let us now assess the new possibilities offered by this staged transformation, and whether we can isolate the usefulness of the CPS transformation into one of the stages. CoreCPS seems quite promising as an intermediate language for compile-time analyses. It enjoys many of the useful properties of both DS and CPS.

Instantiating a CoreCPS term with any scheduling strategy yields a CPS term. Therefore, CoreCPS enjoys the same traditional advantages as CPS and leads to naturally forward program analyses[4] [6] and simpler program analyzers [32]. Accordingly, compile-time analyses still yield more precise results, both intra-procedurally and inter-procedurally [2, 6], given CoreCPS-transformed terms [28].

Compile-time analyses benefit from CPS to differing degrees. Binding-time analysis produces better information on CPS terms because information moves across the boundaries of conditionals and procedure calls. Here sequencing order is irrelevant, so ordinary CPS is adequate. On the other hand, while the fact that sequencing order is explicit in CPS terms improves single-threading detection, Fradet observes that whether a variable is classified to be single-threaded may depend on the choice of the sequencing order [15]. By maintaining the boundaries between independent subexpressions, CoreCPS allows the single-threading analysis to determine the best sequencing order for each expression locally, when it makes a difference. Finally, some analyses do not benefit from CPS transforming the source program. For example, analyses for register allocation require reordering independent subexpressions to produce useful results. This reordering is notoriously complex on CPS terms, but in CoreCPS the independent subexpressions are easily accessible. The analysis can proceed almost unchanged, taking advantage of the bijection between CoreCPS and CoreDS terms.

Acknowledgments

We are grateful to Dan Friedman for his support and to the other members of the programming language groups of Indiana University and Kansas State University for their feedback. Special thanks to Andrzej Filinski, Jürgen Koslowski, Karoline Malmkjær, and David Schmidt for commenting an earlier draft, and to the referees for their guidance.

---

[4] Since CPS terms are tail-recursive, there is nothing to propagate backwards but the final result.

The diagrams illustrating the separation of stages were drawn with Kristoffer Rose's XY-pic package.

References

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

[2] Anders Bondorf. Improving binding times without explicit CPS-conversion. In LFP'92 [21], pages 1–10.

[3] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

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

[5] Charles Consel. *Report on Schism'92.* Oregon Graduate Institute, Beaverton, Oregon, October 1992. Research Report.

[6] Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes [18], pages 496–519.

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

[8] 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.

[9] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4), 1992. To appear.

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

[11] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In LFP'92 [21], pages 299–310.

[12] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[13] 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.

[14] 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.

[15] Pascal Fradet. Syntactic detection of single-threading using continuations. In Hughes [18], pages 241–258.

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

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

[18] John Hughes, editor. *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, Cambridge, Massachusetts, August 1991.

[19] William L. Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *LISP and Symbolic Computation*, 2(3/4):179–396, October 1989.

[20] Neil D. Jones and Steven S. Muchnick. Some thoughts towards the design of an ideal language. In *ACM Conference on Principles of Programming Languages*, pages 77–94, 1976.

[21] *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, California, June 1992.

[22] Karoline Malmkjær. On static properties of specialized programs. In Michel Billaud *et al.*, editor, *Analyse Statique en Programmation Équationnelle, Fonctionnelle et Logique*, volume 74 of *Bigre Journal*, pages 234–241, Bordeaux, France, October 1991. IRISA, Rennes, France.

[23] Karoline Malmkjær. Predicting properties of residual programs. In Charles Consel, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Research Report 909, Department of Computer Science, Yale University, pages 8–13, San Francisco, California, June 1992.

[24] 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.

[25] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[26] 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.

[27] Chetan R. Murthy. An evaluation semantics for classical proofs. In *Proceedings of the Sixth Symposium on Logic in Computer Science*, pages 96–107, Amsterdam, The Netherlands, July 1991. IEEE.

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

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

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

[31] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In LFP'92 [21], pages 288–298.

[32] 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.

[33] 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.

[34] Allen Stoughton. Substitution revisited. *Theoretical Computer Science*, 59:317–325, 1988.

[35] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[36] 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.

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

[38] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In Steve Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 294–311, Pittsburgh, Pennsylvania, March 1991. 7th International Conference.