Denotational style correctness of a CPS-Transform based compiler

Arthur Nunes-Harwitt
ABSTRACT
Correctness is a crucial property for compilers; programmers rely on it when writing code. Ideally, correctness should be proved. Work on compiler correctness has focused on direct translation strategies. However, in practice, the continuation passing style (CPS) transform (or a variant) is often used in the translation process. Here a simple source language and its CPS-transform based compiler are introduced. A tractable proof for this compiler is presented, including a denotational proof of the correctness of a CPS-transform. The benefits of the proof are discussed.

1. INTRODUCTION
Compilers translate source code to target code. Today, most programs are written using a high-level language that is compiled. It is widely agreed[1] that compilers must be correct, since otherwise the generated code is not reliable. A formal correctness proof bolsters one’s confidence that a compiler generates target code that does what is specified by the source code.

Typically, unless explicitly stated otherwise, “compiler correctness” focuses on the middle-end of the compiler: the part that accepts an abstract syntax tree and generates a linear (or almost linear) intermediate-language consisting of relatively low-level instructions which can be easily translated into the machine language of choice. There are two approaches when writing a middle-end: direct and staged. The direct approach translates the source to the target directly without any intermediate transformations, whereas a staged approach breaks the translation into n stages (where n > 1). Each stage transforms its input into output suitable for the next stage’s input; the output language may be different from the input language or the same.

Compiler correctness has been studied extensively using both operational[11, 19] and denotational techniques[23, 4, 16, 17]. And while separate back-end stages have been studied[16, 17], the middle-end is invariably studied with a direct approach. In practice, however, the middle-end is often staged. In particular, many middle-ends make use of the CPS-transform[18, 10], or a variation such as the A-normal form (ANF) transform[9], as an initial stage when generating output (e.g. Scheme[12, 21] implementations such as Rabbit[22], Orbit[13], Twobit/Larceny[5], DrScheme[8], Chicken[3]).

This paper defines a three-staged compiler (middle-end) that makes use of the CPS-transform; it models the staged middle-ends used in practice. Compiler correctness is proved using denotational techniques. Section 2 defines the source language. Section 3 defines the target language. Section 4 defines the intermediate CPS language. Section 5 discusses linearization, the CPS-transform, a normalization stage, and the stage that translates from the intermediate language to the target code. Section 6 discusses the correctness theorems; details of the proofs are in the appendix. Section 7 concludes and mentions possible future work.

2. SOURCE LANGUAGE
The source language is the call-by-value λ-calculus with constants. These constants include numbers as well as operators that act on the numbers, such as addition. Abstractions model user-defined procedures, and applications model procedure calls. Note that there are a number of layers of syntax defined. Values must be defined as a distinct subclass of terms. Another useful subclass is W, which is the set of atomic values.

Evaluation contexts are defined below. A context is an expression with a hole in it, where the hole is written []. As the name suggests, the term that is to be evaluated next fills the hole of an evaluation context. Evaluation contexts are needed in section 5.
\[ \mathcal{E} ::= [] \mid \mathcal{E}[(V)] \mid \mathcal{E}[(M)] \]

What’s missing?

- Procedures with multiple parameters
  These can be simulated using Currying.

- A conditional expression such as ‘if’
  This expression can be simulated using constant operators and thunks. In fact, if one understands the Clinger-Hansen\[6\] dictum “lambda is label” to be bi-directional, and since ‘if’ is implemented using labels, then perhaps one ought to understand ‘if’ this way.

- Recursion
  Recursion can be implemented using the Y-operator

- Loopy constructs
  Loopying can be implemented in terms of recursion.

- Assignment, exceptions, etc.
  This paper will not deal with such features.

There are several ways of defining the semantics of the language above. In particular, a denotational semantics gives a meaning to each kind of term based on the meaning of each of its sub-terms. While this method may seem indirect, since the meaning is expressed in terms of a meta-language, the advantage is that the meta-language is designed for proving equations.

### Denotational Semantics

\[ D[c] = \lambda \rho.\lambda \kappa. (\kappa \, \mathcal{K}(c)) \]
\[ D[x] = \lambda \rho.\lambda \kappa. (\kappa \, \rho(x)) \]
\[ D[(\lambda x. M)] = \lambda \rho.\lambda \kappa. (\lambda v. D[M][\rho[x \mapsto v]]) \]
\[ D[(M \, N)] = \lambda \rho.\lambda \kappa. D[M][\rho \lambda n. (\lambda n. ((m \, n) \, \kappa))] \]

\[ \rho \in \text{Variables} \to E \]
\[ \kappa \in \text{E} \to A \]
\[ \mathcal{K} \in \text{Values} \to E \]

\[ E \text{ is a semantic value} \]
\[ A \text{ is an answer} \]

The function \( \rho \) serves as the environment mapping variables to semantic values, \( \kappa \) serves as the continuation function mapping semantic values to answers, where an answer might be a semantic value or a coarser representation of one, and the function \( \mathcal{K} \) maps constants to semantic values. These definitions are given for the intuition they provide. Neither the domains \( E \) and \( A \) nor the details of the functions \( \rho \) and \( \kappa \) are crucial for the proofs, so these details will be ignored.

### 3. TARGET LANGUAGE

The target language is meant to resemble the instruction set of a load-store machine architecture. For example, both load \( c, x \) and move \( x, z \) instructions are very similar to what one finds on a load-store machine\(^1\). The instruction instr: \( c, x, z \) may look unfamiliar, but that is only because it represents a family of instructions. For example, add \( x, z \) (which adds one to the contents of \( x \) and puts the result in \( z \)) would be expressed by instr:succ \( x, z \), and sub \( x, z \) (which subtracts one from the contents of \( x \) and puts the result in \( z \)) would be expressed by instr:pred \( x, z \). Other instructions are more complex; for example, a single instruction call \( y, x, z \) creates an activation record, and then passes the contents of \( x \) to the procedure \( y \) refers to. The instruction makeClosure \( x, \{S\}, y \) is also complex; given the formal parameter \( x \) and the code sequence \( S \), it creates a closure and puts it in \( y \). Further makeClosure \( x, \{S\}, y \) has internal structure; the instruction is part of a code sequence but also contains \( S \), which is a code sequence. I have in mind that the internal structure represents a pointer; to emphasize this I introduce additional notation and wrap the internal sequence in braces. Furthermore, although this sort of internal structure is easy to eliminate, I leave it in because it makes reasoning about the language simpler.

#### Syntax

\[ c \in \text{Constants} \]
\[ x, y, z \in \text{Variables} \]
\[ I \in \text{Instructions} ::= \]
\[ \text{load} \ c, x \quad | \quad \text{move} \ x, z \]
\[ \text{makeClosure} \ x, \{S\}, y \quad | \quad \text{instr: c, x, z} \]
\[ \text{call} \ y, x, z \]

\[ S \in \text{Sequence} ::= \text{return} \ x \ | \ \text{tailCall} \ y, x \ | \ I; S \]

### Denotational Semantics

\[ D[\text{return} \ x] = \lambda \rho.\lambda \kappa. (\kappa \, \rho(x)) \]
\[ D[\text{tailCall} \ y, x] = \lambda \rho.\lambda \kappa. (\rho(y) \, \rho(x) \, \kappa) \]
\[ D[\text{load} \ c, x; S] = \lambda \rho.\lambda \kappa. D[S][\rho[x \mapsto \mathcal{K}(c)] \kappa] \]
\[ D[\text{move} \ x, z; S] = \lambda \rho.\lambda \kappa. D[S][\rho[z \mapsto \rho(x)] \kappa] \]
\[ D[\text{makeClosure} \ x, \{S\}, y; S] = \lambda \rho.\lambda \kappa. D[S][\rho[y \mapsto (\lambda v. D[S][\rho[v \mapsto \rho(x)] \kappa)] \kappa] \]
\[ D[\text{instr: c, x, z}; S] = \lambda \rho.\lambda \kappa. ((\mathcal{K}(c) \, \rho(x)) \, (\lambda v. D[S][\rho[z \mapsto \rho(v)] \kappa)] \kappa) \]
\[ D[\text{call} \ y, x, z; S] = \lambda \rho.\lambda \kappa. ((\rho(y) \rho(x)) \, (\lambda v. D[S][\rho[z \mapsto \rho(v)] \kappa)] \kappa) \]

### 4. INTERMEDIATE LANGUAGE

Many compilers work by first translating the source code into an intermediate language, which is often the CPS-language\[2\]. The key difference being that the semantics below makes \( x \) and \( z \) environment variables rather than registers. Nevertheless, I am imagining that registers would be used to implement the environment.
5.1 Direct Linearization

The direct approach translates the source to a linear form without any intermediate transformations. A local translation is frequently used for direct linearization[4, 16, 17, 14]; however, a CPS-based translation is also possible[23].

5.1.1 Local

Given an abstract syntax tree, local linearization implicitly involves the following four ideas.

1. Placing an explicit call operator at the end of an application and moving constant operators to the end of an application do not affect the semantics.

2. If we add a marker to do a debug-step operation[15], we see that there are only results to the left of the marker. Further, taking a step either involves moving into an application where the marker ultimately moves over a value, which looks like a push, or reducing an application where some of the values to the left of the marker are eliminated, which looks like a pop.

3. Because the arity of constant operators in known, and the arity of others can be recorded with the call operator, the form from (1) can be written with little or no nesting (in a linear form).

4. Because of (2), values can be translated into push-instructions for a stack-machine.

Consider the following example with the abstract-syntax represented as an s-expression: (f 2 3 (+ 4 5)).

(f 2 3 (4 5 +) call1) is an equivalent representation.

f 2 3 4 5 + call4 is also equivalent.

push f; pushi 2; pushi 3; pushi 4; pushi 5; add; call 4 is simply another variation.

These ideas are easily combined into a local linearization algorithm. An example local linearization is given below for the source language from section 2.

\[
\begin{align*}
\text{pushi } & c \\
\text{push } & x \\
\text{pushClosure } & x, \{L[M]\} \\
\text{L}[c M] & = L[M]; L[\text{const}[c]] \\
\text{L}[M N] & = \text{call2 } if \ M \neq c
\end{align*}
\]

5.1.2 CPS

It is possible to use continuation passing style and higher-order assembly language (HOAL)[23] to develop a non-local linearization algorithm. K is the syntactic continuation called the sequel; the initial sequel is (λx.halt v). An example direct CPS linearization is given below for the source language from section 2; the target is the HOAL variation of the target language presented in section 3.

\[
\begin{align*}
\text{load } & c K \\
\text{move } & x K \\
\text{makeClosure}(λx.K[M](λv.return v)) & K \\
\text{call } & M[N]; L[N]; \text{call2 } if \ M \neq c
\end{align*}
\]
5.2 Staged Linearization
The staged approach consists of stages such that each stage translates an input form to an output form. For the initial stage, the input form is the source language; for the final stage the output is the target language. Frequently, the CPS-transform (or a variant such as the ANF-transform) is used as the initial stage.

I’ll characterize CPS-like transforms by giving an intuitive description of the ANF-transform; however, the compiler given below will make use of the CPS version.

ANF linearization involves two key ideas: (1) There is exactly one application that will be evaluated first, and (2) this expression can be pulled out, and its value named without changing the meaning of the expression.

Consider the following example with the abstract-syntax represented as an s-expression: \((\text{let } ((x (+ 4 5))) (f 2 3 x))\). One can imagine this form being translated into the following: \(\text{add } 4, 5, x; \text{tailCall } f, 2, 3, x\)

An example ANF-transform is given below for the source language from section 2, where \(E\) is the context in which the application to be evaluated first is found[9].

\[
\begin{align*}
A[V] &= V^# \\
A[(V V')] &= (V^# V'^#) \\
A[E[(V V')]] &= \text{let } x = (V^# V'^#) \text{ in } A[E[x]] \\
&\quad \text{if } E \neq [] \text{ and } x \text{ is fresh}
\end{align*}
\]

\[
\begin{align*}
W^# &= W \\
(\lambda x. M)^# &= (\lambda x. A[M])
\end{align*}
\]

The output form of \(A\) is discussed by Sabry and Wandler[20]. It turns out, that it is isomorphic to the CPS-language. Thus, the transform above is easily modified so as to generate the intermediate language in section 4.

\[
\begin{align*}
A'[V] &= (k V^*) \\
A'[(V V')] &= ((V^* V'^*) k) \\
A'[E[(V V')]] &= ((V^* V'^*) (\lambda x. A'[E[x]]) \text{ if } E \neq [] \text{ and } x \text{ is fresh}
\end{align*}
\]

\[
\begin{align*}
W^* &= W \\
(\lambda x. M)^* &= (\lambda x. k A'[M])
\end{align*}
\]

\(A'\) is in fact the CPS-transform. In practice, this algorithm is inefficient; an efficient first-order one-pass CPS-transform by Danvy and Nielsen[7] is given below.

\[
P \in A \setminus \text{Values}
\]

\[
\begin{align*}
F[V]Q_e &= (Q_e V^*) \\
F[(V V')]Q_e &= ((V^* V'^*) Q_e) \\
F[(V P)]Q_e &= F[P](\lambda x.(V^* x) Q_e)) \\
x \text{ is fresh} \\
F[(P V)]Q_e &= F[P](\lambda y.(y V^*) Q_e)) \\
y \text{ is fresh} \\
F[(P P')]Q_e &= F[P](\lambda x. F[P'](\lambda x.(y x) Q_e))) \\
x \text{ and } y \text{ are fresh}
\end{align*}
\]

\[
\begin{align*}
W^* &= W \\
(\lambda x. M)^* &= (\lambda x. k F[M] k)
\end{align*}
\]

5.3 The Staged Compiler
The first stage of the compiler is \(F\) which does a CPS-transform of the source language; its output is the mostly linear CPS-language. Note that since \(F\) requires a continuation parameter to compute its result, it is supplied an initial continuation parameter: the fixed continuation variable \(k\).

The second stage of the compiler is \(N\) (defined below) which normalizes the CPS-language code by naming constants and abstractions; the output is a subset of the CPS-language. The next stage would require a longer case analysis for its definition without this normalization stage.

\[
\begin{align*}
N[(k x)] &= (k x) \\
N[(k R_k)] &= ((\lambda x. (k x)) R_k) \\
x \text{ is fresh} \\
N[((\lambda x. M) v)] &= ((\lambda x. N[M]) v) \\
x \text{ is fresh} \\
N[((V x) Q_v)] &= ((V x) Q^* v) \\
N[((R v) Q_v)] &= ((\lambda x. (V x) Q^* v) R_v) \\
x \text{ is fresh}
\end{align*}
\]

\[
\begin{align*}
W^1 &= W \\
(\lambda x. k N[M])^1 &= (\lambda x. k N[M])
\end{align*}
\]

Assuming that the CPS-code is in the subset determined by \(N\), it closely resembles the target language. The function \(B\) (defined below) maps this subset of the CPS-language to the target language.

\[
\begin{align*}
B[(k x)] &= \text{return } x \\
B[((\lambda x. M) c)] &= \text{load } c, x; B[M] \\
B[((\lambda x. M) x)] &= \text{move } x, z; B[M] \\
B[[\lambda y. M]] &= \text{makeClosure } y, B[N] \end{align*}
\]

The compiler is then the function \(T\) defined below. It is simply the composition of the three stages.
\[ T[M] = B[N[F[M]k]] \]

6. COMPILER CORRECTNESS

Since correct compilers have always been direct, the key to the proof involves a key inductive hypothesis or perhaps a couple of simultaneous hypotheses. I considered the following inductive hypothesis for the compiler presented above.

For any \( M, Q_c, \rho, \kappa, z \notin \text{fv}(M) \cup \text{fv}(Q_c), \) \( D_c[B[N[F[M]k]]]\rho_k = D[M] \rho(\lambda v . D_c[B[N[F[z]Q_c]]] \rho(z \mapsto v) \kappa). \)

It appears that the induction goes through; however, the proof is long and unwieldy, with subcases, and subcases with subcases, and so on.

Instead, I decided to go with a modular proof. There is a result for each stage of the compiler which establishes that it is correct. The result for each stage is reasonably short and straightforward. When these results are put together, they establish the correctness of the compiler.

The first stage is the CPS-transform. Corollary 1 below establishes the correctness of the CPS-transform; as far as I know, this is the first denotational correctness proof of the CPS-transform. Corollary 1 follows from lemma 1 by letting \( Q_c = k. \)

**Lemma 1.** For any \( M, V, Q_c, \rho, \kappa, \)

(i) \( \kappa(V[V^*]) = D[V]\rho_k \)

(ii) \( D_c[F[M]Q_c]\rho_k = D[M] \rho(D_Q[Q_c]\rho_k) \)

**Corollary 1.** For any \( M, D_c[F[M]k] = D[M]. \)

The second stage is normalization of CPS-terms. Corollary 2 below establishes the correctness of normalization of the output of the CPS-transform. It follows from lemma 2 by letting \( M_c = F[M]k. \)

**Lemma 2.** For any \( V_c, Q_c, M_c, \)

(i) \( D_c[V_c] = D[V_c] \)

(ii) \( D_Q[Q_c] = D_Q[Q_c] \)

(iii) \( D_c[M_c] = D_c[N[M_c]] \)

**Corollary 2.** For any \( M, D_c[F[M]k] = D_c[N[F[M]k]]. \)

Corollary 3 establishes that the denotation of the output of both stages one and two is the same as the denotation of the original term. It follows from corollary 1, corollary 2, and the transitive property.

**Corollary 3.** For any \( M, D_c[N[F[M]k]] = D[M]. \)

The third stage translates a subset of the CPS-language to the target language. Corollary 4 establishes that the denotation of the three stages is the same as the denotation of only the first two stages. It follows from lemma 3 and lemma 4 by letting \( M_c = N[F[M]k]. \)

**Lemma 3.** For any \( M_c, N[M_c] \in \text{domain}(B). \)

**Lemma 4.** For any \( M_c \in \text{domain}(B), D_c[B[M_c]] = D_c[M_c]. \)

**Corollary 4.** For any \( M, D_c[B[N[F[M]k]]] = D_c[N[F[M]k]]. \)

Now the correctness theorem is easy. Putting together corollary 3 and corollary 4, we get \( D_c[B[N[F[M]k]]] = D[M]. \)

The result then follows from the definition of \( T. \)

**Theorem 1.** For any \( M, D_c[T[M]] = D[M]. \)

7. CONCLUSION

I have presented a proof that a simple staged CPS-transform based compiler is correct. The proof has several desirable features:

It is tractable. Each stage has a correctness proof that is bite-sized. Further, the arguments are clear, and proofs rely only on structural induction.

Since one of the stages is the CPS-transform, the proof includes a proof of correctness of the CPS-transform. Typically, the correctness of the CPS-transform is established operationally. The denotational proof is no more complicated, and may have benefits beyond its use in the correctness result presented here.

The proof is modular, and so gains all the advantages of modularity: should there be any errors, they are isolated within a particular component; components can be replaced; and components can be added.

There are some features that are absent in this work. The compiler presented does no optimization. Because of the modular nature of the proof, it should be unproblematic to add a correct optimization stage. All that would be required to extend this proof to a proof of correctness of an optimizing compiler would be a proof of the correctness of the optimizer itself. The source language presented does not have assignment. It would be worthwhile to add assignment to make these proofs a more realistic platform on which to build. The addition of these features is future work.

8. ACKNOWLEDGEMENTS

I would like to thank Melissa Nunes-Harwitt and Richard Zanibbi for carefully reading previous drafts of this paper, and Will Clinger for looking carefully at my proofs.
9. REFERENCES


APPENDIX

A. PROOFS

Below are the proofs of the key theorems.

*Proof of lemma 1*

By simultaneous induction on parts (i) and (ii).

Part (i):

- **Suppose** \( V = c \).
  
  \[
  (\kappa \ D V \ [\[ c \] \]) \rho = (\kappa \ k(c)) \rho = (\lambda x. (\kappa' x c)) \rho = D[\[ c \]] \rho.
  \]

- **Suppose** \( V = x \).
  
  \[
  (\kappa \ D V \ [\[ x \] \]) \rho = (\kappa (\rho) x) = (\lambda x. (\kappa' x (\rho) x)) \rho = D[\[ x \]] \rho.
  \]

- **Suppose** \( V = (\lambda x. M) \).
  
  \[
  (\kappa \ D V \ [\[ (\lambda x. M) \] \]) \rho = (\kappa (\lambda x.D[\[ x \] \ M] \rho x \rightarrow v)) \rho = (\kappa (\lambda v.D[\[ v \] \ M] \rho x \rightarrow v(\lambda x. \rho x k x')) \rho = (\kappa (\lambda v.D[\[ v \] \ M] \rho x \rightarrow v(\lambda x. v x'))) \rho = D[\[ (\lambda x. M) \] \rho x \rightarrow v(\lambda x. v x')) \rho = D[\[ (\lambda x. M) \] \rho x \rightarrow v(\lambda x. v x')) \rho = (\kappa (\lambda v.D[\[ v \] \ M] \rho x \rightarrow v(\lambda x. v x'))) \rho = (\lambda (\lambda x.D[\[ x \] \ M] \rho x \rightarrow v(\lambda x. v x'))) \rho = D[\[ (\lambda x. M) \] \rho x \rightarrow v(\lambda x. v x')] \rho = D[\[ (\lambda x. M) \] \rho x \rightarrow v] \rho.
  \]

Part (ii):

- **Suppose** \( M = V \).
  
  \[
  D_0[F[V] Q] \rho = D_0[Q[V] V^*] \rho = ((D_2 Q[V^*] \rho) (D_0 Q[V] V^* \rho) \rho) \rho = D[V] \rho (D_2 Q[V^*] \rho) \rho.
  \]
• Suppose $M = (V \cdot V')$.
  
  $D_V[(V \cdot V')_o] = D_V[(V \cdot V')_{o2}]$
  
  $= (((\lambda x \cdot D_V[V \cdot V'])_o \cdot (\lambda y \cdot x - y)) \cdot (\lambda y \cdot x - y))$
  
  $= (((\lambda x \cdot D_V[V \cdot V'])_o \cdot (\lambda y \cdot x - y)) \cdot (\lambda y \cdot x - y))$
  
  $= (((\lambda x \cdot D_V[V \cdot V'])_o \cdot (\lambda y \cdot x - y)) \cdot (\lambda y \cdot x - y))$
  
  $= (((\lambda x \cdot D_V[V \cdot V'])_o \cdot (\lambda y \cdot x - y)) \cdot (\lambda y \cdot x - y))$

  $QED$

Proof of lemma 2

By simultaneous induction on parts (i), (ii), and (iii).

Part (i):

• Suppose $V_e = W$.
  
  Since $W^1 = W$, it follows immediately that $D_V[W^1] = D_V[W]$.

• Suppose $V_e = (\lambda x \cdot k \cdot M_e)$.
  
  $D_V[(\lambda x \cdot k \cdot M_e)]$
  
  $= \lambda x \cdot D_V[M_e]_o \cdot \rho[x \mapsto v]
  
  $= \lambda v \cdot \lambda \lambda \cdot D_V[N \cdot \lambda M_e]_o \cdot \rho[x \mapsto v\kappa]
  
  $= D_V[(\lambda x \cdot M_e)]$

  Part (ii):

• Suppose $Q_e = k$.
  
  Since $k^1 = k$, it follows immediately that $D_Q[k^1] = D_Q[k]$.

• Suppose $Q_e = (\lambda x \cdot M_e)$.

By structural induction on the domain of $B$.

• Suppose $M_e = (k \cdot x)$.
  
  $D_B[B \cdot (k \cdot x)]$
  
  $= B \cdot [return \ x]
  
  $= \lambda x \cdot B \cdot [\rho(x)]
  
  $= D_B[(k \cdot x)]$

• Suppose $M_e = ((\lambda x : N_e) \ c)$.
  
  $D_B[(\lambda x : N_e) \ c]$
  
  $= D_B[(\lambda x : N_e)] \ c
  
  $= \lambda x \cdot D_B[N_e] \ c
  
  $= \lambda x \cdot D_B[N_e] \ c
  
  $= \lambda x \cdot D_B[N_e] \ c
  
  $= D_B[(\lambda x : N_e) \ c]$
Suppose $M'_c = (\lambda x. N_c) (\lambda x. k N'_c)$.

$D_c[B((\lambda x. N_c) x)]]$

$= D_c[\text{move } x, z; B[N_c]]$

$= \lambda p. \lambda k. D_c[B[N_c]] p[z \mapsto \rho(x)] k$

$= \lambda p. \lambda k. D_c[N_c] p[z \mapsto \rho(x)] k$

$= \lambda p. \lambda k. (\lambda u. D_u[N_c] p[z \mapsto v] k) \rho(x))$

$= D_c[(((\lambda z. N_c) x) x)]$

Suppose $M'_c = ((\lambda w. k N_c) x) k$.

$D_c[B((\lambda w. k N_c) x) x)]]$

$= D_c[\text{move } x, w; B[N_c]]$

$= \lambda p. \lambda k. D_c[B[N_c]] p[w \mapsto \rho(x)] k$

$= \lambda p. \lambda k. D_c[N_c] p[w \mapsto \rho(x)] k$

$= \lambda p. \lambda k. ((\lambda u. D_u[N_c] p[w \mapsto u] (\lambda w. k N_c) x) k)$

$= D_c[(((\lambda w. k N_c) x) x) x)]$

Suppose $M'_c = ((\lambda w. k N_c) x) x) k$.

$D_c[B((\lambda w. k N_c) x) x)]]$

$= D_c[\text{move } x, z; B[N_c]]$

$= \lambda p. \lambda k. (\lambda (c x) (\lambda z. N_c)) (\lambda v. D_v[\text{return } z] p[z \mapsto v] k))$

$= \lambda p. \lambda k. (\lambda (c x) (\lambda z. N_c)) (\lambda v. (\lambda \kappa. v))$

$= D_c[(((\lambda w. k N_c) x) x) x)]$

Suppose $M'_c = ((\lambda w. k N_c) x) x) k$.

$D_c[B((\lambda w. k N_c) x) x)]]$

$= D_c[\text{call } y, x; \text{call } y, x; [B[N_c]]]

$= \lambda p. \lambda k. ((\lambda (c x) (\lambda z. N_c)) p[z \mapsto v] k))$

$= \lambda p. \lambda k. (\lambda (c x) (\lambda z. N_c)) p[z \mapsto v] k))$

$= D_c[(((\lambda w. k N_c) x) x) x)]$

Suppose $M'_c = ((\lambda w. k N_c) x) x) k$.

$D_c[B((\lambda w. k N_c) x) x)]]$

$= D_c[\text{call } y, x, z; [B[N_c]]]

$= \lambda p. \lambda k. (\lambda (c x) (\lambda z. N_c)) p[z \mapsto v] k))$

$= \lambda p. \lambda k. (\lambda (c x) (\lambda z. N_c)) p[z \mapsto v] k))$

$= D_c[(((\lambda w. k N_c) x) x) x)]$

Suppose $M'_c = ((\lambda w. k N_c) x) x) k$.

$D_c[B((\lambda w. k N_c) x) x)]]

$= D_c[\text{call } y, x, z; [B[N_c]]]

$= \lambda p. \lambda k. (\lambda (c x) (\lambda z. N_c)) p[z \mapsto v] k))$

$= \lambda p. \lambda k. (\lambda (c x) (\lambda z. N_c)) p[z \mapsto v] k))$

$= D_c[(((\lambda w. k N_c) x) x) x)]

Suppose $M'_c = ((\lambda w. k N_c) x) x) k$.

$D_c[B((\lambda w. k N_c) x) x)]]

$= D_c[\text{call } y, x, z; [B[N_c]]]

$= \lambda p. \lambda k. (\lambda (c x) (\lambda z. N_c)) p[z \mapsto v] k))$

$= \lambda p. \lambda k. (\lambda (c x) (\lambda z. N_c)) p[z \mapsto v] k))$

$= D_c[(((\lambda w. k N_c) x) x) x)]

Suppose $M'_c = ((\lambda w. k N_c) x) x) k$.

$D_c[B((\lambda w. k N_c) x) x)]]

$= D_c[\text{call } y, x, z; [B[N_c]]]

$= \lambda p. \lambda k. (\lambda (c x) (\lambda z. N_c)) p[z \mapsto v] k))$

$= \lambda p. \lambda k. (\lambda (c x) (\lambda z. N_c)) p[z \mapsto v] k))$

$= D_c[(((\lambda w. k N_c) x) x) x)]

QED