Representing Control

A Study of the CPS transformation *

Olivier Danvy
Kansas State University †
(danvy@cis.ksu.edu)

Andrzej Filinski
Carnegie Mellon University ‡
(Andrzej.Filinski@cs.cmu.edu)

February 1991; Revised June 1992

Abstract

This paper investigates the transformation of λ_v-terms into continuation-passing style (CPS). We show that by appropriate η-expansion of Fischer and Plotkin's two-pass equational specification of the CPS transform, we can obtain a *static* and *context-free* separation of the result terms into "essential" and "administrative" constructs. Interpreting the former as syntax builders and the latter as directly executable code, we obtain a simple and efficient one-pass transformation algorithm, easily extended to conditional expressions, recursive definitions, and similar constructs. This new transformation algorithm leads to a simpler proof of Plotkin's simulation and indifference results.

Further we show how CPS-based control operators similar to but more general than Scheme's **call/cc** can be naturally accommodated by the new transformation algorithm. To demonstrate the expressive power of these operators, we use them to present an equivalent but even more concise formulation of the efficient CPS transformation algorithm.

Finally, we relate the fundamental ideas underlying this derivation to similar concepts from other work on program manipulation; we derive a one-pass CPS transformation of λ_n-terms; and we outline some promising areas for future research.

*To appear in the journal *Mathematical Structures in Computer Science*. Technical Report CIS-91-2 (revised version), Department of Computing and Information Sciences, Kansas State University.

†Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas 66506, USA.
This work was partly supported by NSF under grant CCR-9102625.

‡School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, USA.

Contents

List of Figures

1 Introduction and Motivation

The usual presentation of the continuation-passing style (CPS) transformation [35] is concise and simple, but tends to yield unreasonably large residual terms containing a lot of "administrative redexes." While these redexes turn out to be relatively harmless from a theoretical perspective, they do require a special twist (the so-called "colon-translation") for proving certain important properties of the transformation [35, 39, 31].

In practice, eliminating the administrative redexes is absolutely essential to obtain transformed terms of a manageable size. However, such a "post-reduction" pass is often integrated with other, independent simplifications and optimization steps, and leads to relatively complex CPS transformers [43].

In the following, we will consider a systematization of the two-pass CPS transformation by focusing the attention on redexes that are introduced by the transformation itself, and by explicitly not reducing what would correspond to redexes in the source λ-term. Exploiting this distinction, we show that it is possible to perform all the administrative reductions "on the fly" in a single pass, without ever constructing the unreduced terms explicitly.

The CPS transformation permits a simple definition of generalized escape constructs like Scheme's **call/cc**. Such operators are often perceived to eliminate the need for explicit CPS programs. However, sometimes the greater generality of "genuine" CPS is actually needed to express an algorithm (*e.g.*, to implement backtracking [28].) Our investigation of the CPS transform leads naturally to the introduction of two new control operators, **shift** and **reset**, which allow the additional power of general CPS to be exploited in direct style programs. As an example, we will show how these operators permit us to express the efficient CPS transformation algorithm derived in this paper even more concisely.

Prerequisites

In the following, we will assume a basic familiarity with CPS and the λ_v-calculus, *i.e.*, the applicative order λ-calculus that forms the core of languages such as Scheme [5] and Standard ML [29].

For convenience in referring to individual applications, we will generally express them with an explicit operator @, writing $@M\,N$ instead of the traditional simple juxtaposition $M\,N$. This is a purely syntactical variation: no change or refinement of semantics is implied by the @-notation.

As in Standard ML, but in contrast to Scheme, we will also assume a strict left-to-right evaluation order, *i.e.*, that in an application $@M\,N$, M is evaluated before N. Where this distinction matters (*i.e.*, when both M and N are potentially nonterminating or "escaping" terms), the evaluation order of Scheme programs will be considered fixed through a **let** or a similar construct. Note, however, that we adopt this convention only to simplify the presentation, not to advocate a general style of programming which depends implicitly on argument evaluation order.

Occasionally, we will use Reynolds's notion of "serious" and "trivial" λ_v-terms [38]. Evaluating a serious term might loop so this term must be transformed into CPS, whereas evaluating a trivial one cannot loop so this term does not need to be transformed.

$$
\begin{aligned}
[\![x]\!] &= \lambda\kappa.@\kappa\,x \\
[\![\lambda x.M]\!] &= \lambda\kappa.@\kappa\,(\lambda x.[\![M]\!]) \\
[\![@M\,N]\!] &= \lambda\kappa.@[\![M]\!]\,(\lambda m.@[\![N]\!]\,(\lambda n.@(@m\,n)\,\kappa))
\end{aligned}
$$

Figure 1: Fischer & Plotkin's CPS transformation of λ_v-terms

Finally, let us recall the main property of a CPS λ-term: the independence of its reduction order. Reducing a CPS term with the call-by-name (CBN) or with the call-by-value (CBV) strategies yields the same evaluation steps [38, 35].

Overview

The rest of this paper is organized as follows. Section 2 describes the stepwise derivation of a one-pass CPS transformer from Plotkin's two-pass equational specification. Theorem 1 states that the one-pass transformer computes a result $\beta\eta$-equivalent to the original Fischer/Plotkin transformation. Section 3 investigates the reduction properties of the one-pass transformer. Theorem 2 captures Plotkin's "Indifference" and "Simulation" theorems for the original CPS translation. Section 4 extends the one-pass transformer to handle conditional expressions, recursive definitions, *etc.* Section 5 introduces control operators and their CPS transformation. Using these control operators, it presents a one-pass CPS transformer in direct style. Section 6 reviews related work on continuations and partial evaluation, and Section 7 concludes. Section A reproduces the development of Section 2 on Plotkin's CPS transformer for λ_n-terms.

2 Classical CPS transformation

Let us consider Fischer and Plotkin's equational specification for transforming a λ_v-term into CPS [20, 35], as displayed in Figure 1. Source terms are represented between double brackets and κ is a fresh variable.
Taken literally, this translation yields many artificial "administrative" redexes that must be post-reduced in a second pass; only then do we obtain a result in what is commonly recognized as "continuation-passing style" [43]. For example, translating

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

results in

$$
\begin{aligned}
\lambda k.@k\,(\lambda f.\lambda k.@k\,(\lambda x.\lambda k.@k\,(\lambda y.\lambda k.\;@\,(\lambda k.@(\lambda k.@k\,f)\,(\lambda m.@(\lambda k.@k\,y)\,(\lambda n.@(@m\,n)\,k)))\\
(\lambda m.@(\lambda k.@k\,x)\,(\lambda n.@(@m\,n)\,k)))))
\end{aligned}
$$

whose post-reduction yields

$$\lambda k.@k\,(\lambda f.\lambda k.@k\,(\lambda x.\lambda k.@k\,(\lambda y.\lambda k.@(@f\,y)\,(\lambda m.@(@m\,x)\,k))))$$

Conversely, an overly enthusiastic post-reducer is likely to perform too many reductions, *i.e.*, ones that would correspond to actual reductions in the source term. While this may be useful in its own right, it should not automatically be considered a part of the CPS transformation proper. In particular, excessive post-reduction can lead to uncontrolled "code duplication" in the result or, in the untyped case, even nontermination of the simplification.

In the following, we will therefore concentrate on integrating the first and the second passes subject to the two constraints: (1) a one-pass translation should not introduce any administrative redex; and (2) a one-pass translation should not perform any reduction that would correspond to reducing a source term.

Our derivation is simple. We analyze the original equational specification, identifying where redexes get built, independently of any actual source λ-term. When these redexes are "context-independent" we reduce them at translation time. When they are "context-dependent" we alter the translation with meaning-preserving transformations to make the construction of redexes context-independent. The goal of the game is to stage the CPS transformation into a "translation-time" part and a "run-time" part.

For precision and conciseness in the text, let us label the six lambdas and the six applications of this specification:

$$\begin{array}{rcl} [\![x]\!] & = & \lambda^1\kappa.@^1\kappa\, x \\ [\![\lambda x.M]\!] & = & \lambda^2\kappa.@^2\kappa\,(\lambda^3 x.[\![M]\!]) \\ [\![@M\,N]\!] & = & \lambda^4\kappa.@^3[\![M]\!]\,(\lambda^5 m.@^4[\![N]\!]\,(\lambda^6 n.@^5(@^6 m\, n)\,\kappa)) \end{array}$$

Our development is structured in three steps.

2.1 First step

As can be observed, the result of each elementary transformation (of a variable; of an abstraction; of an application) is an abstraction.

Question 1 *Where can the abstractions λ^1, λ^2, and λ^4 occur in the residual CPS term before post-reduction?*

Answer – by cases: (a) as the body of λ^3; (b) as the first argument of $@^3$; (c) as the first argument of $@^4$. In cases (b) and (c) the translation is building a redex that can be simplified by β-reduction. In case (a) no simplification can take place immediately.

As a consequence, whether the abstractions λ^1, λ^2, and λ^4 are post-reducible is context-dependent.

Question 2 *Can we get rid of this dependence?*

Answer – yes, by introducing one η-redex in the definitional translation of abstractions. This new redex will exhibit the continuation of the body:

$$[\![\lambda x.M]\!] \;=\; \lambda^2\kappa.@^2\kappa\,(\lambda^3 x.\lambda^7 k.@^7[\![M]\!]\,k)$$

The new redex is safe (in the sense of preserving operational behavior under both CBN and CBV) because $[\![M]\!]$ is itself a λ-abstraction. (Expanding "$@f\,a$" to "$\lambda x.@(@f\,a)\,x$" is not in general meaning-preserving, even if "$@\,f\,a$" has a functional type. Expanding "f" to "$\lambda x.@\,f\,x$" or "$\lambda y.E$" to "$\lambda x.@(\lambda y.E)\,x$" is safe.)

> **Remark:** Such an η-expansion may be felt as a step backwards in optimizing the translation, since η-reduction is usually perceived as an actual optimization step. In fact, and as illustrated by this development, the premature optimization in the translation of λ-abstractions contributes to muddying the water in the translated terms.

Now let us repeat Question 1:

Question 3 *Where can the abstractions resulting from each elementary transformation occur in the residual term?*

Answer – by cases: (a) as the first argument of $@^7$; (b) as the first argument of $@^3$; (c) as the first argument of $@^4$.

Therefore, the translation is building a β-redex. This redex can be simplified unconditionally.

2.2 Second step

Since the three λ-abstractions λ^1, λ^2, and λ^4 will be reduced at translation time, let us enumerate their possible arguments.

Question 4 *Which syntactic constructs can be denoted by κ in λ^1, λ^2, and λ^4?*

Answer – by cases: (a) the second argument of $@^7$ is an identifier k; (b) the second argument of $@^3$ is λ^5; (c) the second argument of $@^4$ is λ^6.

Again, the situation is irregular: if the argument of these applications (*i.e.*, the value denoted by κ) later gets applied, this application will be reducible in cases (b) and (c) only, *i.e.*, in a context-dependent fashion.

Question 5 *Can we get rid of this dependence?*

Answer – yes, by introducing another η-redex in the translation of abstractions. This redex will exhibit the application of the continuation.

$$[\![\lambda x.M]\!] \quad = \quad \lambda^2\kappa.@^2\kappa(\lambda^3x.\lambda^7k.@^7[\![M]\!](\lambda^8m.@^8k\,m))$$

Now let us repeat Question 4:

Question 6 *Which syntactic constructs can be denoted by κ?*

Answer – by cases (a) the second argument of $@^7$ is λ^8; (b) the second argument of $@^3$ is λ^5; (c) the second argument of $@^4$ is λ^6.

Now the different occurrences of κ are ensured to denote λ-abstractions only.

2.3 Third step

Question 7 *Where do these κ occur?*

Answer – by cases: (a) as the first argument of $@^1$; (b) as the first argument of $@^2$; (c) as the second argument of $@^5$. In cases (a) and (b), the translation is building a redex that can be simplified by β-reduction. In case (c) no simplification can take place immediately.

As a consequence, whether the application of a κ is post-reducible is context-dependent since in case (c) κ does not occur in function position in an application.

Question 8 *Can we get rid of this dependence?*

Answer – yes, by introducing a last η-redex in the definitional translation of applications. The redex will exhibit sending the result of an application to the continuation.

$$[\![@M\,N]\!] \quad = \quad \lambda^4\kappa.@^3[\![M]\!](\lambda^5 m.@^4[\![N]\!](\lambda^6 n.@^5(@^6 m\,n)(\lambda^9 a.@^9\kappa\,a)))$$

Now let us repeat Question 7:

Question 9 *Where do the κ occur?*

Answer – by cases: (a) as the first argument of $@^1$; (b) as the first argument of $@^2$; (c) as the first argument of $@^9$.

As a consequence, because by construction the translation is building a λ-abstraction which is ensured to occur in function position in an application, we can classify these λ-abstractions and applications to be simplifiable unconditionally.

To summarize, let us overline the λ-abstractions and the applications that will be reduced unconditionally as a part of the translation. Since they exist only at translation time we refer to them as "static."

$$\begin{array}{rcl}
[\![x]\!] & = & \overline{\lambda^1}\kappa.\overline{@^1}\kappa\,x \\
[\![\lambda x.M]\!] & = & \overline{\lambda^2}\kappa.\overline{@^2}\kappa(\lambda^3 x.\lambda^7 k.\overline{@^7}[\![M]\!](\overline{\lambda^8}m.@^8 k\,m)) \\
[\![@M\,N]\!] & = & \overline{\lambda^4}\kappa.\overline{@^3}[\![M]\!](\overline{\lambda^5}m.\overline{@^4}[\![N]\!](\overline{\lambda^6}n.@^5(@^6 m\,n)(\lambda^9 a.\overline{@^9}\kappa\,a)))
\end{array}$$

2.4 Completing the transformation: the other syntactic constructions

We can also list the possible arguments of the κ: they are the second arguments of $@^1$, $@^2$, and $@^9$, *i.e.*, (a) an identifier x; (b) λ^3; (c) an identifier a. These may be bound to m in λ^5, n in λ^6, and m in λ^8.

- In λ^5, m occurs as the first argument of $@^6$.
  - If m is bound to x or a no simplification is possible.
  - If m is bound to λ^3 then a β-reduction is possible but it would correspond to a reduction in the original term; therefore we do not want to perform it. Thus $@^6$ must be classified as irreducible and so is λ^3 and thus λ^7.

$$
\begin{array}{rcl}
\llbracket ... \rrbracket & : & [\text{syntax} \rightarrow \text{syntax}] \rightarrow \text{syntax} \\
\llbracket x \rrbracket & = & \overline{\lambda}\kappa.\overline{@}\kappa\, x \\
\llbracket \lambda x.M \rrbracket & = & \overline{\lambda}\kappa.\overline{@}\kappa\,(\underline{\lambda}x.\underline{\lambda}k.\overline{@}\llbracket M \rrbracket\,(\overline{\lambda}m.\underline{@}k\, m)) \\
\llbracket @M\, N \rrbracket & = & \overline{\lambda}\kappa.\overline{@}\llbracket M \rrbracket\,(\overline{\lambda}m.\overline{@}\llbracket N \rrbracket\,(\overline{\lambda}n.\underline{@}(\underline{@}m\, n)\,(\underline{\lambda}a.\overline{@}\kappa\, a)))
\end{array}
$$

Figure 2: One-pass CPS transformation of λ_v-terms

- In λ^6, n occurs as the second argument of $@^6$ which is irreducible.
- In λ^8, m occurs as the second argument of $@^8$ that cannot be reduced since the first argument is the identifier k.

As a consequence, the first argument of $@^5$ is irreducible and thus $@^5$ is irreducible. As another consequence, the second argument of $@^5$ must be irreducible.

To summarize, let us underline the abstractions and the applications that will be built unconditionally as a part of the translation. Since they are part of the transformed term, we refer to them as "dynamic."

$$
\begin{array}{rcl}
\llbracket x \rrbracket & = & \overline{\lambda^1}\kappa.\overline{@^1}\kappa\, x \\
\llbracket \lambda x.M \rrbracket & = & \overline{\lambda^2}\kappa.\overline{@^2}\kappa\,(\underline{\lambda}^3 x.\underline{\lambda}^7 k.\overline{@^7}\llbracket M \rrbracket\,(\overline{\lambda^8}m.\underline{@}^8 k\, m)) \\
\llbracket @M\, N \rrbracket & = & \overline{\lambda^4}\kappa.\overline{@^3}\llbracket M \rrbracket\,(\overline{\lambda^5}m.\overline{@^4}\llbracket N \rrbracket\,(\overline{\lambda^6}n.\underline{@}^5(\underline{@}^6 m\, n)\,(\underline{\lambda}^9 a.\overline{@^9}\kappa\, a)))
\end{array}
$$

As a simple inspection of this two-level specification shows, the only application that could possibly be a dynamic β-redex (*i.e.*, a redex in the resulting CPS term) is $@^6$. This happens if and only if m is bound to a dynamic λ, which again happens if and only if M is a λ-abstraction, so the source term contains a β-redex at this point. We do not want to reduce a dynamic β-redex because it would correspond to reducing a β-redex in the source term, which is not the job of the CPS transformation.

2.5 The complete transformation

To conclude, Figure 2 shows the final version of the transformer, without labels since they were only used for expository purposes. With its static/dynamic annotations, it can be read as a two-level specification *à la* Nielson and Nielson [34]. Operationally, the overlined λ's and @'s correspond to functional abstractions and applications in the translation program, while only the underlined occurrences represent abstract-syntax constructors.

Transforming a λ-term into CPS amounts to representing *contexts* (*i.e.*, λ-terms with a hole) as λ-abstractions. An empty context (*e.g.*, top-level) is represented with the identity function. An arbitrary ("dynamic") context is represented with some continuation k.

The result of transforming a term M into CPS in an empty context is given by

$$\overline{@}\llbracket M \rrbracket\,(\overline{\lambda}m.m)$$

whereas the result of transforming a term M into CPS in a dynamic context is given by

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

As can be noticed, both initial contexts are represented with translation-time λ-abstractions, as dictated by the answer to Question 6. In the rest of this section, we refer to such translation-time λ-abstractions as *static continuations.*

This instrumented new translation yields terms without extraneous redexes, in one pass. The static/dynamic distinction aimed at defining all the administrative β-redexes. These administrative β-redexes are bound at translation time and therefore they do not occur in residual terms.

The above development is summarized in the following theorem.

Theorem 1 *This equational specification,* i.e.*,* $\underline{\lambda}k.\overline{@}[\![M]\!](\overline{\lambda}m.\underline{@}k\,m)$*, computes a result* $\beta\eta$*-equivalent to the original Fischer/Plotkin transformation.*

Proof: We started from Fischer & Plotkin's specification and altered it in a meaning-preserving way, by introducing three η-redexes. We obtained a staged specification where static and dynamic constructs are not only distinct but context-independent. (Using Nielson and Nielson's terminology [34, 33], our two-level specification is "well-typed.") We can now reduce away all the static β-redexes.

Moreover, since the specification is compositional, a simple typing argument suffices to show that it is well-defined for all source terms, *i.e.*, that the static reductions do in fact terminate. Only the static lambdas and applications matter; for the purpose of termination, the dynamic lambdas/applications are just uninterpreted constructors. And the static part of the transformation is simply typed, with a single base type "syntax"; the κ's all have type "syntax $\rightarrow$ syntax" and the other static variables just have type "syntax". In particular, given an initial continuation κ, $\overline{@}[\![M]\!]\,\kappa$ is a strongly normalizable term of type "syntax". Thus, when interpreted as a functional program, the transformation always terminates (and in essentially linear time, since none of the static data is ever duplicated).

Finally, no redexes of the original term are reduced: the "λx" of an abstraction in the source term is always translated into a dynamic "λx", (*i.e.*, a syntax constructor) in the corresponding CPS term, and hence is never reduced away. □

Observation 1 *A* λ_v*-term and its CPS counterpart are related as follows:*

- *A variable is translated into itself.*
- *A* λ*-abstraction is translated into two* λ*-abstractions and one application.*
- *An application is translated into two applications and one* λ*-abstraction.*

This observation provides a simple correspondence between the size of a term and its CPS counterpart. Moreover, since for all of the new λ-abstractions introduced by the translation, the abstracted variable occurs exactly once in the body, this relationship extends directly to the number of reduction steps performed during evaluation of the two terms (more about this in Section 3).

$$
\begin{array}{rcl}
[\![...]\!] & : & [\text{syntax} \to \text{syntax}] \to \text{syntax} \\
[\![x]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\, x \\
[\![\lambda x.M]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\, (\underline{\lambda} x.\underline{\lambda} k.\overline{@}[\![M]\!]'\, k) \\
[\![@M\, N]\!] & = & \overline{\lambda}\kappa.\overline{@}[\![M]\!]\, (\overline{\lambda} m.\overline{@}[\![N]\!]\, (\overline{\lambda} n.\underline{@}(\underline{@} m\, n)\, (\underline{\lambda} a.\overline{@}\kappa\, a)))
\end{array}
$$

$$
\begin{array}{rcl}
[\![...]\!]' & : & \text{syntax} \to \text{syntax} \\
[\![x]\!]' & = & \overline{\lambda} k.\underline{@} k\, x \\
[\![\lambda x.M]\!]' & = & \overline{\lambda} k.\underline{@} k\, (\underline{\lambda} x.\underline{\lambda} k.\overline{@}[\![M]\!]'\, k) \\
[\![@M\, N]\!]' & = & \overline{\lambda} k.\overline{@}[\![M]\!]\, (\overline{\lambda} m.\overline{@}[\![N]\!]\, (\overline{\lambda} n.\underline{@}(\underline{@} m\, n)\, k))
\end{array}
$$

Figure 3: One-pass, "properly tail-recursive" CPS transformation of λ_v-terms

2.6 Tail-calls and η-redexes

While introducing η-redexes is crucial to avoid building β-redexes in residual terms, in one simple case it yields extra η-redexes in the transformed program. Not surprisingly this arises for tail-calls, as illustrated here:

$$\overline{@}[\![\lambda f.@f\, x]\!]\, (\overline{\lambda} m.m) \;=\; \lambda f.\lambda k.@(@f\, x)\, (\lambda a.@k\, a)$$

instead of $\lambda f.\lambda k.@(@f\, x)\, k$

Four straightforward possibilities come to the mind. We can leave these η-redexes where they are (which might be actually useful if the term is subjected to further transformation or if Observation 1 is used since tail-call optimizations change the number of reduction steps.) We can detect when a newly constructed λ-abstraction is actually an η-redex and simplify it at this time. We can instrument the translation with an inherited attribute identifying tail-call contexts, eliminating the dynamic tests on term structure entirely. Or equivalently we can duplicate the rules to account for tail-call contexts, as in Figure 3, and in a way reminiscent of Clinger's double induction proof in his Scheme compiler [4].

> **Rationale:** The auxiliary translation $[\![...]\!]'$ is used when the static continuation would have the form $\overline{\lambda} m.\underline{@} k\, m$; this avoids building an η-redex in the transformation of applications (hence the term "properly tail-recursive" [43]).

The result of transforming a term M into CPS in an empty context is still given by

$$\overline{@}[\![M]\!]\, (\overline{\lambda} m.m)$$

whereas the result of transforming a term M into CPS in a dynamic context is now simply

$$\underline{\lambda} k.\overline{@}[\![M]\!]'\, k$$

By construction, this instrumented new translation yields terms without extraneous η-redexes, and of course with no β-redexes, in one completely syntax-directed pass. And since the only difference with the transformation in Figure 2 is the elimination of trivial η-redexes, the new translation preserves the statement of Theorem 1.

3 Reduction properties

We can now formulate results about the new CPS translation analogous to the ones for the original specification. Moreover, since we have eliminated all administrative redexes "once and for all", the proofs become considerably simpler than for the unoptimized translation.

It may be worth quickly going over why administrative redexes raise problems in the original translation [35, 39]. Ideally, the following implication would hold:

$$M \rightarrow_v N \;\Rightarrow\; @[\![M]\!](\lambda x.x) \rightarrow_v^* @[\![N]\!](\lambda x.x)$$

Unfortunately, the administrative redexes get in the way of such a result. What really happens is the following: first, by a sequence of administrative reductions, $@[\![M]\!](\lambda x.x) \rightarrow_v^* M'$. Then $M' \rightarrow_v N'$, corresponding to the original redex. However, there is no reason to expect than N' now *reduces* to $@[\![N]\!](\lambda x.x)$; in fact, it would have to *expand* back to recreate the administrative redexes of the latter.

One therefore has to prove instead that the implication above holds "modulo administrative reductions". This is formalized by the so-called "colon-translation" developed by Plotkin [35]. For any term M and value K, one defines *syntactically* a term $M : K$ and proves it equal to $@[\![M]\!]K$ with the first series of administrative reductions performed (*i.e.*, corresponding to the term M' above). One then proves that if $M \rightarrow_v N$ then $M : K \rightarrow_v^* N : K$. Finally, for every reduction sequence

$$M_0 \rightarrow_v M_1 \rightarrow_v \dots \rightarrow_v V$$

where V is a non-functional value, one can take $K = \lambda x.x$ and get

$$@[\![M_0]\!](\lambda x.x) \rightarrow_v^* M_0 : (\lambda x.x) \rightarrow_v^* M_1 : (\lambda x.x) \rightarrow_v^* \dots \rightarrow_v^* V : (\lambda x.x) = @(\lambda x.x)V \rightarrow_v V$$

It is crucial to note that the colon-translation only removes enough redexes to "expose" the first real reduction. In particular, it never removes redexes within λ-abstractions. Nor does it have to, since it only acts as a proof technique, not a "code optimization" pass. In the new translation, on the other hand, we get a *direct* correspondence between reduction steps in the original and the translated term.

> **Remark:** It does seem possible to modify the colon-translation to perform more administrative reductions at translation time [22]. Nevertheless, a practical translation (*i.e.*, for a full language like Scheme) based on such an approach could be awkward because of the combinatorial explosion arising from translation-time distinctions between values and non-values in source terms.

Let us first observe a few elementary properties of the two-level translation. To simplify the presentation, we will generally treat α-equivalent dynamic terms as equal.

Definition 1 *If V is a* value *(*i.e.*, either a variable or an abstraction), we define*

$$\Psi(V) = \overline{@}\llbracket V \rrbracket (\overline{\lambda} x.x)$$

(This differs from Plotkin's Ψ in that the latter does not eliminate administrative redexes inside transformed λ-abstractions). It is immediate that $\Psi(V)$ is itself a value.

Lemma 1 *If V is a value and P is not a value (*i.e.*, an application), the following simple properties hold (where K is any term, that is, a term of type* syntax *and κ a static continuation, that is, a term of type* syntax $\rightarrow$ syntax*):*

$$\begin{array}{rcl} \overline{@}\llbracket V \rrbracket \kappa & = & \overline{@}\kappa (\Psi(V)) \\ \overline{@}\llbracket V \rrbracket' K & = & \underline{@} K (\Psi(V)) \\ \overline{@}\llbracket P \rrbracket \kappa & = & \overline{@}\llbracket P \rrbracket' (\underline{\lambda} a. \overline{@}\kappa\, a) \end{array}$$

Proof: Immediate in all cases. □

Let us write substitution of N for x in M as $M[x \leftarrow N]$. Now, we need to show that the static parts of the translation do not interfere with substitution.

Definition 2 *We say that a variable x occurs free in a static continuation κ if for some M it occurs free in $\overline{@}\kappa M$ but not in M. κ is called* schematic *if for any terms M and N, and variable x not occurring free in κ,*

$$(\overline{@}\kappa M)[x \leftarrow N] \quad = \quad \overline{@}\kappa (M[x \leftarrow N])$$

(informally, this ensures that κ preserves the syntactic structure of its argument, and does not capture any free variables occurring in it).

One easily sees that any κ defined using only static abstraction, static application, and the syntax constructors (with any dynamic abstractions introducing only "new" variables) is schematic.

We can now formalize how substitution of arbitrary values for identifiers commutes with CPS-translation:

Lemma 2 *Let M be a term, V a value, x a variable, and let κ be a schematic continuation and K any term. Then*

$$\begin{array}{rcl} \overline{@}\llbracket M[x \leftarrow V] \rrbracket \kappa & = & (\overline{@}\llbracket M[x \leftarrow x'] \rrbracket \kappa)[x' \leftarrow \Psi(V)] \\ \overline{@}\llbracket M[x \leftarrow V] \rrbracket' K & = & (\overline{@}\llbracket M[x \leftarrow x'] \rrbracket' K)[x' \leftarrow \Psi(V)] \end{array}$$

where x' is a new variable (the renaming is necessary to take care of the case when x occurs free in κ or K).

Proof: By induction on M. All cases are straightforward (let $y \neq x$):

$$
\begin{aligned}
\overline{@}[\![x[x\leftarrow V]]\!]\,\kappa &= \overline{@}[\![V]\!]\,\kappa \;=\; \overline{@}\kappa\,(\Psi(V)) \;=\; \overline{@}\kappa\,(x'[x'\leftarrow\Psi(V)]) \\
&= (\overline{@}\kappa\,x')[x'\leftarrow\Psi(V)] \;=\; (\overline{@}[\![x[x\leftarrow x']]\!]\,\kappa)[x'\leftarrow\Psi(V)]
\end{aligned}
$$

$$
\overline{@}[\![y[x\leftarrow V]]\!]\,\kappa = \overline{@}[\![y]\!]\,\kappa \;=\; (\overline{@}[\![y[x\leftarrow x']]\!]\,\kappa)[x'\leftarrow\Psi(V)]
$$

$$
\overline{@}[\![(\lambda x.M)[x\leftarrow V]]\!]\,\kappa = (\text{same as } y[x\leftarrow V])
$$

$$
\begin{aligned}
\overline{@}[\![(\lambda y.M)[x\leftarrow V]]\!]\,\kappa &= \overline{@}[\![\lambda y.M[x\leftarrow V]]\!]\,\kappa \;=\; \overline{@}\kappa\,(\underline{\lambda} y.\underline{\lambda} k.\overline{@}[\![M[x\leftarrow V]]\!]'\,k) \\
&= \overline{@}\kappa\,(\underline{\lambda} y.\underline{\lambda} k.(\overline{@}[\![M[x\leftarrow x']]\!]'\,k)[x'\leftarrow\Psi(V)]) \\
&= (\overline{@}\kappa\,(\underline{\lambda} y.\underline{\lambda} k.\overline{@}[\![M[x\leftarrow x']]\!]'\,k))[x'\leftarrow\Psi(V)] \\
&= (\overline{@}[\![\lambda y.M[x\leftarrow x']]\!]\,\kappa)[x'\leftarrow\Psi(V)] \\
&= (\overline{@}[\![(\lambda y.M)[x\leftarrow x']]\!]\,\kappa)[x'\leftarrow\Psi(V)]
\end{aligned}
$$

$$
\begin{aligned}
\overline{@}[\![(@M\,N)[x\leftarrow V]]\!]\,\kappa &= \overline{@}[\![@(M[x\leftarrow V])\,(N[x\leftarrow V])]\!]\,\kappa \\
&= \overline{@}[\![M[x\leftarrow V]]\!]\,(\overline{\lambda} m.\overline{@}[\![N[x\leftarrow V]]\!]\,(\overline{\lambda} n.\underline{@}(\underline{@} m\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a))) \\
&= (\overline{@}[\![M[x\leftarrow x']]\!]\,(\overline{\lambda} m.(\overline{@}[\![N[x\leftarrow x']]\!]\,(\overline{\lambda} n. \\
&\qquad \underline{@}(\underline{@} m\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a)))[x'\leftarrow\Psi(V)]))[x'\leftarrow\Psi(V)] \\
&= (\overline{@}[\![M[x\leftarrow x']]\!]\,(\overline{\lambda} m.\overline{@}[\![N[x\leftarrow x']]\!]\,(\overline{\lambda} n. \\
&\qquad \underline{@}(\underline{@} m\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a))))[x'\leftarrow\Psi(V)] \\
&= (\overline{@}[\![@(M[x\leftarrow x'])\,(N[x\leftarrow x'])]\!]\,\kappa)[x'\leftarrow\Psi(V)] \\
&= (\overline{@}[\![(@M\,N)[x\leftarrow x']]\!]\,\kappa)[x'\leftarrow\Psi(V)]
\end{aligned}
$$

The cases for the second equation are analogous.

□

Let us now recall the formal definition of the reduction strategies [35]:

Definition 3 *One-step by-value reduction is defined as follows:*

$$
@(\lambda x.M)\,V \rightarrow_v M[x\leftarrow V] \qquad \frac{M \rightarrow_v M'}{@M\,N \rightarrow_v @M'\,N} \qquad \frac{N \rightarrow_v N'}{@V\,N \rightarrow_v @V\,N'}
$$

(where V is a value), and similarly one-step by-name reduction:

$$
@(\lambda x.M)\,N \rightarrow_n M[x\leftarrow N] \qquad \frac{M \rightarrow_n M'}{@M\,N \rightarrow_n @M'\,N} \qquad \frac{N \rightarrow_n N'}{@x\,N \rightarrow_n @x\,N'}
$$

We write $M \rightarrow_a N$ if both $M \rightarrow_v N$ and $M \rightarrow_n N$. For any of the three reduction relations $\rightarrow$, we use $\rightarrow^+$ and $\rightarrow^*$ to refer to its transitive and reflexive-transitive closure, respectively.

Lemma 3 *Let M and N be terms such that $M \rightarrow_v N$, and let κ be a schematic continuation. Then*

$$\overline{@}[\![M]\!]\,\kappa \rightarrow_a^+ \overline{@}[\![N]\!]\,\kappa$$

(in fact, in either two or three reductions).

Proof: By induction on the derivation of $\rightarrow_v$:

- Base case: $@(\lambda x.M)\,V \rightarrow_v M[x{\leftarrow}V]$.

$$\begin{array}{rcl}
\overline{@}[\![@(\lambda x.M)\,V]\!]\,\kappa & = & \overline{@}[\![\lambda x.M]\!]\,(\overline{\lambda} m.\overline{@}[\![V]\!]\,(\overline{\lambda} n.\underline{@}(\underline{@} m\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a))) \\
& = & \overline{@}[\![V]\!]\,(\overline{\lambda} n.\underline{@}(\underline{@}(\underline{\lambda} x.\underline{\lambda} k.\overline{@}[\![M]\!]'\,k)\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a)) \\
& = & \underline{@}(\underline{@}(\underline{\lambda} x.\underline{\lambda} k.\overline{@}[\![M]\!]'\,k)\,(\Psi(V)))\,(\underline{\lambda} a.\overline{@}\kappa\,a) \\
& \rightarrow_a & \underline{@}(\underline{\lambda} k.(\overline{@}[\![M]\!]'\,k)[x{\leftarrow}\Psi(V)])\,(\underline{\lambda} a.\overline{@}\kappa\,a) \\
& = & \underline{@}(\underline{\lambda} k.(\overline{@}[\![M[x{\leftarrow}x']]\!]'\,k)[x'{\leftarrow}\Psi(V)])\,(\underline{\lambda} a.\overline{@}\kappa\,a) \\
& = & \underline{@}(\underline{\lambda} k.\overline{@}[\![M[x{\leftarrow}V]]\!]'\,k)\,(\underline{\lambda} a.\overline{@}\kappa\,a) \\
& \rightarrow_a & \overline{@}[\![M[x{\leftarrow}V]]\!]'\,(\underline{\lambda} a.\overline{@}\kappa\,a)
\end{array}$$

  Now the following two cases arise:

  - $M[x{\leftarrow}V]$ is not a value. Then by Lemma 1, the last term is equal to $\overline{@}[\![M[x{\leftarrow}V]]\!]\,\kappa$, as required. The two reductions correspond to transferring the argument value and return continuation (if we use uncurried CPS translation, as in Section 4.3, only one reduction is needed).
  - $M[x{\leftarrow}V]$ is a value. Then again Lemma 1 gives

$$\begin{array}{rcl}
\overline{@}[\![M[x{\leftarrow}V]]\!]'\,(\underline{\lambda} a.\overline{@}\kappa\,a) & = & \underline{@}(\underline{\lambda} a.\overline{@}\kappa\,a)\,(\Psi(M[x{\leftarrow}V])) \\
& \rightarrow_a & \overline{@}\kappa\,(\Psi(M[x{\leftarrow}V])) \\
& = & \overline{@}[\![M[x{\leftarrow}V]]\!]\,\kappa
\end{array}$$

    The additional reduction in this case corresponds to an application of the return continuation to the value just computed.

- Inductive case 1: $@M\,N \rightarrow_v @M'\,N$ because $M \rightarrow_v M'$.

$$\begin{array}{rcl}
\overline{@}[\![@M\,N]\!]\,\kappa & = & \overline{@}[\![M]\!]\,(\overline{\lambda} m.\overline{@}[\![N]\!]\,(\overline{\lambda} n.\underline{@}(\underline{@} m\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a))) \\
& \rightarrow_a^+ & \overline{@}[\![M']\!]\,(\overline{\lambda} m.\overline{@}[\![N]\!]\,(\overline{\lambda} n.\underline{@}(\underline{@} m\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a))) \\
& = & \overline{@}[\![@M'\,N]\!]\,\kappa
\end{array}$$

- Inductive case 2: $@V\,N \rightarrow_v @V\,N'$ because $N \rightarrow_v N'$.

$$\begin{array}{rcl}
\overline{@}[\![@V\,N]\!]\,\kappa & = & \overline{@}[\![V]\!]\,(\overline{\lambda} m.\overline{@}[\![N]\!]\,(\overline{\lambda} n.\underline{@}(\underline{@} m\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a))) \\
& = & \overline{@}[\![N]\!]\,(\overline{\lambda} n.\underline{@}(\underline{@}(\Psi(V))\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a)) \\
& \rightarrow_a^+ & \overline{@}[\![N']\!]\,(\overline{\lambda} n.\underline{@}(\underline{@}(\Psi(V))\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a)) \\
& = & \overline{@}[\![V]\!]\,(\overline{\lambda} m.\overline{@}[\![N']\!]\,(\overline{\lambda} n.\underline{@}(\underline{@} m\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a))) \\
& = & \overline{@}[\![@V\,N']\!]\,\kappa
\end{array}$$

□

If we restrict evaluation to closed terms, any term is either already a value or contains a redex. However, the results extend easily to open terms, with free variables treated as uninterpreted constants (*i.e.*, with no associated δ-rules). In this case, there is a third possibility: evaluation may halt at a non-value term like $@x\,y$ from which no further progress is possible.

Following Plotkin, we define:

Definition 4 *A (necessarily open) term S is said to be* stuck *under a given strategy if it is neither a value nor reducible by any of the reduction rules for that strategy. A quick inspection of Definition 3 shows that such terms must be of the following form (where V is a value and N is any term):*

$$\begin{array}{rcll} S_v & ::= & @x\,V \mid @S_v\,N \mid @V\,S_v & \textit{(for call-by-value)} \\ S_n & ::= & @x\,V \mid @S_n\,N \mid @x\,S_n & \textit{(for call-by-name)} \end{array}$$

We note that every term stuck under CBN is also stuck under CBV.

Lemma 4 *Let M be any term and κ a static continuation. If M is stuck under CBV then $\overline{@}[\![M]\!]\,\kappa$ is stuck under any strategy.*

Proof: By structural induction on the stuck term M.
Base case, $M = @x\,V$:

$$\begin{array}{rcl} \overline{@}[\![@x\,V]\!]\,\kappa & = & \overline{@}[\![x]\!]\,(\overline{\lambda} m.\overline{@}[\![V]\!]\,(\overline{\lambda} n.\underline{@}(\underline{@}m\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a))) \\ & = & \underline{@}(\underline{@}x\,(\Psi(V)))\,(\underline{\lambda} a.\overline{@}\kappa\,a) \end{array}$$

where the inner application, and hence the entire term, is stuck.
Inductive case 1, $M = @S\,N$:

$$\overline{@}[\![@S\,N]\!]\,\kappa \;=\; \overline{@}[\![S]\!]\,(\overline{\lambda} m.\overline{@}[\![N]\!]\,(\overline{\lambda} n.\underline{@}(\underline{@}m\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a)))$$

which is stuck, by the induction hypothesis.
Inductive case 2, $M = @V\,S$:

$$\begin{array}{rcl} \overline{@}[\![@V\,S]\!]\,\kappa & = & \overline{@}[\![V]\!]\,(\overline{\lambda} m.\overline{@}[\![S]\!]\,(\overline{\lambda} n.\underline{@}(\underline{@}m\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a))) \\ & = & \overline{@}[\![S]\!]\,(\overline{\lambda} n.\underline{@}(\underline{@}(\Psi(V))\,n)\,(\underline{\lambda} a.\overline{@}\kappa\,a)) \end{array}$$

which again is stuck by induction hypothesis. □

> **Remark:** In a statically-typed setting, the above problem does not occur: the typing rules ensure that a non-functional constant will never be applied to an argument. However, the argument extends easily to languages with the possibility of runtime errors (*e.g.*, division by zero). A simple refinement of the proof shows that the CPS translation can even distinguish properly between different error conditions, *i.e.*, if evaluation of the original program causes a specific error, so does evaluation of the CPS-transformed program.

We can now state the main result, analogous to Plotkin's "Indifference" and "Simulation" theorems for the original CPS translation:

Theorem 2 *Let M be any λ-term (not necessarily closed) and V a value. If $M \rightarrow_v^* V$ then $\overline{@}[\![M]\!](\overline{\lambda}x.x) \rightarrow_a^* \overline{@}[\![V]\!](\overline{\lambda}x.x) = \Psi(V)$ (and in at most three times as many steps). In particular, if V is a free variable of the original term, M evaluates to $\Psi(V) = V$. Conversely, if M does not evaluate to a value under the call-by-value strategy, then for no strategy will $\overline{@}[\![M]\!](\overline{\lambda}x.x)$ evaluate to one.*

Proof: The first part follows immediately from Lemma 3 applied to every step of the reduction. Conversely, any infinite value-reduction sequence starting from M gives rise to an infinite, strategy-independent reduction sequence starting from $\overline{@}[\![M]\!](\overline{\lambda}x.x)$. Finally, if the original reduction sequence stops at a CBV-stuck term S, the corresponding CPS reduction sequence ends in the term $\overline{@}[\![S]\!](\overline{\lambda}x.x)$ which is stuck under any strategy (Lemma 4). □

4 Enriching the CPS transformation

This section investigates the translation of λ-terms as found in usual applicative-order functional languages and the problem of currying functions *vs.* tupling arguments.

4.1 CPS translation of extended and applied λ_v-terms

We now turn to translating λ_v-terms applied over constants and (pure) primitive operations and extended with conditional expressions, **let** and **letrec** expressions (restricting the values that are bound recursively to be functions). Such applied and extended λ-terms come straight from Scheme and Standard ML.

Following the line of Section 2, Figure 4 displays a one-pass equational specification of the CPS transformation. Primitive operations are treated differently from ordinary applications to simplify the equations. If a primitive operator q is to be passed as a functional value, it must be written with an explicit abstraction $\lambda x.q(x)$ — which can be done at syntax-analysis time.
Again, the result of transforming a term M into CPS in an empty context is given by

$$\overline{@}[\![M]\!](\overline{\lambda}m.m)$$

and the result of transforming M in a dynamic context is given by

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

This instrumented new translation yields terms without extraneous redexes (if tail-calls are handled as in Section 2.6), in one pass. We obtained these equations by analyzing the binding times of the valuation functions of a continuation semantics of the λ_v-calculus, along the line of Section 2. The simulation and indifference properties of CPS-transformed terms generalize to the above translation, but with one exception: unless the primitive operations are also transformed into continuation-passing versions, there is no longer a direct correspondence between δ-reductions in the original and the CPS-transformed term. Thus, the optimized translation for primitives should *only* be used for "completely pure" operations (see also Section 6.2).

$$
\begin{array}{rcl}
[\![...]\!] & : & [\text{syntax} \rightarrow \text{syntax}] \rightarrow \text{syntax} \\
[\![x]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\, x \\
[\![\lambda x.M]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\,(\underline{\lambda} x.\underline{\lambda} k.\overline{@}[\![M]\!]\,(\overline{\lambda} m.\underline{@} k\, m)) \\
[\![@M\, N]\!] & = & \overline{\lambda}\kappa.\overline{@}[\![M]\!]\,(\overline{\lambda} m.\overline{@}[\![N]\!]\,(\overline{\lambda} n.\underline{@}(\underline{@} m\, n)\,(\underline{\lambda} a.\overline{@}\kappa\, a))) \\
[\![P \rightarrow M,\ N]\!] & = & \overline{\lambda}\kappa.\overline{@}[\![P]\!]\,(\overline{\lambda} p.p \underline{\rightarrow} \overline{@}[\![M]\!]\,\kappa \underline{,}\, \overline{@}[\![N]\!]\,\kappa) \\
[\![q]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\, q \\
[\![q(M)]\!] & = & \overline{\lambda}\kappa.\overline{@}[\![M]\!]\,(\overline{\lambda} m.\overline{@}\kappa\,(\underline{q}(m))) \\
[\![q(M, N)]\!] & = & \overline{\lambda}\kappa.\overline{@}[\![M]\!]\,(\overline{\lambda} m.\overline{@}[\![N]\!]\,(\overline{\lambda} n.\overline{@}\kappa\,(\underline{q}(m, n)))) \\
[\![\mathbf{let}\ x = N\ \mathbf{in}\ M]\!] & = & \overline{\lambda}\kappa.\overline{@}[\![N]\!]\,(\overline{\lambda} n.\underline{\mathbf{let}}\ x' = n\ \underline{\mathbf{in}}\ \overline{@}[\![M[x \leftarrow x']]\!]\,\kappa) \\
[\![\mathbf{letrec}\ f = \lambda x.N\ \mathbf{in}\ M]\!] & = & \overline{\lambda}\kappa.\,\underline{\mathbf{letrec}}\ f' = \underline{\lambda} x.\underline{\lambda} k.\overline{@}[\![N[f \leftarrow f']]\!]\,(\overline{\lambda} n.\underline{@} k\, n)) \\
 & & \quad \underline{\mathbf{in}}\ \overline{@}[\![M[f \leftarrow f']]\!]\,\kappa
\end{array}
$$

Figure 4: One-pass CPS transformation of extended and applied λ_v-terms

As always with non-trivial symbolic reductions in λ-calculus, we face the problem of name clashes. These can occur when a context is moved inside the scope of a binding construct, as illustrated in the following example:

$$\overline{@}([\![\lambda x.x + (\mathbf{let}\ x = 3\ \mathbf{in}\ x)]\!])\,(\overline{\lambda} v.v) \quad = \quad \lambda x.\lambda k.\mathbf{let}\ x' = 3\ \mathbf{in}\ @k\,(x + x')$$

Had we not renamed the x introduced by the **let**, we would have inadvertently captured a free variable of the static continuation κ. However, variables declared in λ-abstractions do not need to be renamed because contexts are never moved inside the scope of a λ-abstraction.

Let us also note that if the subterm N in the translation of a **let**-expression is a value (notably, an abstraction), the corresponding variable in the transformed term will also be **let**-bound (as opposed to λ-bound) to $\Psi(N)$. Thus, the translation seems compatible with a restricted variant of ML-style polymorphism in which generalization can only be applied to values [24].

4.2 On duplicating contexts

The CPS translation above duplicates contexts for each conditional expression:

$$[\![P \rightarrow M,\ N]\!] \quad = \quad \overline{\lambda}\kappa.\overline{@}[\![P]\!]\,(\overline{\lambda} p.p \underline{\rightarrow} \overline{@}[\![M]\!]\,\kappa \underline{,}\, \overline{@}[\![N]\!]\,\kappa)$$

which increases the size of residual terms, as pointed out by Steele [43]. For example, translating the direct style expression

$$@f\,((x \rightarrow y,\ z) \rightarrow 4,\ 5)$$

yields the more voluminous

$$\begin{array}{l}
\lambda\kappa.x \rightarrow (y \rightarrow @(@f\,4)(\lambda v.@k\,v),\ @(@f\,5)(\lambda v.@k\,v)\,), \\
\qquad\qquad (z \rightarrow @(@f\,4)(\lambda v.@k\,v),\ @(@f\,5)(\lambda v.@k\,v)\,)
\end{array}$$

and thus a bigger term to compile and correspondingly more object code to produce. For this reason, compiler writers usually refrain from duplicating contexts by introducing a dynamic **let**-expression to share the static continuation between the two branches of the conditional:

$$\begin{aligned}[\![P \to M,\, N]\!] \;=\; & \overline{\lambda}\kappa.\underline{\mathbf{let}}\; k = \underline{\lambda} a.\overline{@}\kappa\, a \\ & \underline{\mathbf{in}}\; \overline{@}[\![P]\!]\,(\overline{\lambda} p.p \underline{\to} \overline{@}[\![M]\!]\,(\overline{\lambda} m.\underline{@}k\, m)\underline{,}\; \overline{@}[\![N]\!]\,(\overline{\lambda} n.\underline{@}k\, n))\end{aligned}$$

where we have inserted the appropriate η-redexes, in the line of Section 2. This modification restores the linearity property of the *static* part of the translation, *i.e.*, that bound variables of static λ-abstractions are used only once in their bodies, and thus maintains a linear relationship between the sizes of the original and transformed terms.

Because the **let**-expression introduces an explicit name for the context, each conditional branch de facto occurs in a dynamic context, just like the body of translated λ-abstractions. As in Section 2.6, we can use the special "tail-call" translation for these contexts:

$$[\![P \to M,\, N]\!] \;=\; \overline{\lambda}\kappa.\underline{\mathbf{let}}\; k = \underline{\lambda} a.\overline{@}\kappa\, a\; \underline{\mathbf{in}}\; \overline{@}[\![P]\!]\,(\overline{\lambda} p.p \underline{\to} \overline{@}[\![M]\!]'\, k\underline{,}\, \overline{@}[\![N]\!]'\, k)$$

For example, the expression above is translated into

$$\begin{aligned}\lambda k_0.\,\mathbf{let}\; k_1 = \lambda v_0.\,&\mathbf{let}\; k_2 = \lambda v_1.@(@f\, v_1)\, k_0 \\ &\quad\mathbf{in}\; v_0 \to @k_2\, 4,\; @k_2\, 5 \\ \mathbf{in}\; x \to @k_1\, y&,\; @k_1\, z\end{aligned}$$

Still there is room left for further simplification, *e.g.*, in the case of **let**-expressions. As revealed by a simple inspection of the equations, translating **let**-expressions whose headers are an application or a conditional expression will produce identity **let** expressions. These redundant **let**-expressions can be prevented at translation time:

$$[\![\mathbf{let}\; x = @N_0\, N_1\; \mathbf{in}\; M]\!] \;=\; \overline{\lambda}\kappa.\overline{@}[\![N_0]\!]\,(\overline{\lambda} n_0.\overline{@}[\![N_1]\!]\,(\overline{\lambda} n_1.\underline{@}(\underline{@} n_0\, n_1)\,(\underline{\lambda} x'.\overline{@}[\![M[x \leftarrow x']]\!]\,\kappa)))$$

4.3 Currying vs. tupling

For practical applications of the CPS transformation algorithm described above, we need the slight refinement considered in this section.

In a call-by-name language, there is a one-to-one correspondence between curried and non-curried forms of multi-argument functions, but for call-by-value this property is lost: there may be several, non-equivalent ways of "currying" a function. For example, the curried form of the two-argument function

$$g = \lambda(a, b).(@f\, a) + b$$

can be written as either of the two terms:

$$\tilde{g}_1 = \lambda a.\lambda b.(@f\, a) + b \qquad \text{or} \qquad \tilde{g}_2 = \lambda a.\mathbf{let}\; x = @f\, a\; \mathbf{in}\; \lambda b.x + b$$

Both of these qualify as curried forms of g, in the sense that for any pair of values a and b,

$$@(@\tilde{g}_1\, a)\, b \;=\; @(@\tilde{g}_2\, a)\, b \;=\; @g\,(a, b)$$

but $\tilde{g}_1$ and $\tilde{g}_2$ behave differently if applied to only a single argument a for which f fails to terminate.

Since the CPS transform clearly needs to accommodate curried functions like $\tilde{g}_2$ above, programs with "simple" curried functions appear unnecessarily complex. For example, curried addition

$$\lambda a.\lambda b.(a+b)$$

must be translated to the somewhat awkward

$$\lambda a.\lambda k.@k\,(\lambda b.\lambda k'.@k'\,(a+b))$$

instead of the more natural

$$\lambda a.\lambda b.\lambda k.@k\,(a+b)$$

The problem is that a higher-order type like $\tau_1 \to [\tau_2 \to \tau_3]$ does not make it clear whether the outermost function is "serious" or "trivial", as in these two last examples. This can be solved by extending the source language with either a product type or a new "trivial function space" (restoring the equivalence between "trivially curried" and uncurried forms of functions).

Moreover, the result of translating a function of type

$$\sigma \to \tau$$

into CPS with "answer" type o, can itself be expressed in either "curried CPS":

$$\sigma' \to [\tau' \to o] \to o$$

or "uncurried CPS:"

$$\sigma' \times [\tau' \to o] \to o$$

The functions resulting from the curried CPS transform can be easily checked to be always trivial (*i.e.*, immediately return a closure), but this is not clear from their type. Thus, if we want to treat the transformation result itself as a call-by-value term, the uncurried form of the translation is more precise.

4.4 Multi-argument functions

For languages like Standard ML, which have an explicit product type for expressing functions of several arguments, we can essentially use the equations of Figure 4 directly. Multiple variables in λ-abstractions can be treated as syntactic sugar for projections from a single argument, and tuple construction becomes a new primitive operator. For a Scheme-like language, however, the "argument tuple" is not an autonomous entity, but is closely tied in with functional abstraction and application. We thus need to adjust the CPS equations of these two constructs slightly if we want an uncurried source-to-source transformation:

$$\begin{array}{rcl}
[\![\lambda(x_1,\ldots,x_n).E]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\,(\underline{\lambda}(x_1,\ldots,x_n,k).\overline{@}[\![E]\!]\,(\overline{\lambda}x.\underline{@}k\,x)) \\
[\![@E_0\,(E_1,\ldots,E_n)]\!] & = & \overline{\lambda}\kappa.\overline{@}\,[\![E_0]\!] \\
 & & \qquad (\overline{\lambda}f.\overline{@}[\![E_1]\!]\,(\overline{\lambda}a_1.\ldots\overline{@}[\![E_n]\!]\,(\overline{\lambda}a_n.\underline{@}f\,(a_1,\ldots,a_n,\underline{\lambda}x.\overline{@}\kappa\,x))))
\end{array}$$

More precisely, an argument list is translated as follows:

$$\begin{array}{rcl}
[\![()]\!]^* & = & \overline{\lambda}\kappa.\overline{@}\kappa\,() \\
[\![E :: E^*]\!]^* & = & \overline{\lambda}\kappa.\overline{@}[\![E]\!]\,(\overline{\lambda}h.\overline{@}[\![E^*]\!]^*\,(\overline{\lambda}t.\overline{@}\kappa\,(h \mathbin{\overline{::}} t)))
\end{array}$$

and a full application as:

$$[\![@E_0\,(E_1, ..., E_n)]\!] \;=\; \overline{\lambda}\kappa.\overline{@}[\![E_0]\!]\,(\overline{\lambda}f.\overline{@}[\![(E_1, ..., E_n)]\!]^*\,(\overline{\lambda}l.\underline{@}f\,(l \,\overline{\diamond}\, [\underline{\lambda}x.\overline{@}\kappa\, x])))$$

where $\diamond$ represents list concatenation.

Remark: To make the Scheme-style connection between lists and argument tuples even closer, we could put the continuation argument *first* in the argument list rather than last. This would allow us to translate "variadic procedures" such as

`(lambda l (cdr l))` or `(lambda (a b . l) b)`

directly into

`(lambda (k . l) (k (cdr l)))` and `(lambda (k a b . l) (k b))`

respectively. However, readability appears to suffer when the continuation argument, which is often large, must be followed by others in an application. Making continuations occur first is used sometimes to compile functional programs by program transformation [21]. It is simple to write a one-pass CPS transformer where continuations precede values — just swap values and continuations in any of the specifications displayed in the figures.

5 Abstracting Control

So far we have been investigating *how* to perform the CPS transformation. This section explores properties and extensions of the transformation function.

5.1 Reynolds's escape operator

Not every λ-calculus term is obtainable as a result of the CPS transformation. Some of the "unused" terms correspond to control operators in the source language. For example, the operator **escape** (interdefinable with Scheme's **call/cc**) can be defined by the equation:

$$[\![\textbf{escape}\ c\ \textbf{in}\ M]\!] \;=\; \lambda\kappa.\textbf{let}\ c' = \lambda a.\lambda\kappa'.@\kappa\, a\ \textbf{in}\ @[\![M[c{\leftarrow}c']]\!]\,\kappa$$

As a control operator, **escape** captures the current continuation and provides a representation of this current continuation as if it were a function in the source program. Applying this function to a value amounts to abandoning the current context of computation and passing this value to the captured continuation. For example, transforming the term

$$\lambda f.\textbf{escape}\ c\ \textbf{in}\ 1 + (@f\, c)$$

into CPS in an empty context (and unfolding the **let**) yields

$$\lambda f.\lambda c.@(@f\,(\lambda x.\lambda k.@c\, x))(\lambda v.@c\,(1+v))$$

As can be observed in the equation above, **escape** duplicates the code for the current continuation. Along the lines of Section 4.2, this can be prevented by the following equation:

$$[\![\textbf{escape}\ c\ \textbf{in}\ M]\!] \;=\; \overline{\lambda}\kappa.\underline{\textbf{let}}\ k = \underline{\lambda}a.\overline{@}\kappa\, a\ \underline{\textbf{in}}\ \underline{\textbf{let}}\ c = \underline{\lambda}a.\underline{\lambda}k'.\underline{@}k\, a\ \underline{\textbf{in}}\ \overline{@}[\![M]\!]\,(\overline{\lambda}m.\underline{@}k\, m)$$

In the rest of this section, we shall waive concerns about such duplications of contexts.

5.2 Shift and reset

Let us note that even with escaping constructs, the result of the translation is in "ordinary" CPS form, *i.e.*, with no nested function applications. This suggests that there is still a considerable amount of untapped expressive power in the CPS formalism, reflecting control structures whose translations are more general λ-terms. In particular, we can define the two operators **shift** and **reset**, conceptually serving as composition and identity for continuation functions:

$$\begin{aligned} [\![\textbf{shift}\ c\ \textbf{in}\ M]\!] &= \lambda\kappa.\textbf{let}\ c = \lambda a.\lambda\kappa'.@\kappa'(@\kappa\, a)\ \textbf{in}\ @[\![M]\!](\lambda m.m) \\ [\![\langle M\rangle]\!] &= \lambda\kappa.@\kappa(@[\![M]\!](\lambda m.m)) \end{aligned}$$

Shift abstracts the current context as an ordinary, composable procedure (in contrast to the exceptional, non-composable procedures yielded by **escape**) and **reset** delimits the scope of such a context. **Shift** also differs from **escape** by not implicitly duplicating the current continuation (Felleisen's $\mathcal{C}$-operator introduced the behavior of not duplicating continuations [17]).

For example,

$$1 + \langle 10 + \textbf{shift}\ c\ \textbf{in}\ @c(@c\,100)\rangle \quad\Rightarrow\quad 1 + (10 + (10 + 100)) \quad\Rightarrow\quad 121$$

With the fuller control over contexts afforded by these two operators, we can express in a functional style many control structures that would otherwise have required us to either rewrite the program extensively or introduce side effects. For example, let us consider a functional representation of "applicative" nondeterministic (in the sense of backtracking) programming, as embodied, *e.g.*, in the programming language Icon [23].

Let us define a basic "nondeterministic choice" procedure:

$$\mathit{flip} \ = \ \lambda().\textbf{shift}\ c\ \textbf{in}\ @c\ \mathit{tt} \vee @c\ \mathit{ff}$$

When invoked, *flip* will "return" twice: once with each possible truth value. Here, we have specified that the final answer of the nondeterministic program should be true if *either* of these two return values causes the context to evaluate to true. For a canonical example, let φ be a boolean expression with free variables $b_1, ..., b_n$. To determine whether φ is satisfiable (*i.e.*, whether there exists an assignment of truth values to the variables making the whole expression true), we can now simply evaluate the natural direct-style program

$$\langle \textbf{let}\ b_1 = @\mathit{flip}()\ \textbf{in}\ \ldots\ \textbf{let}\ b_n = @\mathit{flip}()\ \textbf{in}\ \varphi\rangle$$

This approach to nondeterministic programming also easily handles irregular search structures, where further tests may depend on outcome of previous "guesses", *e.g.*, for simulating a nondeterministic finite automaton [10].

5.3 Control and prompt

While **shift** and **reset** are very similar to Felleisen's operators **control** and **prompt** [14], there is a significant semantical difference between **shift/reset** and **control/prompt**: the context abstracted by **shift** is determined *statically* by the static program text, while **control** captures the context up to the nearest *dynamically* enclosing **prompt**.

The difference between **shift** and **control** is probably best displayed by the following two characteristic equations:

$$\begin{array}{rcl}\langle @f\,(\textbf{shift}\ c\ \textbf{in}\ M)\rangle &=& \langle \textbf{let}\ c = \lambda x.\langle @f\,x\rangle\ \textbf{in}\ M\rangle \\ \langle @f\,(\textbf{control}\ c\ \textbf{in}\ M)\rangle &=& \langle \textbf{let}\ c = \lambda x.@f\,x\ \textbf{in}\ M\rangle\end{array}$$

Now if f is bound to a procedure that itself abstracts control, the context it will capture with the **shift** semantics is still determined by the context where c is *defined*. On the other hand, with the **control** semantics, a control operator occurring in f will capture the context at the point where c is *applied*; in particular, it can capture part of M. (In the case where f is a simple procedure with no control effects, the effects of the two operators coincide.)

The **shift/reset** approach is based on viewing a program as computing a function expressed in CPS, *i.e.*, on representing control with a function: the continuation. In contrast, **prompt** and **control** were introduced independently of CPS and therefore they admit no such simple static interpretation. Their two denotational descriptions introduce an algebra of control and lead to a representation of continuations as prompt-delimited sequences of activation frames, and their composition as the concatenation of these sequences [18]. Earlier on, prompts were specified with an operational description in terms of textual reductions [14]. In general, these static *vs.* dynamic interpretations lead to different behaviors [8].

In our framework, **reset** naturally is the direct style counterpart of initializing the continuation of a CPS λ-term with the identity function. **Reset** seems to be equivalent to **prompt**, but terms using **control** in general have no CPS counterpart.

5.4 CPS translation of terms that use shift and reset

Let us note that the definitions of **shift** and **reset** do not yield CPS terms (because continuations may be applied to non-values). Therefore the defining terms lose the important property of enforcing strict call-by-value evaluation ensured by proper CPS. However, we can restore that property by translating the defining (pure λ_v-calculus) terms *once more* into CPS, yielding a term in "meta-continuation passing style." Here, the κ is treated as an ordinary functional parameter, while the proper evaluation order is ensured by a new continuation.

For example, and leaving the @-notation aside for readability, the following term occurring in an empty context

$$\begin{array}{l}\textbf{let}\ f = \lambda x.\textbf{shift}\ k\ \textbf{in}\ k(k(x)) \\ \textbf{in}\ 1 + \langle 10 + f(100)\rangle\end{array}$$

gets CPS-transformed into

$$\begin{array}{l}\textbf{let}\ f_c = \lambda x.\lambda k.k(k(x)) \\ \textbf{in}\ 1 + (f_c\,100\,(\lambda v.10 + v))\end{array}$$

As can be noticed, this term is *not* in CPS because the call to f_c is not a tail-call and there is a nested call to k in the definition of f_c. But the second CPS transformation now gives a proper CPS term (with h as the continuation parameter):

$$\begin{array}{l}\textbf{let}\ f_{cc} = \lambda x.\lambda k.\lambda h.k\,x\,(\lambda a.k\,a\,h) \\ \textbf{in}\ f_{cc}\,100\,(\lambda v.\lambda h.h(10 + v))\,(\lambda a.1 + a)\end{array}$$

Remark: Iterating this construction leads to "extended CPS" and a whole hierarchy of control operators [10]. This is the real forte of a CPS-based approach to advanced control structures: we obtain a natural notion of "levels" of control, allowing us to express, *e.g.*, collections over *all* paths of a nondeterministic subcomputation, as defined in Section 5.2.

5.5 Translating control operations in one pass

Following a binding time analysis of the equations of Section 5.2 (as in Section 2), let us express the CPS transformation of control operations using two-level λ-terms:

$$\begin{array}{rcl}
[\![\textbf{escape}\ c\ \textbf{in}\ M]\!] & = & \overline{\lambda}\kappa.\underline{\textbf{let}}\ c' = \underline{\lambda}a.\underline{\lambda}\kappa'.\overline{@}\kappa\, a\ \underline{\textbf{in}}\ \overline{@}[\![M[c \leftarrow c']]\!]\,\kappa \\
[\![\textbf{shift}\ c\ \textbf{in}\ M]\!] & = & \overline{\lambda}\kappa.\underline{\textbf{let}}\ c' = \underline{\lambda}a.\underline{\lambda}\kappa'.\underline{@}\kappa'\,(\overline{@}\kappa\, a)\ \underline{\textbf{in}}\ \overline{@}[\![M[c \leftarrow c']]\!]\,(\overline{\lambda}m.m) \\
[\![\langle M\rangle]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\,(\overline{@}[\![M]\!]\,(\overline{\lambda}m.m))
\end{array}$$

The CPS transformation now introduces **let**-expressions. These could be unfolded by substituting the control abstractions for the identifiers in the translated terms, as we first specified it [10]:

$$\begin{array}{rcl}
[\![\textbf{escape}\ c\ \textbf{in}\ M]\!] & = & \overline{\lambda}\kappa.(\overline{@}[\![M[c \leftarrow c']]\!]\,\kappa)[c' \leftarrow \underline{\lambda}a.\underline{\lambda}\kappa'.\overline{@}\kappa\, a] \\
[\![\textbf{shift}\ c\ \textbf{in}\ M]\!] & = & \overline{\lambda}\kappa.(\overline{@}[\![M[c \leftarrow c']]\!]\,(\overline{\lambda}m.m))[c' \leftarrow \underline{\lambda}a.\underline{\lambda}\kappa'.\underline{@}\kappa'\,(\overline{@}\kappa\, a)]
\end{array}$$

However, these substitutions introduce residual β-redexes when control abstractions are applied within the scope of their declaration. For example,

$$\overline{@}[\![\lambda x.\textbf{shift}\ c\ \textbf{in}\ 1 + (@c\,x)]\!]\,(\overline{\lambda}m.m) \;=\; \lambda x.\lambda c.@(@(\lambda a.\lambda k.1 + (@k\,a))\,x)\,c$$

instead of the preferable $\lambda x.\lambda c.1 + (@c\,x)$

To keep this β-reduction at translation time, but again at the risk of duplicating contexts (as in **shift** c **in** $(@c\,1) + (@c\,2)$), we can distinguish between identifiers that are declared within a λ-abstraction and identifiers that are declared within a control abstraction. Unfortunately, this decision clutters the transformation, much in the same way as introducing "first-class" primitive operators (*cf.* Section 4.1). We adopt the same simple solution: at syntax-analysis time, occurrences of identifiers declared within a control abstraction are guaranteed to occur only in application position, which we single out by tagging this application with **throw**, as in Standard ML of New Jersey [13].

$$\begin{array}{rcl}
[\![\textbf{escape}\ c\ \textbf{in}\ M]\!] & = & \overline{\lambda}\kappa.(\overline{@}[\![M[c \leftarrow c']]\!]\,\kappa)[c' \leftarrow \overline{\lambda}a.\overline{\lambda}\kappa'.\overline{@}\kappa\, a] \\
[\![\textbf{shift}\ c\ \textbf{in}\ M]\!] & = & \overline{\lambda}\kappa.(\overline{@}[\![M[c \leftarrow c']]\!]\,(\overline{\lambda}m.m))[c' \leftarrow \overline{\lambda}a.\overline{\lambda}\kappa'.\overline{@}\kappa'\,(\overline{@}\kappa\, a)] \\
[\![\textbf{throw}\ c\ M]\!] & = & \overline{\lambda}\kappa.\overline{@}[\![M]\!]\,(\overline{\lambda}m.\overline{@}(\overline{@}c\,m)\,\kappa)
\end{array}$$

To avoid dealing with substitutions over translated terms, we can introduce a translation-time environment mapping identifiers to a translation-time λ-abstraction if they are declared within a control abstraction. The following is the corresponding version of the CPS transformer, completed with the translation-time environment. The other equations are unchanged, except for the addition of an environment ρ passively transmitted everywhere.

$$
\begin{array}{rcl}
\mathit{env} & = & \mathrm{var} \rightarrow \mathrm{syntax} \rightarrow [\mathrm{syntax} \rightarrow \mathrm{syntax}] \rightarrow \mathrm{syntax} \\
\llbracket ... \rrbracket & : & \mathit{env} \rightarrow [\mathrm{syntax} \rightarrow \mathrm{syntax}] \rightarrow \mathrm{syntax} \\
\llbracket \mathbf{escape}\ c\ \mathbf{in}\ M \rrbracket & = & \overline{\lambda}\rho.\overline{\lambda}\kappa.\overline{@}(\overline{@}\llbracket M[c\leftarrow c'] \rrbracket\,[c' \mathrel{\overline{\mapsto}} \overline{\lambda}a.\overline{\lambda}\kappa'.\overline{@}\kappa\, a]\rho)\,\kappa \\
\llbracket \mathbf{shift}\ c\ \mathbf{in}\ M \rrbracket & = & \overline{\lambda}\rho.\overline{\lambda}\kappa.\overline{@}(\overline{@}\llbracket M[c\leftarrow c'] \rrbracket\,[c' \mathrel{\overline{\mapsto}} \overline{\lambda}a.\overline{\lambda}\kappa'.\overline{@}\kappa'\,(\overline{@}\kappa\, a)]\rho)\,(\overline{\lambda}m.m) \\
\llbracket \mathbf{throw}\ c\ M \rrbracket & = & \overline{\lambda}\rho.\overline{\lambda}\kappa.\overline{@}(\overline{@}\llbracket M \rrbracket\,\rho)\,(\overline{\lambda}m.\overline{@}(\overline{@}(\overline{@}\rho\, c)\, m)\,\kappa)
\end{array}
$$

Note how all the terms in the translation are static. All administrative reductions are performed at translation time, and no β-redexes are built for applications of control abstractions.

5.6 The problem of name clashes

Now only one problem remain: name clashes. These can occur when a control operator moves part of the context inside the scope of *any* binding construct, as illustrated in the following example:

$$\overline{@}(\overline{@}\llbracket \lambda x.@f\,(\mathbf{escape}\ c\ \mathbf{in}\ \lambda f.\mathbf{throw}\ c\ f) \rrbracket\,\rho_0)\,(\overline{\lambda}v.v) \;\;=\;\; \lambda x.\lambda k.@k\,(\lambda f'.\lambda k'.@(@f\,f')\,k)$$

The simplest solution is thus probably to systematically rename *all* bound variables.

5.7 Direct Transformation into CPS

We can view the CPS transformation of Section 4.1 as an applicative order program expressing the transformation algorithm. This program is expressed in a CPS-like style. This might lead one to believe that this program could be expressed more concisely using a "traditional" control operator like **call/cc**. Unfortunately, this is not quite possible because this program is *not* in CPS. Notably, in the translation of $@M\,N$ the continuation application $\overline{@}\kappa\, a$ is a subterm of the *syntax-constructor* $\underline{\lambda} a. \cdots$. Conversely, in $\llbracket \lambda x.M \rrbracket$, the continuation κ is applied to a potentially non-trivial term. Neither of these situations can arise from the translation of any conventional direct-style term (*i.e.*, λ-calculus + **escape**). However, with **shift/reset** we can express exactly such behavior. Let us rewrite the transformation equations using implicit continuations. The result is displayed in Figure 5.

The result of transforming a term M into CPS in an empty context is given by

$$\overline{\langle} \llbracket M \rrbracket \overline{\rangle}$$

Similarly, the result of transforming M in an arbitrary context is given by

$$\underline{\lambda} k.\overline{\langle} \underline{@} k\,\llbracket M \rrbracket \overline{\rangle}$$

The CPS counterpart of the applicative-order program expressing this transformation algorithm (using the equations of Section 4.1 and the first set of equations of Section 5.5) can be verified easily to coincide with the original translator. Thus, the two transformers yield textually the same output for a given input; only their internal organization differs.

$$
\begin{array}{rcl}
\llbracket x \rrbracket & = & x \\
\llbracket \lambda x.M \rrbracket & = & \underline{\lambda} x.\underline{\lambda} k.\overline{\langle}\underline{@}k\,\llbracket M \rrbracket\overline{\rangle} \\
\llbracket @M\,N \rrbracket & = & \overline{\mathbf{shift}}\ c\ \overline{\mathbf{in}}\ \underline{@}(\underline{@}\llbracket M \rrbracket\llbracket N \rrbracket\,)\,(\underline{\lambda} a.\overline{@}c\,a) \\
\llbracket P \rightarrow M,\ N \rrbracket & = & \overline{\mathbf{shift}}\ c\ \overline{\mathbf{in}}\ \llbracket P \rrbracket \underline{\rightarrow} \overline{\langle}\overline{@}c\llbracket M \rrbracket\overline{\rangle}\underline{,}\,\overline{\langle}\overline{@}c\llbracket N \rrbracket\overline{\rangle} \\
\llbracket q \rrbracket & = & q \\
\llbracket q(M) \rrbracket & = & \underline{q}(\llbracket M \rrbracket) \\
\llbracket q(M, N) \rrbracket & = & \underline{q}(\llbracket M \rrbracket, \llbracket N \rrbracket) \\
\llbracket \mathbf{let}\ x = N\ \mathbf{in}\ M \rrbracket & = & \overline{\mathbf{shift}}\ c\ \overline{\mathbf{in}}\ \underline{\mathbf{let}}\ x' = \llbracket N \rrbracket\ \underline{\mathbf{in}}\ \overline{\langle}\overline{@}c\llbracket M[x \leftarrow x'] \rrbracket\overline{\rangle} \\
\llbracket \mathbf{letrec}\ f = \lambda x.N\ \mathbf{in}\ M \rrbracket & = & \overline{\mathbf{shift}}\ c\ \overline{\mathbf{in}}\ \ \underline{\mathbf{letrec}}\ f' = \underline{\lambda} x.\underline{\lambda} k.\overline{\langle}\underline{@}k\,\llbracket M[f \leftarrow f'] \rrbracket\overline{\rangle} \\
 & & \qquad\qquad\quad \underline{\mathbf{in}}\ \overline{\langle}\overline{@}c\llbracket M[f \leftarrow f'] \rrbracket\overline{\rangle} \\
\llbracket \mathbf{escape}\ k\ \mathbf{in}\ M \rrbracket & = & \overline{\mathbf{shift}}\ c\ \overline{\mathbf{in}}\ \underline{\mathbf{let}}\ k' = \underline{\lambda} v.\underline{\lambda} k''.\overline{@}c\,v\ \underline{\mathbf{in}}\ \overline{\langle}\overline{@}c\llbracket M[k \leftarrow k'] \rrbracket\overline{\rangle} \\
\llbracket \mathbf{shift}\ k\ \mathbf{in}\ M \rrbracket & = & \overline{\mathbf{shift}}\ c\ \overline{\mathbf{in}}\ \underline{\mathbf{let}}\ k' = \underline{\lambda} v.\underline{\lambda} k''.\underline{@}k''\,(\overline{@}c\,v)\ \underline{\mathbf{in}}\ \overline{\langle}\llbracket M[k \leftarrow k'] \rrbracket\overline{\rangle} \\
\llbracket \langle M \rangle \rrbracket & = & \overline{\langle}\llbracket M \rrbracket\overline{\rangle}
\end{array}
$$

Figure 5: Direct-style, one-pass CPS transformation of λ_v-terms

This set of equations can be seen as a meta-circular compiler from a language with the new control operators into its purely functional subset. Alternatively (by omitting the equations for **shift** and **reset**), it translates terms of a Scheme-like language (*i.e.*, λ-calculus + **escape**) into standard CPS. Such a transformation has a practical interest for compiling, *e.g.*, Scheme or Standard ML programs [43, 1], and thus constitutes a significant example of using **shift**/**reset**: even the pure CPS translation is expressed naturally using the new control operators.

As with all meta-circular definitions, we need to bootstrap it. If we have an interpreter for a language with **shift**/**reset**, we can use it to execute the translator on itself, obtaining a CPS transformer written in pure λ-calculus. On the other hand, we can get an interpretive semantics for the extended language by translating a trivial (*i.e.*, defining **shift** in terms of **shift**, *etc.*) self-interpreter into extended CPS [10]. This correspondence helps to ensure consistency between the two methods of language definition.

6 Related Work

6.1 CPS transformation

Two other works have independently employed CPS translations similar to the one presented here. The first one is Appel's CPS transformer in the Standard ML of New Jersey compiler [1]. The second one is Wand's combinator-based compilation technique [45]. But neither motivate their transformer, *e.g.*, as we do in Section 2, nor extend it to control operators or normal order, as we do in Sections 5 and A.

As revealed in the source code of the SML/NJ compiler, the CPS transformer operates in one pass by keeping a translation time continuation, based on η-redexes identical to those in Section 2. However, in contrast to our work, the goal is not to aim at the "exact" continuation-passing counterparts of source programs but to simplify them as much as possible, even if some of the simplifications correspond to source reductions. Still no particular care is taken to avoid building extraneous η-redexes, such as those pointed out in Section 4.2. Instead, the compiler relies on a powerful and blind simplifier of CPS terms that processes both these redexes and what corresponds to source reductions. It would be interesting to measure whether and how much our more precise CPS transformer relieves the simplifier.

Similarly, devising a particular representation of run time procedures and their application (instead of our $\underline{\lambda}$ and $\underline{@}$), Wand also compiles programs based on a CPS transformer with the same η-redexes as in Section 2, but again without motivating them.

Today Sabry and Felleisen are also investigating the CPS transformation in one pass, and Lawall and the first author are investigating the inverse "Direct Style" transformation [9, 11, 40].

6.2 Primitive operators

Most CPS-based compilers [43, 1, 45] and program analyzers [41] also use continuation-passing forms of even the primitive operators. However, the practical justification of such a "radical CPS" transform is not completely clear. In particular, the oft-quoted advantage of having explicit names for all subexpressions can be realized equally well with let-expressions. Clearly, side-effecting operators need to be tied down by explicit conversion to continuation-passing variants. However, expressing "trivial computations" like tuple construction/destruction or arithmetic in CPS introduces unnecessary sequentialization and obscures the fact that such computations can be rearranged or even eliminated or duplicated without affecting the meaning of the program.

> **Remark:** A possible problem here concerns primitive operators like division that can signal error conditions, but are otherwise "pure". Clearly, the translation should preserve any exception-raising behavior of the original program, and not compromise the order-of-evaluation independence of CPS terms. On the other hand, going to full CPS for such almost-functional operators may still be overkill, for the reasons outlined above. However, it often seems possible to factor the original operator into two aspects: the control behavior (the process of computation, possibly error-raising) to be expressed in CPS and the pure (and hence freely rearrangable) function computed, in which case we actually get the best of both worlds.

6.3 Control operators

From Reynolds's **escape** to **call/cc** in Scheme, control operators are nicely introduced within the CPS transformation [38, 16]. However, because CPS appears to constrain expressive power, Felleisen and others have successively proposed new control operators to compose continuations [15] and to limit their extent [14]. As later shown by Sitaram and Felleisen [42], inclusion of control delimiters is also necessary to obtain fully abstract models of control for CPS models with

escape. The motivation for **shift** and **reset** was somewhat different: rather than devising new theories, new models, and new representations of control, we have set out to explore CPS more thoroughly. In particular, **shift** and **reset** are introduced together as representing composition and identity on continuation functions respectively; proper CPS form is restored by iterating the CPS transformation [10].

6.4 Partial evaluation

Partial evaluation (or more accurately: program specialization [25]) makes heavy use of binding time information to process the static and the dynamic semantics of source programs [6], as we do here. Recent work by Bondorf and the first author emphasize the issues of code duplication and termination properties [3], and use the technique of enumerating finitary constructs [2], as we do in Sections 2 and 4.2. The latter is also central to Shivers's work on higher-order flow analysis [41]. All these concepts were pervasive in our derivation of a one-pass CPS transformer. In particular, the notion of a two-level λ-calculus as advocated in Nielson and Nielson's TML [34] proves useful to develop and to express new CPS transformations that distinguish properly between translation-time and run-time constructs.

With respect to partial evaluation, this development illustrates the connection between a CPS transformer and a λ-calculus interpreter expressed in CPS. The former is a two-level version of the latter.

Moreover, our derivation illustrates a new trend in partial evaluation: using CPS to improve binding time properties of source programs, leading to better specialization [7]. Work is going on to further automate the process.

7 Conclusion and Issues

As proven constructively in this paper, transforming λ-terms into CPS can be expressed in one pass by moving administrative redexes to translation time in a context-free way. While the actual transformation algorithm seems to have been independently discovered several times in slightly different forms, we believe that ours is the first systematic derivation and analysis of its correctness and properties. The translation is easily extended to the usual constructs of applicative order functional languages and also to account for control operators. And using two control operators **shift** and **reset** derived naturally from the CPS formalism, the translation can be formulated even more concisely and directly.

The role of continuations in programming language design and implementation has long been dominated by pragmatic concerns. In the last few years, however, the subject has seen renewed theoretical interest, especially with the introduction of concepts and methods from mathematical logic and category theory, *e.g.*, [22, 31, 19]. We believe that any investigation of advanced control structures based on the CPS transform will be able to pick up and integrate such developments more directly than a free-standing approach derived from more intuitively "desirable" operational behavior could. And in fact, recent developments seem to support this conviction [32].

Moreover, there is a close relationship between computational monads [30] and "generalized CPS", as suggested in "Abstracting Control" [10] and properly formalized by Wadler [44]. Effectively, this implies that CPS-based control operators like **shift** and **reset** can by themselves uniformly express a rich class of computational behaviors, including partiality, nondeterminism, and state. It seems natural to take this as another indication that both the theoretical and practical significance of functional representations of control will only grow stronger in the years to come.

Acknowledgements

We are grateful to the editor and the three referees. Thanks are also due to Karoline Malmkjær and Dave Schmidt for their patience and to Chet Murthy for his enthusiasm.

References

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

[2] Anders Bondorf. Automatic autoprojection of higher-order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[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. The Scheme 311 compiler, an exercise in Denotational Semantics. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, Austin, Texas, August 1984.

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

[6] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In POPL'91 [36], pages 14–24.

[7] 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.

[8] Olivier Danvy. Programming with tighter control. *Special issue of the BIGRE journal: Putting Scheme to Work*, (65):10–29, July 1989.

[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 [26], pages 151–160.

[11] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In LFP'92 [27].

[12] Olivier Danvy and Carolyn L. Talcott, editors. *Proceedings of the ACM SIGPLAN Workshop on Continuations*, San Francisco, California, June 1992. Technical report STAN-CS-92-1426, Stanford University.

[13] Bruce F. Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In POPL'91 [36], pages 163–173.

[14] Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988.

[15] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, Bloomington, Indiana, February 1987.

[16] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. Reasoning with continuations. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 131–141, Cambridge, Massachusetts, June 1986. IEEE.

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

[18] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988.

[19] Andrzej Filinski. Linear continuations. In POPL'92 [37], pages 27–38.

[20] 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.

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

[22] 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.

[23] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, 1983.

[24] Bob Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Danvy and Talcott [12]. Technical report, Stanford University.

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

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

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

[28] Chris Mellish and Steve Hardy. Integrating Prolog in the POPLOG environment. In John A. Campbell, editor, *Implementations of PROLOG*, pages 147–162. Ellis Horwood, 1984.

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

[30] 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.

[31] Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Department of Computer Science, Cornell University, 1990.

[32] Chethan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In Danvy and Talcott [12]. Technical report, Stanford University.

[33] Flemming Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69(2):117–242, 1989.

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

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

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

[37] *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992. ACM Press.

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

[39] Jon G. Riecke. Should a function continue? Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1989.

[40] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In LFP'92 [27].

[41] Olin Shivers. The semantics of Scheme control-flow analysis. In Paul Hudak and Neil D. Jones, editors, *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, pages 190–198, New Haven, Connecticut, June 1991. ACM, ACM Press.

[42] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In LFP'90 [26], pages 161–175.

[43] 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.

[44] Philip Wadler. The essence of functional programming. In POPL'92 [37], pages 1–14.

[45] 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*, Pittsburgh, Pennsylvania, March 1991. 7th International Conference.

$$
\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\lambda x.M]\!] & = & \lambda\kappa.@\kappa\,(\lambda x.[\![M]\!]) \\
[\![@M\,N]\!] & = & \lambda\kappa.@[\![M]\!]\,(\lambda m.@(@m\,[\![N]\!])\,\kappa)
\end{array}
$$

Figure 6: Plotkin's CPS transformation of λ_n-terms

$$
\begin{array}{rcl}
[\![...]\!] & : & [\text{syntax} \rightarrow \text{syntax}] \rightarrow \text{syntax} \\
[\![x]\!] & = & \overline{\lambda}\kappa.\underline{@}x\,(\underline{\lambda}a.\overline{@}\kappa\,a) \\
[\![\lambda x.M]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\,(\underline{\lambda}x.\underline{\lambda}k.\overline{@}[\![M]\!]\,(\overline{\lambda}m.\underline{@}k\,m)) \\
[\![@M\,N]\!] & = & \overline{\lambda}\kappa.\overline{@}[\![M]\!]\,(\overline{\lambda}m.\underline{@}(\underline{@}m\,(\underline{\lambda}k.\overline{@}[\![N]\!]\,(\overline{\lambda}n.\underline{@}k\,n)))\,(\underline{\lambda}a.\overline{@}\kappa\,a))
\end{array}
$$

Figure 7: One-pass CPS transformation of λ_n-terms

A Call by Name

Let us consider Plotkin's equational specification for transforming a λ_n-term into CPS [35], as displayed in Figure 6. We want to apply the method of Section 2 to stage the CPS$_n$-transformation. Again, taken literally, this translation yields artificial redexes that must be post-reduced in a second pass, although the size explosion is not as drastic as for call-by-value. For example, translating

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

yields

$$\lambda k.@k\,(\lambda f.\lambda k.@k\,(\lambda x.\lambda k.@k\,(\lambda y.\lambda k.@(\lambda k.@f\,(\lambda m.@(@m\,y)\,k))\,(\lambda m.@(@m\,x)\,k))))$$

whose post-reduction yields

$$\lambda k.@k\,(\lambda f.\lambda k.@k\,(\lambda x.\lambda k.@k\,(\lambda y.\lambda k.@f\,(\lambda m.@(@m\,y)\,(\lambda m.@(@m\,x)\,k)))))$$

This CPS counterpart of the λ_n-term can be equally evaluated using call-by-name or call-by-value. As a simple consequence, the continuation can be implemented as a strict function without altering the meaning of the original λ_n-term.

Can we subject the original specification to the same treatment as in Section 2 and get a one-pass CPS transformer? The answer is yes but the resulting transformation shows less immediate success. The original specification can be transformed into the one in Figure 7 which does not build any β-redex.

The result of transforming a term M into CPS in an empty context is then given by

$$\overline{@}[\![M]\!]\,(\overline{\lambda}m.m)$$

Again, this specification, viewed as an applicative order program, can be re-expressed using **shift** and **reset**.

$$
\begin{array}{rcl}
[\![...]\!] & : & [\text{syntax} \rightarrow \text{syntax}] \rightarrow \text{syntax} \\
[\![x]\!] & = & \overline{\lambda}\kappa.\underline{@}x\,(\underline{\lambda}a.\overline{@}\kappa\,a) \\
[\![\lambda x.M]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\,(\underline{\lambda}x.[\![M]\!]'') \\
[\![@M\,N]\!] & = & \overline{\lambda}\kappa.\overline{@}[\![M]\!]\,(\overline{\lambda}m.\underline{@}(\underline{@}m\,[\![N]\!]'')\,(\underline{\lambda}a.\overline{@}\kappa\,a)) \\
 & & \\
[\![...]\!]'' & : & \text{syntax} \\
[\![x]\!]'' & = & x \\
[\![\lambda x.M]\!]'' & = & \underline{\lambda}k.\underline{@}k\,(\underline{\lambda}x.[\![M]\!]'') \\
[\![@M\,N]\!]'' & = & \underline{\lambda}k.\overline{@}[\![M]\!]\,(\overline{\lambda}m.\underline{@}(\underline{@}m\,[\![N]\!]'')\,k)
\end{array}
$$

Figure 8: One-pass, "properly tail-recursive" CPS transformation of λ_n-terms

However this new CPS translation suffers from a deficiency, as pinpointed by the following observation.

Observation 2 *A λ_n-term and its CPS counterpart are related as follows:*

- *A variable is translated into one application and one λ-abstraction.*
- *A λ-abstraction is translated into two λ-abstractions and one application.*
- *An application is translated into three applications and two λ-abstractions.*

Whereas the two last points also hold for Plotkin's specification, the first point reveals that the new translation actually produces more redexes! However, the new specification only produces more η-redexes, which are not nearly as hard to get rid of as the β-redexes produced by the original translation, as outlined now.

η-redexes are only constructed in tail-contexts, for identifiers occurring as λ-abstraction bodies, and as arguments of functions (this corresponds to the ALGOL 60 situation of "suspending a suspension"). As in Section 2.6, they can eliminated at translation time by duplicating the rules (*cf.* Figure 8).

> **Rationale:** The auxiliary translation $[\![M]\!]''$ is used for the special case $\underline{\lambda}k.\overline{@}[\![M]\!]\,(\overline{\lambda}m.\underline{@}k\,m)$, thereby avoiding the construction of extraneous η-redexes.

The result of transforming a term M into CPS in an empty context is then given by

$$\overline{@}[\![M]\!]\,(\overline{\lambda}m.m)$$

whereas the result of transforming a term M into CPS in a dynamic context is given by

$$[\![M]\!]''$$

By construction, this instrumented new translation yields terms without β-redexes, in one pass.

$$
\begin{array}{rcl}
\llbracket ... \rrbracket & : & [\text{syntax} \rightarrow \text{syntax}] \rightarrow \text{syntax} \\
\llbracket x \rrbracket & = & \overline{\lambda}\kappa.\underline{@}x\,(\underline{\lambda}a.\overline{@}\kappa\,a) \\
\llbracket \lambda x.M \rrbracket & = & \overline{\lambda}\kappa.\overline{@}\kappa\,(\underline{\lambda}k.\underline{\lambda}x.\overline{@}\llbracket M\rrbracket' k) \\
\llbracket @M\,N \rrbracket & = & \overline{\lambda}\kappa.\overline{@}\llbracket M\rrbracket\,(\overline{\lambda}m.\underline{@}(\underline{@}m\,(\underline{\lambda}a.\overline{@}\kappa\,a))\,\llbracket N\rrbracket'') \\
 & & \\
\llbracket ... \rrbracket' & : & \text{syntax} \rightarrow \text{syntax} \\
\llbracket x \rrbracket' & = & \overline{\lambda}k.\underline{@}x\,k \\
\llbracket \lambda x.M \rrbracket' & = & \overline{\lambda}k.\underline{@}k\,(\underline{\lambda}k.\underline{\lambda}x.\overline{@}\llbracket M\rrbracket' k) \\
\llbracket @M\,N \rrbracket' & = & \overline{\lambda}k.\overline{@}\llbracket M\rrbracket\,(\overline{\lambda}m.\underline{@}(\underline{@}m\,k)\,\llbracket N\rrbracket'') \\
 & & \\
\llbracket ... \rrbracket'' & : & \text{syntax} \\
\llbracket x \rrbracket'' & = & x \\
\llbracket \lambda x.M \rrbracket'' & = & \underline{\lambda}k.\underline{@}k\,(\underline{\lambda}k.\underline{\lambda}x.\overline{@}\llbracket M\rrbracket' k) \\
\llbracket @M\,N \rrbracket'' & = & \underline{\lambda}k.\overline{@}\llbracket M\rrbracket\,(\overline{\lambda}m.\underline{@}(\underline{@}m\,k)\,\llbracket N\rrbracket'')
\end{array}
$$

Figure 9: One-pass, "properly tail-recursive" CPS transformation of λ_n-terms with continuations first

A.1 Continuations first

Making continuations occur first introduces a new opportunity for extraneous η-redexes in residual CPS terms. The result of transforming a term can occur (1) in function position where the argument is a static lambda that will be applied to a static continuation; (2) in function position where the argument is a static lambda that will be applied to a dynamic continuation; and (3) not in function position. These cases can be handled by a suitable series of tests on intermediate result or again by duplicating the rules as in Figure 9. Notice how $\llbracket ... \rrbracket$ terms correspond to case (1), $\llbracket ... \rrbracket'$ terms correspond to case (2), and $\llbracket ... \rrbracket''$ terms correspond to case (3). This suggests that a Clinger-style compiler for λ_n-terms would be proven using a triple induction hypothesis [4].

As usual, the result of transforming a term M into CPS in an empty context is then given by

$$\overline{@}\llbracket M\rrbracket\,(\overline{\lambda}m.m)$$

whereas the result of transforming a term M into CPS in a dynamic context is given by

$$\underline{\lambda}k.\overline{@}\llbracket M\rrbracket' k$$

or better, by

$$\llbracket M\rrbracket''$$

This final translation yields terms without β-redexes nor new η-redexes, in one pass.