2012-11-26

A CPS-Like Transformation of Continuation Marks

Kimball Richard Germane
Brigham Young University - Provo

A CPS-Like Transformation of Continuation Marks

Kimball R. Germane

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Jay McCarthy, Chair
Sean Warnick
Dennis Ng

Department of Computer Science

Brigham Young University

December 2012

ABSTRACT

A CPS-Like Transformation of Continuation Marks

Kimball R. Germane
Department of Computer Science, BYU
Master of Science

Continuation marks are a programming language feature which generalize stack inspection. Despite its usefulness, this feature has not been adopted by languages which rely on stack inspection, e.g., for dynamic security checks. One reason for this neglect may be that continuation marks do not yet enjoy a transformation to the plain λ-calculus which would allow higher-order languages to provide continuation marks at little cost.

We present a CPS-like transformation from the call-by-value λ-calculus augmented with continuation marks to the pure call-by-value λ-calculus. We discuss how this transformation simplifies the construction of compilers which treat continuation marks correctly. We document an iterative, feedback-based approach using Redex. We accompany the transformation with a meaning-preservation theorem.

Keywords: continuation marks, continuation-passing style, Redex

ACKNOWLEDGMENTS

Thanks to Jay McCarthy for advising me and my wife Bethany for her patience.

Table of Contents

1 Introduction **1**

2 Continuation marks **4**

2.1 Example 5

3 λ-calculus **8**

4 λ_v and λ_{cm} **11**

4.1 λ_v 11

4.2 λ_{cm} 11

5 CPS transformation **14**

5.1 Introduction 14

5.2 Example 16

6 $\mathcal{C}$ **18**

6.1 Intuition 18

6.2 Concept 19

6.3 Initiation 21

6.4 Some Final Subtleties 22

6.5 Definition of $\mathcal{C}$ 22

6.6 Definition of $\mathcal{C}$ in CPS 24

6.7 Example 25

7 Testing **29**

7.1 Redex . . . 29

7.1.1 Toy Language . . . 29

7.2 Flavors of λ . . . 33

7.2.1 λ_v . . . 34

7.2.2 λ_{cm} . . . 35

7.3 Transformation definition . . . 36

7.3.1 Direct Style . . . 36

7.3.2 Continuation-passing Style . . . 37

7.4 Transformation testing . . . 39

8 Proof **40**

9 Conclusion **45**

References **46**

A Proof of Lemma 1 **48**

A.1 Application Form . . . 48

A.2 wcm Form . . . 53

A.3 ccm Form . . . 57

A.4 Value Form . . . 58

A.5 Variable Form . . . 60

B Reductions **63**

B.1 Application Form . . . 63

B.2 wcm form . . . 64

B.3 ccm form . . . 66

B.4 Value Form . . . 66

B.5 Variable Form . . . 67

Chapter 1

Introduction

Thesis: A CPS-like global transformation can compile the λ-calculus with continuation marks into the plain λ-calculus in a semantics-preserving way.

Numerous programming language instruments rely on stack inspection to function. Statistical profilers sample the stack regularly to record active functions, algebraic steppers observe the stack to represent the evaluation context of an expression, and debuggers naturally require consistent access to the stack. Each of these relies on implementation-specific information and must be maintained as the instrumented language undergoes optimizations and ventures across platforms. This makes these tools brittle and increases the porting cost of the language ecosystem. Each of these examples would benefit from a generalized stack-inspection mechanism available within the instrumented language itself. If written in such an enhanced language, each instrument would be more robust, more easily modified, and would port for free.

Continuation marks [4] are a programming language feature which generalizes stack inpection. Not only do they dramatically simplify correct instrumentation [6], they have been used to allow inspection-based dynamic security checks in the presence of tail-call optimization [5] and to express aspect-oriented programming in higher-order languagues [16].

In spite of their usefulness, continuation marks have remained absent from programming languages at large. One reason for this is that retrofitting virtual machines to accommodate the level of stack inspection continuation marks must provide is expensive, especially when the virtual machines use the host stack for efficiency.

For example, the ubiquitous JavaScript is an ideal candidate for the addition of continuation marks. However, as the lingua franca of the web, it has numerous mature implementations which have been heavily optimized; to add continuation marks to JavaScript amounts to modifying each implementation upstream, to say nothing of amending the JavaScript standard. (Clements et al. successfully added continuation marks in Mozilla's Rhino compiler [7], but it remains a proof-of-concept.)

To avoid this roadblock, we instead take a macro-style approach; that is, we enhance the core language with facilities to manipulate continuation marks and desugar the enhanced language back to the core. To make our desugaring transformation apply to many languages, we define it over the λ-calculus, the common core of most higher-order languages. With such a transformation, language semanticists do not need to reconcile the feature with other features in the language (provided they have already done so with λ) and their compiler writers do not need to worry about complicating their implementations (for the same reason).

The λ-calculus is a Turing-complete formal logic based on variable binding and substitution. Where the Turing machine model of computation is machine-centric, the λ-calculus model is language-centric. Thus, it conveniently serves as an intermediate representation for a compiler or a base from which to define higher-level languages. The standard reference to the λ-calculus is Barendregt [2].

The continuation-passing style (CPS) transformation is actually a family of language transformations designed to make certain analyses simpler. Every member of this family shares a common trait: their performance augments each function with an additional formal parameter, the *continuation*, a functional representation of currently pending computation. Functions in CPS never explicitly return; instead, they call the continuation argument with their result. Because no function ever returns, function calls are the final act of the caller. Thus, all calls are tail calls. The CPS transformation then simplifies programs by representing all control and data transfer uniformly and explicitly. In general, the "spirit" of the CPS transformation is to represent all transfers of control uniformly [15].

We take the core of computation, the λ-calculus, and add facilities to manipulate continuation marks. These two together comprise a language which we term λ_{cm}. By expressing λ_{cm} in terms of the plain λ-calculus, we uncover the meaning of continuation marks in a pure computational language absent other language features and implementation details. To do this, we construct $\mathcal{C}$, a transformation from λ_{cm} programs to λ-calculus programs in the spirit of CPS. We use Redex [8] to test candidate transformations for correctness. This allows us to increase our confidence in a functioning transformation but cannot demonstrate correctness. We address this shortcoming by providing and proving a meaning preservation theorem.

Chapter 2

Continuation marks

There are certain tools that are indispensable to some programmers that concern the behavior of their programs: debuggers, profilers, steppers, etc. Without these tools, these programmers cannot justify the adoption of a language, however compelling it might otherwise be. Traditionally, these tools are developed at the same level the language is, privy to incidental implementation detail, precisely because that detail enables these tools to function. This is problematic for at least two reasons. First, it couples the implementation of the tool with the implementation of the language, which increases the cost to port to other platforms. If users become dependent upon these tools, it can stall the advancement of the language and the adoption of new language features. Second and more critical, it makes these tools unsound. For instance, debuggers typically examine programs which have been compiled without optimizations. In general, this means that the debugged program has different behavior than the deployed program. This is obviously undesirable.

It is desirable to implement such tools at the same level as the language, removing dependency upon the implementation and instead relying on definitional and behavioral invariants. Continuation marks are a language-level feature that provide the information necessary for these tools to function. Furthermore, languages which require stack inspection to enforce security policies (*Java*, *C#*) or support aspect oriented programming (*aspectj*) can be defined in terms of a simpler language with continuation marks [5, 16].

Continuation marks originated in PLT Scheme (now Racket [10]) as a stack inspection mechanism. In fact, the *Java* and *C#* languages rely on a similar stack inspection to enforce

security policies of which continuation marks can be seen as a generalization. Surprisingly, continuation marks can be encoded in any language with exception facilities [13].

The feature of continuation marks itself is accessible via two surface level syntactic forms: **with-continuation-mark** and **current-continuation-marks**.

In Scheme-like syntax, a **with-continuation-mark** expression appears as (**with-continuation-mark** *key-expr value-expr body-expr*). The evaluation of this expression proceeds by first evaluating *key-expr* to *key* and *value-expr* to *value*. Thereafter, *body-expr* is evaluated during which *value* is associated with *key*.

In the same Scheme-like syntax, a **current-continuation-marks** expression appears as (**current-continuation-marks** *key-list*). This expression evaluates to a list of all the present key-value associations referenced in *key-list*. The **with-continuation-mark** form admits a notion of ordering–inner associations are more recent than outer ones–and this in reflected in the list yielded by the **current-continuation-marks**.

Importantly, the result of **current-continuation-marks** provides no evidence of any portion of the dynamic context lacking continuation marks with the specified keys. This preserves the ability to perform optimizations without exposing details which would render the optimizations unsound. This also requires special consideration of a language that supports tail call optimization (which is not an optimization in the above sense since its behavior is defined in the semantics of the language).

2.1 Example

The canonical example to illustrate the behavior of continuation marks in the presence and absence of tail call optimization is the factorial function.

Figure 2.1 illustrates the properly recursive variant of the factorial function. In this variation, a cascade of multiplication operations builds as the recursive calls are made. Each multiplication is pending computation of which the machine must keep track.

```
(define (fac n)
  (if (= n 0)
      1
      (* n (fac (- n 1)))))
```

Figure 2.1: The definitionally recursive factorial function

```
(define (fac-tr n acc)
  (if (= n 0)
      acc
      (fac-tr (- n 1) (* n acc))))
```

Figure 2.2: A tail-recursive variant of the factorial function

Figure 2.2 illustrates the tail recursive variant of the factorial function. In contrast to the function in figure 2.1, this variation performs the multiplication before the recursive call. Because the function has no pending computations after the evaluation of the recursive call, the execution context need not grow. Such a call is said to be in tail position.

Figures 2.3 and 2.4 represent these two variants of the factorial function augmented with continuation marks. Using these definitions, the result of (**fac** 3) would be

```
(((fac 1)) ((fac 2)) ((fac 3)))
6
```

whereas the result of (**fac-tr** 3 1) would be

```
(((fac 1)))
6
```

```
(define (fac n)
  (if (= n 0)
      (begin
        (display (current-continuation-marks 'fac))
        1)
      (with-continuation-mark 'fac n (* n (fac (- n 1))))))
```

Figure 2.3: The definitionally recursive factorial function augmented with continuation marks

```
(define (fac-tr n acc)
  (if (= n 0)
      (begin
        (display (current-continuation-marks '(fac)))
        acc)
      (with-continuation-mark 'fac n (fac-tr (- n 1) (* n acc)))))
```

Figure 2.4: The tail-recursive factorial function augmented with continuation marks

This difference is due to the growing continuation in the properly recursive **fac**. Each call to **fac** has a pending computation–namely, the multiplication–after the recursive call and so each necessitates the creation of additional evaluation context. The effect of this additional context is that each annotation is applied to new, "blank" context, so all the previous annotations are preserved. In the tail-recursive variant, there is no pending computation and therefore no additional evaluation context. In this instance, the previous mark is overwritten with the new.

Chapter 3

λ-calculus

The λ-calculus [2] is a Turing-complete system of logic extensively used as a formal system for expressing computation. Terms in the λ-calculus are defined inductively; they take the form of variables x drawn from an infinite set, abstractions $(\lambda\,(x)\,M)$ where M is itself a λ-calculus term, and applications $(M\,N)$ where M and N are λ-calculus terms.

Variables in the λ-calculus are either *free* or *bound*. A variable x is free if it does not reside in the scope of a *binding instance* of x. Otherwise, x is bound. Terms with no free variables can be called closed terms, combinators, or programs. The fact that variables are drawn from an infinite set means that, given an arbitrary λ-calculus term, we can always obtain a *fresh variable*, a variable not present in the term at hand. This is critical as we will see shortly.

In an abstraction of the form $(\lambda\,(x)\,M)$, x is a binding instance which binds all free occurrences of x in the body M. To a first approximation, abstractions are functions. For instance, the identity function can be expressed as $(\lambda\,(x)\,x)$ where x is *any* variable. Thus, there are an infinite number of ways to express the identity function: $(\lambda\,(x)\,x)$, $(\lambda\,(y)\,y)$, $(\lambda\,(z)\,z)$, etc. A consequence of this is that terms which are not syntactically equivalent may be semantically equivalent. This fact naturally gives rise to the notion of α-equivalence which captures the idea that consistent renaming of bound variables, and their binding instances, does not change the meaning of a term.

There is some subtlety in valid renaming as there is the possibility of *variable capture* which occurs when the new name of a binding instance is identical to that of some variable

already present in the body. For instance, in the term $(\lambda\,(x)\,y)$, if each x is renamed to y, we obtain the term $(\lambda\,(y)\,y)$, a fundamentally different term. For this reason, we need to take special care when we rename variables, which we will need to do regularly.

One of the ways abstractions approximate functions is that we can apply them to arguments. This is signified simply by juxtaposition of function (or operator) and argument (or operand). In the correct context, an application of the form $(M\,N)$ can be *reduced* in which the operator M is applied to the operand N. For M of the form $(\lambda\,(x)\,M')$ for some M', this entails the substitution of every free occurrence of x in M' with N. The notation we adopt for this is $M'[x \leftarrow N]$. For instance, the application $((\lambda\,(x)\,x)\,y)$ signifies $x[x \leftarrow y]$ and so reduces to y. Here, we must protect against another strain of variable capture. As an example, consider the reduction of $((\lambda\,(x)\,(\lambda\,(y)\,x))\,y)$. If we reduce naively, we obtain $(\lambda\,(y)\,y)$ which does not reflect the intended meaning of the reduction–the argument y has been captured by the abstraction, an act which destroys its meaning within the environment. In order to avoid this, we must rename capturing abstractions in M to be outside the set of free variables of N. Within the greater term $((\lambda\,(x)\,(\lambda\,(y)\,x))\,y)$, we rename y to z in $(\lambda\,(x)\,(\lambda\,(y)\,x))$ obtaining $(\lambda\,(x)\,(\lambda\,(z)\,x))$. The subsequent reduction of $((\lambda\,(x)\,(\lambda\,(z)\,x))\,y)$ to $(\lambda\,(z)\,y)$ correctly reflects the intended meaning of the original term.

In the λ-calculus, evaluation occurs during reduction, and reduction is merely application. There is, however, yet more subtlety of which we must be aware: namely, in which contexts applications are performed and terms evaluated. In the call-by-value λ-calculus, denoted λ_v, evaluation of operands occurs before application. In contrast, in the call-by-name λ-calculus, denoted λ_n, application is performed as soon as the operator is resolved.

For instance, in $((\lambda\,(x)\,x)\,((\lambda\,(y)\,y)\,(\lambda\,(z)\,z)))$,

$$\begin{aligned}
&((\lambda\,(x)\,x)\,((\lambda\,(y)\,y)\,(\lambda\,(z)\,z)))\\
\rightarrow_{\lambda_v}&((\lambda\,(x)\,x)\,(\lambda\,(z)\,z))\\
\rightarrow_{\lambda_v}&(\lambda\,(z)\,z)
\end{aligned}$$

whereas

$$\begin{aligned}
&((\lambda\,(x)\,x)\,((\lambda\,(y)\,y)\,(\lambda\,(z)\,z)))\\
\rightarrow_{\lambda_n}&((\lambda\,(y)\,y)\,(\lambda\,(z)\,z))\\
\rightarrow_{\lambda_n}&(\lambda\,(z)\,z)
\end{aligned}$$

Although both terms reduce to the same term in this example, this distinction is not merely pedantic: terms may reduce definitively in one reduction regime and fail to reduce completely in the other! Consider $((\lambda\,(x)\,(\lambda\,(y)\,y))\,((\lambda\,(x)\,(x\,x))\,(\lambda\,(x)\,(x\,x))))$ where

$$((\lambda\,(x)\,(\lambda\,(y)\,y))\,((\lambda\,(x)\,(x\,x))\,(\lambda\,(x)\,(x\,x)))) \rightarrow_{\lambda_n} (\lambda\,(y)\,y)$$

but

$$\begin{aligned}
&((\lambda\,(x)\,(\lambda\,(y)\,y))\,((\lambda\,(x)\,(x\,x))\,(\lambda\,(x)\,(x\,x))))\\
\rightarrow_{\lambda_v}&((\lambda\,(x)\,(\lambda\,(y)\,y))\,((\lambda\,(x)\,(x\,x))\,(\lambda\,(x)\,(x\,x))))\\
\rightarrow_{\lambda_v}&\cdots
\end{aligned}$$

Historically at least, the call-by-name and call-by-value reduction regimes underlie the distinction between so-called lazy and eager languages.

One final observation we should make about the λ-calculus is which terms denote values. A value should be, in a sense, irreducible and that criterion disqualifies applications from being considered as values. A value should should not merely be a placeholder for arbitrary values, and that criterion disqualifies lone variables from being considered as values.[1] Thus, we shall consider abstractions to be the sole form values can take in the λ-calculus, habituating ourselves to the idea that functions are data.

[1]To be precise, a value is a *closure*: an irreducible λ-calculus term paired with an environment which provides values for constituent free variables.

Chapter 4

λ_v and λ_{cm}

4.1 λ_v

The λ_v language is merely the call-by-value λ-calculus [14].

Figure 4.1 presents the language terms and evaluation contexts of λ_v. The definition of e specifies the form of terms in λ_v similar to our previous discussion of the λ-calculus. The definition of E specifies evaluation contexts which, like terms, are defined inductively. An evaluation context is either empty (denoted by $\bullet$), an application wherein the operator is being evaluated, or an application wherein the operand is being evaluated.

Figure 4.2 presents the sole semantic definition of λ_v, the meaning of application.

4.2 λ_{cm}

We now consider an extension of λ_v with facilities to manipulate continuation marks, introduced by Pettyjohn et al. [13], which we term λ_{cm}. As an extension of λ_v, it inherits its definitions of language terms, evaluation contexts, and semantics.

$$
\begin{array}{ll}
E = (E\, e) & e = (e\, e) \\
\quad (v\, E) & \quad x \\
\quad \bullet & \quad v \\
 & v = (\lambda\, (x)\, e)
\end{array}
$$

Figure 4.1: λ_v forms

$$E[(\lambda x.e)\, v] \rightarrow E[e[x \leftarrow v]]$$

Figure 4.2: λ_v evaluation rule

$$\begin{array}{ll} E = (\text{wcm}\, v\, F) & e = (e\, e) \\ \quad F & \quad x \\ F = \bullet & \quad v \\ \quad (E\, e) & \quad (\text{wcm}\, e\, e) \\ \quad (v\, E) & \quad (\text{ccm}) \\ \quad (\text{wcm}\, E\, e) & v = (\lambda\, (x)\, e) \end{array}$$

Figure 4.3: λ_{cm} forms

Figure 4.3 presents the syntactic forms of λ_{cm}. Definitions of E and F signify evaluation contexts. The separation of E from F prevents $(\text{wcm}\, v\, F)$ contexts from directly nesting within the entire evaluation context to enforce proper tail-call behavior. The definition of e is identical to that of λ_v with the addition of contionuation mark forms $(\text{wcm}\, e\, e)$ and (ccm). (λ_{cm} expresses unkeyed marks which obviates the need to specify a key to which a value will be associated. Hence, the **wcm** and **ccm** forms need one parameter fewer than their Scheme counterparts.)

Figure 4.4 presents the semantics of λ_{cm} in the form of a list of reduction rules. Rule 1 defines the meaning of application as inherited from λ_v. Rule 2 defines the tail behavior of the **wcm** form. Rule 3 expresses that the **wcm** form takes on the value of its body. Finally, rule 4 defines the value of the **ccm** form in terms of the χ metafunction.

The definition of the χ-metafunction is given in figure 4.5. Conceptually, the χ-metafunction traverses the context from the outside in, accumulating values as it encounters $(\text{wcm}\, v\, E)$ contexts. Since the χ metafunction is defined over evaluation contexts of λ_{cm}, its domain corresponds to the definitions of E and F in figure 4.3.

$$E[(\lambda x.e)\, v] \to E[e[x \leftarrow v]] \quad (1)$$
$$E[(\text{wcm}\, v\, (\text{wcm}\, v'\, e))] \to E[(\text{wcm}\, v'\, e)] \quad (2)$$
$$E[(\text{wcm}\, v\, v')] \to E[v'] \quad (3)$$
$$E[(\text{ccm})] \to E[\chi(E)] \quad (4)$$

Figure 4.4: λ_{cm} semantics

$$\chi(\bullet) = \text{nil}$$
$$\chi(E[(\bullet\, e)]) = \chi(E)$$
$$\chi(E[(v\, \bullet)]) = \chi(E)$$
$$\chi(E[(\text{wcm}\, \bullet\, e)]) = \chi(E)$$
$$\chi(E[(\text{wcm}\, v\, \bullet)]) = v : \chi(E)$$

Figure 4.5: Definition of χ metafunction

In λ_{cm}, evaluation of an application form proceeds as

$$\begin{aligned}
E[(e_0\, e_1)] &\to_{\lambda_{cm}} E[(\bullet\, e_1)][e_0] \\
&\to^*_{\lambda_{cm}} E[(\bullet\, e_1)][v_0] \\
&\to_{\lambda_{cm}} E[(v_0\, e_1)] && (1) \\
&\to_{\lambda_{cm}} E[(v_0\, \bullet)][e_1] \\
&\to^*_{\lambda_{cm}} E[(v_0\, \bullet)][v_1] \\
&\to_{\lambda_{cm}} E[(v_0\, v_1)] && (2) \\
&= E[((\lambda\, (x)\, e'_0)\, v_1)] && \text{for some } x \text{ and } e'_0 \\
&\to_{\lambda_{cm}} E[e'_0[x \leftarrow v_1]] \\
&= E[e_2] && \text{for some } e_2
\end{aligned}$$

Steps 1 and 2 denote the insertion of a value into the hole in the context.

Chapter 5

CPS transformation

5.1 Introduction

The CPS transform is actually a family of language transformations derived from Plotkin [14] designed to simplify programs by representing all data and control flow uniformly and explicitly. This, in turn, simplifies compiler construction and analyses such as optimization and verification [1, 15]. The standard variation of CPS adds a formal parameter to every function definition and an argument to every call site.

As an example, consider once again the two variants of the factorial function, sans continuation marks, given earlier. In CPS, the properly-recursive variant can be expressed as

```
(define (fac n k)
  (if (= n 0)
      (k 1)
      (fac (- n 1) (λ (acc) (k (* n acc))))))
```

and the tail-recursive variant as

```
(define (fac-tr n acc k)
  (if (= n 0)
      (k acc)
      (fac-tr (- n 1) (* n acc) k)))
```

(For clarity, we have treated "primitive" functions–equality comparison, subtraction, and multiplication–in a direct manner. In contrast, a comprehensive CPS transformation would affect *every* function.)

Notice that, in the first variation, each recursive call receives a newly-constructed k encapsulating additional work to be performed at the completion of the recursive computation.

In the second, k is passed unmodified, so while computation occurs within each context, no *additional* computation pends. From this example, we see that the CPS representation is ideal for understanding tail-call behavior as it is explicit that the continuation is preserved by the tail call.

The purpose of CPS does not lie solely in pedagogy, however. The reification of and consequent ability to directly manipulate the continuation is powerful, analogous in power to the ability to *capture* a continuation which some languages provide. In Scheme, this is accomplished with **call/cc**, short for "call with current continuation". This call takes one argument which itself is a function of one argument. **call/cc** calls its argument, passing in a functional representation of the current continuation–the continuation present when **call/cc** was invoked. This continuation function takes one argument which is treated as the result of **call/cc** and runs this continuation to completion.

As a simple example,

```
(+ 1 (call/cc
       (λ (k)
         (k 1))))
```

returns 2. In effect, invoking k with 1 is the same as replacing the entire **call/cc** invocation with 1.

Much of the power of **call/cc** lies in the manifestation of the continuation as a function, giving it first-class status. It can be passed as an argument in function calls, invoked, and, amazingly, reinvoked at leisure. It is this reinvokeability that makes **call/cc** the fundamental unit of control from which all other control structures can be built, including generators, coroutines, and threads.

In direct style, the definition of **call/cc** is conceptually

```
(define call/cc
  (λ (f)
    (f (get-function-representing-continuation))))
```

where **get-function-representing-continuation** is an opaque function which leverages sweeping knowledge of the language implementation. The CPS definition is notably simpler:

```
(define call/cc
  (λ (f k)
    (f (λ (x w) (k x)) k)))
```

Within the definition of **call/cc**, we define an anonymous function which, when given a value x and continuation w, applies the captured continuation k to x.

Variations of the standard CPS transformation make the expression of certain control structures more straightforward. For instance, the "double-barrelled" CPS transformation is a variation wherein each function signature receives not one but two additional formal parameters, each a continuation. One application of this particular variation is error handling with one continuation argument representing the remainder of a successful computation and the other representing the failure contingency. It is especially useful in modelling exceptions and other non-local transfers of control in situations where the computation might fail. In general, the nature of the CPS transformation allows it to untangle complicated, intricate control structures.

Similar transformations exist which express other programming language features such as security annotations [17] and control structures such as procedures, exceptions, labelled jumps, coroutines, and backtracking. On top of other offerings, this places it in a category of tools to describe and analyze programming language features. (This category is also occupied by Moggi's computational λ-calculus–monads [12].)

5.2 Example

We will now focus our attention on a CPS transform defined over λ_v.

A CPS transformation is a global syntactic transformation of language terms. Recall that terms in the λ-calculus take the form of lone variables x, λ-abstractions $(\lambda\,(x)\,M)$, and

applications $(M\,N)$ where M and N are themselves λ-calculus terms. A comprehensive CPS transform definition then need only specify transformations for these three categories. As an example, consider Fischer's CPS transform [9]:

$$\begin{aligned}
\mathcal{F}[x] &= (\lambda\,(k)\,(k\,x)) \\
\mathcal{F}[(\lambda\,(x)\,M)] &= (\lambda\,(k)\,(k\,(\lambda\,(x)\,\mathcal{F}[M]))) \\
\mathcal{F}[(M\,N)] &= (\lambda\,(k)\,(\mathcal{F}[M]\,(\lambda\,(m)\,(\mathcal{F}[N]\,(\lambda\,(n)\,((m\,n)\,k))))))
\end{aligned}$$

Fischer's CPS transform abstracts each term in the λ-calculus: lone variables wait on a continuation, abstractions receive a degree of indirection, and even applications, the sole reduction facility of the λ-calculus, become abstractions. In essence, terms become suspended in wait of a continuation argument. By priming a term so-transformed with a continuation function–even as simple as the identity function–we instigate a cascade of computation.

In a sense, the CPS transform contaminates abstractions, the values of the λ-calculus. For example, consider the transformation of $\lambda x.x$

$$\begin{aligned}
\mathcal{F}[\lambda x.x] &= \lambda k.k\,(\lambda x.\mathcal{F}[x]) \\
&= \lambda k.(k\,\lambda x.\lambda k.(k\,x))
\end{aligned}$$

If we apply this term to the identity function, it reduces as

$$\begin{aligned}
&\lambda k.(k\lambda x.\lambda k.(k\,x))\;\lambda y.y \\
\rightarrow_{\lambda_v} &\lambda y.y\;\lambda x.\lambda k.(k\,x) \\
\rightarrow_{\lambda_v} &\lambda x.\lambda k.(k\,x)
\end{aligned}$$

The result of this reduction contains residue of the CPS transform. This must be accounted for when attempting to formally relate direct and continuation-passing style terms.

Chapter 6

$\mathcal{C}$

A CPS-like global transformation can compile the λ-calculus with continuation marks into the plain λ-calculus in a semantics-preserving way.

In order to demonstrate this, we must first clarify (and later, formalize) the semantics-preservation property. We will define a transformation from λ_{cm} to λ_v and term it $\mathcal{C}$, as in *compile*, since we are, in essence, compiling away continuation marks. In order to preserve the meaning of λ_{cm}, $\mathcal{C}$ must commute with evaluation. More precisely, $\mathcal{C}$ satisfies

$$\begin{array}{ccc} p & \rightarrow^*_{\lambda_{cm}} & v \\ \downarrow\mathcal{C} & & \downarrow\mathcal{C} \\ \mathcal{C}[p] & \rightarrow^*_{\lambda_v} & \mathcal{C}[v] \end{array}$$

for any program $p \in \lambda_{cm}$.

The burden of demonstration is then reduced to the existence of $\mathcal{C}$. We do this by construction.

6.1 Intuition

The essence of λ_{cm} is that programs can apply information to and observe information about the context in which they are evaluated. Programs in λ_v have no such facility. We can simulate this facility by explicitly passing contextual information to each term as it is evaluated. We can define $\mathcal{C}$ to transform **wcm** directives to manipulate this information and **ccm** directives to access it. Intuitively, we can transform λ_{cm} programs to mark-passing style.

However, marks alone do not account for the tail-call behavior specified by rule 2 of figure 4.4. Since tail-call behavior is observable (if indirectly) by λ_{cm} programs, we must also provide to each term information about the position in which it is evaluated. Specifically, each transformed **wcm** directive must be notified whether it is evaluated in tail position of an enclosing **wcm** directive as it must behave specially if so. Thus, in addition to passing the current continuation marks, the transform should pass a flag to each term indicating whether it is evaluated in tail position of a **wcm** directive.

These two pieces of information suffice to correctly simulate continuation marks.

6.2 Concept

The definition of $\mathcal{C}$ entails transformation over each syntactic form of λ_{cm}.

With this in mind, consider a conceptual transformation of application, $\mathcal{C}$[(*rator-expr rand-expr*)], as

```
(λ (flag)
  (λ (marks)
    (let ((rator-value ((C[rator-expr] false) marks))
          (rand-value ((C[rand-expr] false) marks))
      (((rator-value rand-value) flag) marks))))
```

ignoring for the moment that **let** is in neither λ_v or λ_{cm}. This definition captures that

1. before evaluation, we expect *flag* to indicate tail position information and *marks* to provide a list of the current continuation marks,
2. we would like to evaluate $\mathcal{C}$[*rator-expr*] and $\mathcal{C}$[*rand-expr*] in the same manner, providing to each its contextual information–specifically that neither is evaluated in tail position of a **wcm** directive and the continuation marks for each are unchanged from the parent context, and
3. following evaluation of operator and operand and application, evaluation of the resultant term is performed with the original contextual information.

Now consider a conceptual transformation of a **wcm** directive, $\mathcal{C}$[(**wcm** *mark-expr body-expr*)], as

```
(λ (flag)
  (λ (marks)
    ((C[body-expr] true) (let ((mark-value ((C[mark-expr] false) marks))
                               (rest-marks (if flag (snd marks) marks)))
                          (cons mark-value rest-marks)))))
```

with similar caveats as the previous case. This definition captures that

1. as in application, we expect *flag* to indicate tail position information and *marks* to provide a list of the current continuation marks,
2. we evaluate *mark-expr* with correct contextual information,
3. we discard the first continuation mark of the parent context if evaluation is occurring in tail position of a **wcm** directive, and
4. we evaluate $\mathcal{C}$[*body-expr*] with the correct tail-position flag and current continuation marks.

Finally, consider the conceptual transformation of a **ccm** directive, $\mathcal{C}$[(**ccm**)], as

```
(λ (flag)
  (λ (marks)
    marks))
```

wherein we reap the fruits of simplicity from our laborious passing: this definition is gratifyingly direct.

The conceptual transformation of variables x and values $(\lambda\ (x)\ e)$ is straightforward.

We now address the absence of **let**, **if**, *cons*, etc. from λ_v.

We can express the **let** construct in λ_v with application. An expression such as

```
(let ((x_1 e_1)
      ...
      (x_n e_n))
  e)
```

Definition 1. $\mathbf{true} = (\lambda\,(x)\,(\lambda\,(y)\,x))$

Definition 2. $\mathbf{false} = (\lambda\,(x)\,(\lambda\,(y)\,y))$

Definition 3. $\mathbf{cons} = (\lambda\,(a)\,(\lambda\,(b)\,(\lambda\,(z)\,((z\,a)\,b))))$

Definition 4. $\mathbf{fst} = (\lambda\,(p)\,(p\,\mathbf{true}))$

Definition 5. $\mathbf{snd} = (\lambda\,(p)\,(p\,\mathbf{false}))$

Definition 6. $\mathbf{nil} = \mathbf{false}$

Figure 6.1: Church encodings for booleans and lists.

can be interpreted as

$$(\ldots((\lambda\ (x\_1)\ \ldots\ (\lambda\ (x\_n)\ e)\ldots)\ e\_1)\ \ldots)\ e\_n)$$

which is the curried form of

$$((\lambda\ (x\_1\ \ldots\ x\_n)\ e)\ e\_1\ \ldots\ e\_n)$$

We unfold this characterization of **let** to guide the construction of $\mathcal{C}$ before simplifying.

To achieve **if** and conditionals as well as list primitives *cons*, *snd*, and *nil*, we use the Church encodings of fig. 6.2.

6.3 Initiation

Abstracting terms has the effect of suspending evaluation. When an entire program is transformed, all evaluation is suspended, and awaits arguments representing contextual information. At the top level, the context is empty, so we pass the contextual information for the empty context: **false**, indicating evaluation is *not* occurring in **wcm** tail position and **nil**, an empty list of marks.

We can accommodate this by defining a top-level transform $\hat{\mathcal{C}}$ in terms of $\mathcal{C}$ by

$$\hat{\mathcal{C}}[p] = ((\mathcal{C}[p]\,\mathbf{false})\,\mathbf{nil}) \tag{6.1}$$

and stating our commutativity property as

$$\hat{\mathcal{C}}[\text{eval}_{cm}(p)] = \text{eval}_v(\hat{\mathcal{C}}[p]) \tag{6.2}$$

which is equivalent to

$$((\mathcal{C}[\text{eval}_{cm}(p)]\,\textbf{false})\,\textbf{nil}) = \text{eval}_v(((\mathcal{C}[p]\,\textbf{false})\,\textbf{nil})) \tag{6.3}$$

6.4 Some Final Subtleties

Our choice to keep the core language small by omitting lists as primitive values has the consequence of complicating our transform somewhat. Because lists are defined in terms of λ-calculus values which are themselves touched by the transform and because of the commutativity property that $\mathcal{C}$ must satisfy, we cannot deal with a list of continuation marks directly–we must instead deal with a transformed list of transformed continuation marks, and manipulation of this list within transformed terms must occur at the transformed level.

Additionally, after evaluation, values are "truncated" with their leading abstractions applied away. For instance, the transformation of the value $(\lambda\ (x)\ x)$ to $(\lambda\ (\mathit{flag})\ (\lambda\ (\mathit{marks})\ (\lambda\ (x)\ (\lambda\ (\mathit{flag})\ (\lambda\ (\mathit{marks})\ x)))))$ will yield, following evaluation, $(\lambda\ (x)\ (\lambda\ (\mathit{flag})\ (\lambda\ (\mathit{marks})\ x)))$. For convenience, we define

$$\mathcal{C}'[(\lambda\,(x)\,e)] = (\lambda\,(x)\,\mathcal{C}[e]) \tag{6.4}$$

and we adjust $\hat{\mathcal{C}}$ so that

$$\hat{\mathcal{C}}[p] = ((\mathcal{C}[p]\,\textbf{false})\,\mathcal{C}'[\textbf{nil}]) \tag{6.5}$$

6.5 Definition of $\mathcal{C}$

Finally, we present the definition of $\mathcal{C}$ over the five syntactic forms of λ_{cm}.

Definition 7. $\mathcal{C}[(\textit{rator-expr}\ \textit{rand-expr})]$

The formal transformation of application follows the **let** version exactly except the definitions of *rator-value* and *rand-value* are folded directly in.

$(\lambda\ (\textit{flags})$
  $(\lambda\ (\textit{marks})$
    $(((((\mathcal{C}[\textit{rator-expr}]\ \textbf{false})\ \textit{marks})$
       $((\mathcal{C}[\textit{rand-expr}]\ \textbf{false})\ \textit{marks}))$
      $\textit{flags})$
     $\textit{marks})))$

Definition 8. $\mathcal{C}[(\textbf{wcm}\ \textit{mark-expr}\ \textit{body-expr})]$

The formal transformation of a **wcm** directive is also extremely similar to the **let** version. The definition of $\mathcal{C}[\textit{cons}]$ is unfolded and simplified.

$(\lambda\ (\textit{flag})$
  $(\lambda\ (\textit{marks})$
    $((\mathcal{C}[\textit{body-expr}]\ \textbf{true})$
     $(((\lambda\ (\textit{mark-value})\ (\lambda\ (\textit{rest-marks})\ \hat{\mathcal{C}}_{\text{cps}}[((\textbf{cons}\ \textit{mark-value})\ \textit{rest-marks})]))$
       $((\mathcal{C}[\textit{mark-expr}]\ \textbf{false})\ \textit{marks}))$
      $((\textit{flag}\ \hat{\mathcal{C}}[(\textbf{snd}\ \textit{marks})])\ \textit{marks})))))$

Definition 9. $\mathcal{C}[(\textbf{ccm})]$

The **let** version of the transformation of a **ccm** directive remains unchanged.

$(\lambda\ (\textit{flag})$
  $(\lambda\ (\textit{marks})$
    $\textit{marks}))$

Definition 10. $\mathcal{C}[v]$=$\mathcal{C}[(\lambda\ (x)\ e)]$

Like other terms, values are modified to receive contextual information. However, being unaffected by context, values discard this information.

$(\lambda\ (\textit{flag})$
  $(\lambda\ (\textit{marks})$
    $(\lambda\ (x)\ \mathcal{C}[e])))$

Definition 11. $\mathcal{C}[x]$

Variables have the property that, when substitution occurs, they reconstitute transformed values. That is, in the midst of application in $\mathcal{C}$, terms of the form $(\mathcal{C}'[(\lambda\ (x)\ x)]\ \mathcal{C}'[(\lambda\ (y)\ y)])$ appear, reducing to $\mathcal{C}[x][x \leftarrow \mathcal{C}'[(\lambda\,(y)\,y)]] = \mathcal{C}[x[x \leftarrow (\lambda\,(y)\,y)] = \mathcal{C}[(\lambda\,(y)\,y)]$.

```
(λ (flag)
  (λ (marks)
    x))
```

6.6 Definition of $\mathcal{C}$ in CPS

We present a continuation mark transformation integrated in CPS.

Definition 12. $\mathcal{C}_{\text{cps}}[(\textit{rator-expr}\ \textit{rand-expr})]$

```
(λ (kont)
  (λ (flag)
    (λ (marks)
      (((Ccps[rator-expr]
        (λ (rator-value)
          (((Ccps[rand-expr]
            (λ (rand-value)
              ((((rator-value rand-value) kont) flag) marks)))
            false) marks)))
        false) marks))))
```

Definition 13. $\mathcal{C}[(\textbf{wcm}\ \textit{mark-expr}\ \textit{body-expr})]$

```
(λ (kont)
  (λ (flag)
    (λ (marks)
      (((Ccps[mark-expr]
        (λ (mark-value)
          ((λ (rest-marks)
            (((Ccps[body-expr] kont) true) Ĉcps[((cons mark-value) rest-marks)]))
           ((flag Ĉcps[(snd marks)]) marks))))
        false) marks))))
```

Definition 14. $\mathcal{C}_{\text{cps}}[(\textbf{ccm})]$

```
(λ (kont)
  (λ (flag)
```

$(\lambda\ (marks)$
$\quad (kont\ marks))))$

Definition 15. $\mathcal{C}_{\text{cps}}[v]$=$\mathcal{C}_{\text{cps}}[(\lambda\ (x)\ e)]$

$(\lambda\ (kont)$
$\quad (\lambda\ (\mathit{flag})$
$\quad\quad (\lambda\ (marks)$
$\quad\quad\quad (kont\ (\lambda\ (x)\ \mathcal{C}_{\text{cps}}[e])))))$

Definition 16. $\mathcal{C}_{\text{cps}}[x]$

$(\lambda\ (kont)$
$\quad (\lambda\ (\mathit{flag})$
$\quad\quad (\lambda\ (marks)$
$\quad\quad\quad (kont\ x))))$

We include corresponding definitions for $\mathcal{C}'$ and $\hat{\mathcal{C}}$.

Definition 17. $\mathcal{C}'_{\text{cps}}[(\lambda\ (x)\ e)]$

$(\lambda\ (x)\ \mathcal{C}_{\text{cps}}[e])$

Definition 18. $\hat{\mathcal{C}}_{\text{cps}}[p]$

$(((\mathcal{C}_{\text{cps}}[p]\ (\lambda\ (x)\ x))\ \mathbf{false})\ \mathcal{C}'_{\text{cps}}[nil])$

6.7 Example

To better illustrate what the transformation does, we step through the reduction of a program which exhibits its more interesting aspects. One λ_{cm} program suited to this purpose is (**wcm** 0 ((λ (x) (**wcm** x (**ccm**))) 1)). It reduces according to λ_{cm} semantics as

(**wcm** 0 ((λ (x) (**wcm** x (**ccm**))) 1))
(**wcm** 0 (**wcm** 1 (**ccm**)))
(**wcm** 1 (**ccm**))
(**wcm** 1 (λ (z) ((z 1) (λ (x) (λ (y) y)))))
(λ (z) ((z 1) (λ (x) (λ (y) y))))

Now consider the reduction of the same program transformed. We apply the transformation just-in-time as we reduce to prevent term size explosion and promote clarity and omit uninteresting reductions.

$\hat{\mathcal{C}}[(\mathbf{wcm}\ \mathsf{0}\ ((\lambda\ (x)\ (\mathbf{wcm}\ x\ (\mathbf{ccm})))\ \mathsf{1}))]$

By definition this is

$((\mathcal{C}[(\mathbf{wcm}\ \mathsf{0}\ ((\lambda\ (x)\ (\mathbf{wcm}\ x\ (\mathbf{ccm})))\ \mathsf{1}))]\ \mathbf{false})\ \mathcal{C}'[\mathbf{nil}])$

which explodes upon expansion to

$$\begin{array}{l}(((\lambda\ (\mathit{flag})\\ \quad (\lambda\ (\mathit{marks})\\ \quad\quad ((\mathcal{C}[((\lambda\ (x)\ (\mathbf{wcm}\ x\ (\mathbf{ccm})))\ \mathsf{1})]\ \mathbf{true})\\ \quad\quad\ (((\lambda\ (\mathit{mark\text{-}value})\ (\lambda\ (\mathit{rest\text{-}marks})\ \mathcal{C}'[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})]))\\ \quad\quad\quad ((\mathcal{C}[\mathsf{0}]\ \mathbf{false})\ \mathit{marks}))\ ((\mathit{flag}\ \hat{\mathcal{C}}[(\mathbf{snd}\ \mathit{marks})])\ \mathit{marks})))))\\ \quad \mathbf{false})\ \mathcal{C}'[\mathbf{nil}])\end{array}$$

After the application of contextual information, we reach

$$\begin{array}{l}((\mathcal{C}[((\lambda\ (x)\ (\mathbf{wcm}\ x\ (\mathbf{ccm})))\ \mathsf{1})]\ \mathbf{true})\\ \ (((\lambda\ (\mathit{mark\text{-}value})\ (\lambda\ (\mathit{rest\text{-}marks})\ \mathcal{C}'[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})]))\\ \quad ((\mathcal{C}[\mathsf{0}]\ \mathbf{false})\ \mathcal{C}'[\mathbf{nil}]))\ ((\mathbf{false}\ \hat{\mathcal{C}}[(\mathbf{snd}\ \mathbf{nil})])\ \mathcal{C}'[\mathbf{nil}])))\end{array}$$

the transformation of the **wcm** body. Terms within are arranged so that correct evaluation occurs within the native call-by-value regime. This evaluates *mark-expr* and and prepends its value to the list of continuation marks before proceeding with evaluation of *body-expr*. This reduction soon yields the following term:

$((\mathcal{C}[((\lambda\ (x)\ (\mathbf{wcm}\ x\ (\mathbf{ccm})))\ \mathsf{1})]\ \mathbf{true})\ \mathcal{C}'[((\mathbf{cons}\ \mathsf{0})\ \mathbf{nil})])$

It is evident that this term will behave exactly as a top-level term except as this contextual information influences it, and this is exactly the property we have strived for. Expansion of this term yields

$$\begin{array}{l}(((\lambda\ (\mathit{flag})\\ \quad (\lambda\ (\mathit{marks})\\ \quad\quad (((((\mathcal{C}[(\lambda\ (x)\ (\mathbf{wcm}\ x\ (\mathbf{ccm})))]\ \mathbf{false})\ \mathit{marks})\\ \quad\quad\quad ((\mathcal{C}[\mathsf{1}]\ \mathbf{false})\ \mathit{marks}))\end{array}$$

flag)
marks))) **true**) $\mathcal{C}'$[((**cons** 0) **nil**)])

the expansion of an application. In this example, both the operator and operand are values, so are essentially unaffected by the application of contextual information; this application has the effect of preparing the terms for application:

((((λ (*x*) $\mathcal{C}$[(**wcm** *x* (**ccm**))])
1) **true**) $\mathcal{C}'$[((**cons** 0) **nil**)])

reduces to

(($\mathcal{C}$[(**wcm** 1 (**ccm**))]
true) $\mathcal{C}'$[((**cons** 0) **nil**)])

This expands and reduces as the **wcm** term seen previously:

(((λ (*flag*)
(λ (*marks*)
(($\mathcal{C}$[(**ccm**)] **true**)
(((λ (*mark-value*) (λ (*rest-marks*) $\mathcal{C}'$[((**cons** *mark-value*) *rest-marks*)]))
(($\mathcal{C}$[1] **false**) *marks*))
((*flag* $\hat{\mathcal{C}}$[(**snd** *marks*)]) *marks*)))))
true) $\mathcal{C}'$[((**cons** 0) **nil**)])

Of interest in this process is the effective collapse of the previous *mark* context by virtue of the value of *flag*. When we reach

((λ (*marks*) *marks*)
((λ (*rest-marks*) $\mathcal{C}'$[((**cons** 1) *rest-marks*)])
((**true** $\hat{\mathcal{C}}$[(**snd** ((**cons** 0) **nil**))]) $\mathcal{C}'$[((**cons** 0) **nil**)])))

the list is beheaded to simulate mark overwriting:

((λ (*marks*) *marks*)
((λ (*rest-marks*) $\mathcal{C}'$[((**cons** 1) *rest-marks*)])
$\hat{\mathcal{C}}$[(**snd** ((**cons** 0) **nil**))]))

Once given the contextual information, the evaluation of **ccm** is simple:

((λ (*marks*) *marks*)
$\mathcal{C}'$[((**cons** 1) **nil**)])

reduces to

$\mathcal{C}'[((\mathbf{cons}\ 1)\ \mathbf{nil})]$

and we are left with just what we hoped for.

Chapter 7

Testing

We do not attempt to construct a correct transformation *ex nihilo*. A pragmatic approach to the discovery of a correct transformation involves consistent feedback and testing to validate candidate transforms. Testing is no substitute for proof, but, as Klein et al. [11] show, proof is no substitute for testing. Lightweight mechanization is a fruitful middle ground between pencil-and-paper analysis and fully- mechanized formal proof. We use Redex to provide feedback, thoroughly exercise candidates, and perform exploratory analysis.

7.1 Redex

Redex [8] is a domain-specific language for exploring language semantics. It lives very close to the semantics notation we have used so far in this discussion.

7.1.1 Toy Language

To illustrate how easily langagues can be defined in Redex, we will examine a Redex program which defines a toy language. In contrast to a Redex tutorial, we will not concern ourselves with the syntax and structure of roads not taken and will instead briefly explain each component of the program.

(**define-language** toy
  (x **variable-not-otherwise-mentioned**)
  (v **number** undefined)
  (e (+ e e) (**with** (x e) e) x v)
  (E • (+ E e) (+ v E) (**with** (x E) e)))

This expression defines the abstract syntactic structure of a language named *toy*. There are four categories of structures: x, v, e, and E. The category x is defined to contain any token not otherwise mentioned in the definition. The category v is defined to contain numbers and the token undefined. The category e is defined to contain the expression forms of the language, of which there are four: addition expressions, **with** expressions, lone variables, and lone values. The last category, E, does not define abstract syntax but instead reduction contexts. The first reduction context is a • (a special token in Redex) which will be filled in with the result of the expression that previously resided in its place. The next two are addition contexts, the first representing the evaluation of the first argument and the second representing the evaluation of the second; the composition of these contexts imposes an order on the evaluation of the arguments. The final context, a **with** context, specifies a variable, a value, and an expression within which that variable is bound to that value.

```
(define toy-rr
  (reduction-relation toy
   (→ (in-hole E (+ number_1 number_2))
      (in-hole E (+ (term number_1) (term number_2)))
      "+")
   (→ (in-hole E (with (x_1 v_1) e_1))
      (in-hole E (substitute x_1 v_1 e_1))
      "with")
   (→ (in-hole E x_1)
      (in-hole E undefined)
      "free variable")
   (→ (in-hole E (+ undefined e_1))
      (in-hole E undefined)
      "undefined in first position")
   (→ (in-hole E (+ number_1 undefined))
      (in-hole E undefined)
      "undefined in second position")))
```

This term defines a reduction relation on the toy language. The five defined reductions, signalled by →, match specified patterns and manipulate them according to the defined rules. These define: the addition of two numbers; the substitution of a **with** expression; a lone

variable; the addition of an undefined value on the left; and the addition of an undefined value on the right.

```
(define-metafunction toy
  substitute : x v e -> e
  [(substitute x_1 v_1 (+ e_1 e_2))
   (+ (substitute x_1 v_1 e_1) (substitute x_1 v_1 e_2))]
  [(substitute x_1 v_1 (with (x_1 e_1) e_2))
   (with (x_1 (substitute x_1 v_1 e_1)) e_2)]
  [(substitute x_1 v_1 (with (x_2 e_1) e_2))
   (with (x_2 (substitute x_1 v_1 e_1)) (substitute x_1 v_1 e_2))]
  [(substitute x_1 v_1 x_1)
   v_1]
  [(substitute x_1 v_1 x_2)
   x_2]
  [(substitute x_1 v_1 v_2)
   v_2])
```

The definition of the **with** reduction rule relies on the **substitute** metafunction. (The language used to define toy (Redex) is the metalanguage. As functions are in the language, metafunctions are in the metalanguage.) The **substitute** metafunction recursively substitutes a variable in an expression with a value. The substitution is only propagated as long as a binding with the same name is not encountered. At that point, the substitution is performed in the value expression of the binding, but not the body. This allows for expressions like

```
(with (x 5)
  (with (x x)
    x))
```

to behave as we expect (returning 5).

Testing

Now that the syntactic forms and reduction rules of the language are defined, we can use the randomized testing built into Redex to investigate properties of the language. We start by defining the helper function

```
(define (reduces-to-one-value? e)
```

```
(let ((results (apply-reduction-relation* toy-rr e)))
  (and (= (length results) 1)
       (value? (first results)))))
```

which has its own helper function

```
(define value? (redex-match toy v))
```

The $*$ at the end of the function name **apply-reduction-relation**$*$ signifies that all possible reduction rules will be applied as many times as possible. If some of the reduction rules don't actually reduce terms, the relation may produce a reducible term indefinitely. The function **apply-reduction-relation**$*$ is in a sense strict in the reduction relation and will likewise run indefinitely if this is the case.

After the language and some properties have been established, the randomized testing, initiated by

```
(redex-check toy e (reduces-to-one-value? (term e)))
```

is simple. We merely provide the name of the language we wish to work with, the nonterminal in the grammar we wish to use to generate language terms, and a predicate that checks terms for properties. This function generates terms gradually increasing in size, applying the predicate to each in turn, and terminates with a counterexample or after a set number of terms have been checked (1000 by default).

Proof

Randomized testing can increase our confidence in various assertions but is no substitute for proof. We express the property of reducing to one value with the following theorem:

Toy Language One-Value Theorem. *For all terms e of the toy language, e reduces to exactly one value.*[1]

[1]We do not use the term *value* loosely here; the toy language definition specifies what constitute values, and we appeal to this.

We proceed by induction on the structure of terms e of the toy language. First, we consider the base cases.

Case v. A term e of the form v is exactly one value and cannot be reduced, so the statement holds. □

Case x. A term e of the form x, a variable, reduces to undefined, a value term, so the statement holds. □

*Case (**with** (x e_1) e_2).* By induction, we assume that *e_1* reduces to exactly one value. Then the "with"-rule can only be applied once, resulting in a single term *e_2* in e which, by our inductive hypothesis, reduces to only one value. □

Case (+ e_1 e_2). By induction, we assume both *e_1* and *e_2* reduce to a single value. We consider two subcases: If *e_1* reduces to undefined, the "undefined in first position"-rule is applied, and the whole term reduces to undefined. If *e_1* reduces to a number, we consider two further subcases: If *e_2* reduces to a number, the "+"-rule is applied, and the entire expression reduces to the sum of the two numbers obtained. If *e_2* reduces to undefined, the "undefined in second position"-rule is applied, and the entire term reduces to undefined. Thus, in all subcases, the whole term reduces to exactly one value. □

This property is fairly trivial and its proof is similarly trivial, but it is a shadow of the approach we will ultimately take to verify certain transformation properties.

7.2 Flavors of λ

Our first task in developing a testing environment for transformations is to define interpreters for λ_{cm} and λ_v.

We begin with λ_v, the simpler language.

7.2.1 λ_v

To begin, we define language terms and evaluation contexts.

```
(define-language λv
  (e (e e) x v error)
  (x variable-not-otherwise-mentioned)
  (v (λ (x) e))
  (E (E e) (v E) •))
```

The characterization of λ_v seen in figures 4.1 and 4.2 was influenced by the knowledge that an interpreter would need to be built. Because Redex lives so close to this characterization, the only change we make in translation is the addition of error.

```
(define λv-rr
  (reduction-relation λv
   (→ (in-hole E ((λ (x) e) v))
      (in-hole E (λv-subst x v e))
      "betav")
   (→ (in-hole E x)
      (in-hole E error)
      "error: unbound identifier")
   (→ (in-hole E (error e))
      (in-hole E error)
      "error in operator")
   (→ (in-hole E (v error))
      (in-hole E error)
      "error in operand")))
```

The first rule in the reduction relation corresponds with the sole semantic rule found in 4.2. The remaining handle cases introduced by error.

```
(define-metafunction λv
  λv-subst : x v e -> e
  ;; 1. substitute in application
  [(λv-subst x_1 v_1 (e_1 e_2))
   ((λv-subst x_1 v_1 e_1) (λv-subst x_1 v_1 e_2))]
  ;; 2a. substitute in variable (same)
  [(λv-subst x_1 v_1 x_1)
   v_1]
  ;; 2b. substitute in variable (different)
  [(λv-subst x_1 v_1 x_2)
```

```
 x_2]
;; 3a. substitute in abstraction (bound)
[(λv-subst x_1 v_1 (λ (x_1) e_1))
 (λ (x_1) e_1)]
;; 3b. substitute in abstraction (free)
[(λv-subst x_1 v_1 (λ (x_2) e_1))
 (λ (x_2) (λv-subst x_1 v_1 e_1))]
;; 4. substitute in error
[(λv-subst x_1 v_1 error)
 error])
```

The λv-subst metafunction presents the conceptual definition of substitution. Because we generate fresh identifiers within the transform, we don't need to worry about capture avoidance.

7.2.2 λ_{cm}

Since λ_{cm} is a superset of λ_v, we need only extend the definition of the λ_v interpreter to accommodate the additions λ_{cm} brings.

```
(define-extended-language λcm λv
  (e .... (wcm e e) (ccm))
  (E (wcm v F) F)
  (F (E e) (v E) (wcm E e) •))
```

Redex allows us to easily define a proper extension of a language, inheriting anything left unspecified. As similar as the λ_v interpeter definition is to the λ_v definition in figure 4.1, this λ_{cm} interpreter definition is to the λ_{cm} definition in figure 4.3.

```
(define λcm-rr
  (extend-reduction-relation λv-rr λcm
   (→ (in-hole E ((λ (x) e) v))
      (in-hole E (λcm-subst x v e))
      "betav")
   (→ (in-hole E (wcm v_1 (wcm v_2 e)))
      (in-hole E (wcm v_2 e))
      "wcm-collapse")
   (→ (in-hole E (wcm v_1 v_2))
      (in-hole E v_2)
      "wcm")
```

```
  (→ (in-hole E (ccm))
     (in-hole E (chi E (λ (x) (λ (y) y))))
     "chi")
  (→ (in-hole E (wcm error e))
     (in-hole E error)
     "error in wcm mark expression")
  (→ (in-hole E (wcm v error))
     (in-hole E error)
     "error in wcm body expression")))
```

The first three rules in the reduction relation correspond with the three additional semantic rules found in 4.4. The remaining handle cases introduced by the the new language forms' interaction with error.

```
(define-metafunction/extension λv-subst λcm
  λcm-subst : x v e -> e
  ;; 1. substitute in wcm form
  [(λcm-subst x_1 v_1 (wcm e_1 e_2))
   (wcm (λcm-subst x_1 v_1 e_1) (λcm-subst x_1 v_1 e_2))]
  ;; 2. substitute in ccm form
  [(λcm-subst x_1 v_1 (ccm))
   (ccm)])
```

The λcm-subst metafunction is extended to accommodate the additional forms in λ_{cm}.

```
(define-metafunction λcm
  chi : E v -> v
  [(chi • v_ms)      v_ms]
  [(chi (E e) v_ms)     (chi E v_ms)]
  [(chi (v E) v_ms)     (chi E v_ms)]
  [(chi (wcm E e) v_ms) (chi E v_ms)]
  [(chi (wcm v E) v_ms) (chi E (λ (p) ((p v) v_ms)))])
```

Finally, we define the χ metafunction. Its definition does not map directly to the formal definition, but matches the intuitive definition that underlies it.

7.3 Transformation definition

7.3.1 Direct Style

```
(define (c e)
```

```
    (let ([flag (gensym 'f)]
          [marks (gensym 'm)]
          [mark-value (gensym 'a)]
          [rest-marks (gensym 'r)])
      (match e
        [(list 'ccm)
         `(λ (,flag)
            (λ (,marks)
              ,marks))]
        [(list 'wcm mark-expr body-expr)
         `(λ (flag)
            (λ (marks)
              ((,(c body-expr) (λ (x) (λ (y) x)))
               (((λ (,mark-value) (λ (,rest-marks) ,(c-hat `(λ (z) ,(c `((z ,mark-value) ,rest-marks))))))
                 ((,(c mark-expr) (λ (x) (λ (y) y))) ,marks))
                ((flag ,(c-hat `((λ (p) (p (λ (x) (λ (y) y)))) ,marks))) ,marks)))))]
        [(list 'λ (list x0) e0)
         `(λ (,flag)
            (λ (,marks)
              (λ (,x0)
                ,(c e0)))))]
        [(list rator-expr rand-expr)
         `(λ (,flag)
            (λ (,marks)
              (((((,(c rator-expr) (λ (x) (λ (y) y))) ,marks)
                 ((,(c rand-expr) (λ (x) (λ (y) y))) ,marks))
                ,flag)
               ,marks))))]
        ['error
         'error]
        [x0
         `(λ (flag) (λ (marks) ,x0))])))

(define (c-hat e)
  (let ([f (gensym 'f)]
        [m (gensym 'm)])
    `((,(c e) (λ (x) (λ (y) y))) (λ (x) ,(c '(λ (y) y))))))
```

7.3.2 Continuation-passing Style

```
(define (c-cps e)
  (let ([kont (gensym 'k)]
        [flag (gensym 'f)]
        [marks (gensym 'm)]
```

```
    [rator-value (gensym 't)]
    [rand-value (gensym 'n)]
    [rest-marks (gensym 'r)]
    [a (gensym 'a)]
    [b (gensym 'b)]
    [f (gensym 'f)])
(match e
  [(list 'ccm)
   `(λ (,kont)
      (λ (,flag)
        (λ (,marks)
          (,kont ,marks))))]
  [(list 'wcm mark-expr body-expr)
   `(λ (,kont)
      (λ (,flag)
        (λ (,marks)
          (((,(c-cps mark-expr)
             (λ (,mark-value)
               ((λ (,rest-marks)
                  (((,(c-cps body-expr)
                     ,kont)
                    (λ (x) (λ (y) x)))
                   ,(c-hat `(λ (z) ((z ,mark-value) ,rest-marks)))))
                ((flag ,(c-hat `((λ (p) (p (λ (x) (λ (y) y)))) ,marks))) ,marks))))
            (λ (x) (λ (y) y)))
           ,marks))))]
  [(list 'λ (list x0) e0)
   `(λ (,kont)
      (λ (,flag)
        (λ (,marks)
          (kont (λ (,x0) ,(c-cps e0)))))))]
  [(list rator-expr rand-expr)
   `(λ (,kont)
      (λ (,flag)
        (λ (,marks)
          (((,(c-cps rator-expr)
             (λ (,rator-value)
               (((,(c-cps rand-expr)
                  (λ (,rand-value)
                    ((((,rator-value ,rand-value) ,kont) ,flag) ,marks)))
                 (λ (x) (λ (y) y)))
                ,marks)))
            (λ (x) (λ (y) y)))
           ,marks))))]
  ['error
```

```
        'error]
       [x0
        `(λ (,kont)
           (λ (,flag)
             (λ (,marks)
               (,kont ,x0))))]))))

(define (c-hat-cps e)
  (let ([k (gensym 'k)]
        [f (gensym 'f)]
        [m (gensym 'm)])
    `(((,(c-cps e) (λ (x) x)) (λ (x) (λ (y) y))) (λ (x) ,(c-cps '(λ (y) y))))))
```

7.4 Transformation testing

We can test that the property described by equation 6.2 holds for a given program p with

```
(define (meaning-preserved? p)
  (alpha-eq? (eval λv (c-hat (eval λcm p))) (eval λv (c-hat p))))
```

where **alpha-eq?** determines α-equivalence between two λ-calculus terms and **eval** is an alias for the Redex native **apply-reduction-relation**$*$.

Redex provides convenient functions to initiate random testing.

```
(redex-check λcm e (meaning-preserved? e))
```

redex-check generates random terms according to the grammar of the given language (λcm) and category (e) in search of counterexamples to the predicate. It gradually increases the size of the terms it generates, which we found useful in obtaining minimal test cases. We subjected both the direct and CPS transformation to random testing and each eventually withstood 10,000 random tests. Interestingly, no incorrect transformation withstood more than 500 random tests before failing.

Chapter 8

Proof

The evaluation of a λ_{cm} program proceeds with the evolution of an evaluation context and possibly reducible expression. When evaluation begins, the evaluation context is merely $\bullet$, a placeholder for the eventual result, and the reducible expression, or redex, is the program itself. The evaluation of arguments–both in application and continuation mark forms–defers evaluation of the expression at hand by storing its evaluation context and evaluating subterms. As these results are applied and evaluation continues, the size of the context fluctuates until finally, if the program terminates, we are left with a single value to plug in $\bullet$. This value is the value of the program.

The state of evaluation at any given point can be encapsulated by a pair of an evaluation context E and an expression e which we write in unorthodox style as $E[e]$. In order to prove that evaluation in the transformation corresponds to native evaluation, we must relate this state with its corresponding transformation.

We do this by overloading $\mathcal{C}_{\text{cps}}$ to accommodate evaluation contexts which allows us to formally relate $E[e]$ and $\mathcal{C}_{\text{cps}}[E[e]]$. We first define

Definition 19.

$$\xi(E) = \begin{cases} \textbf{true} & \text{if } E = E'[(\text{wcm}\, v' \bullet)] \text{ for some } E' \text{ and } v' \\ \textbf{false} & \text{otherwise} \end{cases}$$

to denote the *flags* argument and assume that the *marks* argument is $\mathcal{C}'_{\text{cps}}[\chi(E)]$. We can now define $\mathcal{C}_{\text{cps}}$ over contexts $E \in \lambda_{cm}$:

Definition 20. $\mathcal{C}_{\text{cps}}[\bullet]$

$(\lambda\ (\mathit{value})$
  $\mathit{value})$

Definition 21. $\mathcal{C}_{\text{cps}}[E[(\bullet\ \mathit{rand\text{-}value})]]$

$(\lambda\ (\mathit{rator\text{-}value})$
  $(((\mathcal{C}_{\text{cps}}[\mathit{rand\text{-}expr}]$
    $(\lambda\ (\mathit{rand\text{-}value})$
      $((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathcal{C}_{\text{cps}}[E])\ \xi(E))\ \mathcal{C}'_{\text{cps}}[\chi(E)]))))$
    $\mathbf{false})$
  $\mathcal{C}'_{\text{cps}}[\chi(E)]))$

Definition 22. $\mathcal{C}_{\text{cps}}[E[(v_0\ \bullet)]]$

$(\lambda\ (\mathit{rand\text{-}value})$
  $((((v_0\ \mathit{rand\text{-}value})\ \mathcal{C}_{\text{cps}}[E])\ \xi(E))\ \mathcal{C}'_{\text{cps}}[\chi(E)]))$

Definition 23. $\mathcal{C}_{\text{cps}}[E[(\mathbf{wcm}\ \bullet\ \mathit{body\text{-}expr})]]$

$(\lambda\ (\mathit{mark\text{-}value})$
  $((\lambda\ (\mathit{rest\text{-}marks})$
      $(((\mathcal{C}_{\text{cps}}[\mathit{body\text{-}expr}]\ \mathcal{C}_{\text{cps}}[E])\ \mathbf{true})\ \hat{\mathcal{C}}_{\text{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})]))$
    $((\xi(E)\ \hat{\mathcal{C}}_{\text{cps}}[(\mathbf{snd}\ \chi(E))])\ \mathcal{C}'_{\text{cps}}[\chi(E)])))$

Definition 24. $\mathcal{C}_{\text{cps}}[E[(\mathbf{wcm}\ v_0\ \bullet)]]$

$\mathcal{C}_{\text{cps}}[E]$

This allows us to define $\mathcal{C}_{\text{cps}}$ over a context-expression pair.

Definition 25. $\mathcal{C}_{\text{cps}}[E[e]]$

$(((\mathcal{C}_{\text{cps}}[e]\ \mathcal{C}_{\text{cps}}[E])\ \xi(E))\ \mathcal{C}'_{\text{cps}}[\chi(E)])$

From this definition, it is apparent that $\hat{\mathcal{C}}_{\text{cps}}[p]=(((\mathcal{C}_{\text{cps}}[p]\ \mathcal{C}_{\text{cps}}[\bullet])\ \xi(\bullet))\ \mathcal{C}'_{\text{cps}}[\bullet])=\mathcal{C}_{\text{cps}}[\bullet[p]]$.

Now we show that substitution is preserved by the transformation.

Lemma 1 (Substitution)**.** *For all* $e, x, v \in \lambda_{cm}$, $\mathcal{C}[e[x \leftarrow v]] = \mathcal{C}[e][x \leftarrow \mathcal{C}'[v]]$.

See appendix A for proof.

Finally, we define "filling the hole", the insertion of a value in the context from which it came.

Definition 26. $\mathcal{C}_{\text{cps}}[E[v]]$

$(\mathcal{C}_{\text{cps}}[E]\ \mathcal{C}'_{\text{cps}}[v])$

With each significant step of native evaluation formally related with the transformation, we can express a simulation lemma.

Lemma 2. *For all contexts* $E \in \lambda_{cm}$ *and expressions* $e \in \lambda_{cm}$, $E[e] \rightarrow_{\lambda_{cm}} E'[e'] \implies \mathcal{C}_{\text{cps}}[E[e]] \rightarrow^*_{\lambda_v} \mathcal{C}_{\text{cps}}[E'[e']]$

We will reason by structural induction on both contexts E and terms e. Instead of nesting the induction, which requires the consideration of $|E| \cdot |e|$ cases, we will take first E and then e in isolation, in each assuming the correctness of the other, which requires the consideration of only $|E| + |e|$ cases.

First, we prove it holds for terms e. In each case, let E be an arbitrary context.

Proof. Case $e = (e_0\, e_1)$ By steps app1-app3, $\mathcal{C}_{\text{cps}}[E[(e_0\, e_1)]] \rightarrow^*_{\lambda_v} \mathcal{C}_{\text{cps}}[E[(\bullet\, e_1)][e_0]]$. □

Proof. Case $e = (\text{wcm}\, e_0\, e_1)$ By steps wcm1-wcm3, $\mathcal{C}_{\text{cps}}[E[(\text{wcm}\, e_0\, e_1)]] \rightarrow^*_{\lambda_v} \mathcal{C}_{\text{cps}}[E[(\text{wcm}\ \bullet\ e_1)][e_0]]$. □

Proof. Case $e = (\text{ccm})$ By steps ccm1-ccm3, $\mathcal{C}_{\text{cps}}[E[(\text{ccm})]] \rightarrow^*_{\lambda_v} \mathcal{C}_{\text{cps}}[E[\chi(E)]]$. □

Proof. Case $e = v_0$ By steps value1-value3, $\mathcal{C}_{\text{cps}}[E[v_0]] \rightarrow^*_{\lambda_v} \mathcal{C}_{\text{cps}}[E[v_0]]$. □

Proof. Case $e = x$ By steps x1-x4, $\mathcal{C}_{\text{cps}}[E[x]] \rightarrow^*_{\lambda_v} \mathcal{C}_{\text{cps}}[E[\text{error}]]$. □

Now, we prove it holds for contexts E. In each case, let v_0 be an arbitrary value.

Proof. Case $E = \bullet$ This is identical to the case that $e = v_0$. □

Proof. Case $E = E'[(\bullet\, e_1)]$ By step app4, $\mathcal{C}_{\text{cps}}[E[v_0]] \rightarrow_{\lambda_v} \mathcal{C}_{\text{cps}}[E'[(v_0\, \bullet)][e_1]]$. □

Proof. Case $E = E'[(v_0\ \bullet)]$ By step app5, $\mathcal{C}_{\text{cps}}[E[v_0]] \to_{\lambda_v} \mathcal{C}_{\text{cps}}[E'[(v_0\ v_1)]]$.

By step app6 and lemma 1, $\mathcal{C}_{\text{cps}}[E'[(v_0\ v_1)]] \to_{\lambda_v} \mathcal{C}_{\text{cps}}[E'[e']]$. □

Proof. Case $E = E'[(\text{wcm}\ \bullet\ e_1)]$ If $E' = E''[(\text{wcm}\ v'\ \bullet)$ for some E'' and v', then $\mathcal{C}_{\text{cps}}[E'[(\text{wcm}\ \bullet\ e_1)][v_0]] \to^*_{\lambda_v} \mathcal{C}_{\text{cps}}[E''[(\text{wcm}\ v_0\ \bullet)][e_1]]$ by steps wcm4tail-wcm6tail.

Otherwise, $\mathcal{C}_{\text{cps}}[E'[(\text{wcm}\ \bullet\ e_1)][v_0]] \to^*_{\lambda_v} \mathcal{C}_{\text{cps}}[E'[(\text{wcm}\ v_0\ \bullet)][e_1]]$ by steps wcm4nontail-wcm6nontail. □

Proof. Case $E = E'[(\text{wcm}\ v_0\ \bullet)]$ By definition, $\mathcal{C}_{\text{cps}}[E[v_1]] = \mathcal{C}_{\text{cps}}[E'[v_1]]$. □

Theorem 1 (Simulation)**.** *For all contexts E and terms $e \in \lambda_{cm}$, if $E[e] \to^*_{\lambda_{cm}} \bullet[v]$, then $\mathcal{C}_{\text{cps}}[E[e]] \to^*_{\lambda_v} \mathcal{C}_{\text{cps}}[\bullet[v]]$.*

By induction on the number n of reduction steps by $\to_{\lambda_{cm}}$, we show that the $\mathcal{C}_{cps}$ preserves $\to_{\lambda_{cm}}$.

Proof. Case $n = 0$

This is the case that $E[e] = \bullet[v]$ and holds immediately. □

Proof. Case $n = k$

Suppose that $E[e] \to_{\lambda_{cm}} E_1[e_1] \to_{\lambda_{cm}} \cdots \to_{\lambda_{cm}} E_k[e_k]$ in k steps. Then $\mathcal{C}_{\text{cps}}[E[e]] \to^*_{\lambda_v} \mathcal{C}_{\text{cps}}[E_1[e_1]] \to^*_{\lambda_v} \cdots \to^*_{\lambda_v} \mathcal{C}_{\text{cps}}[E_k[e_k]]$ in some finite number of steps, by the inductive hypothesis. If $E_k[e_k] = \bullet[v]$, it holds. Otherwise, by lemma 2, if $E_k[e_k] \to_{\lambda_{cm}} E_{k+1}[e_{k+1}]$, then $\mathcal{C}_{\text{cps}}[E_k[e_k]] \to^*_{\lambda_v} \mathcal{C}_{\text{cps}}[E_{k+1}[e_{k+1}]]$. Thus, it holds for $n = k + 1$. □

Correctness of $\hat{\mathcal{C}}$. *For all programs $p \in \lambda_{cm}$, $\hat{\mathcal{C}}_{\text{cps}}[\text{eval}_{cm}(p)] \equiv \text{eval}_v(\hat{\mathcal{C}}_{\text{cps}}[p])$.*

Proof.

$$
\begin{aligned}
p \to^*_{\lambda_{cm}} v &\implies \text{eval}_{cm}(p) = v && \text{by definition of eval}_{cm} \\
&\implies \hat{\mathcal{C}}_{\text{cps}}[\text{eval}_{cm}(p)] \to^*_{\lambda_v} \mathcal{C}'_{\text{cps}}[v] && \text{by definition of } \hat{\mathcal{C}}_{cps} \\
&\implies \hat{\mathcal{C}}_{\text{cps}}[\text{eval}_{cm}(p)] \equiv \mathcal{C}'_{\text{cps}}[v]
\end{aligned}
$$

$$
\begin{aligned}
p \to^*_{\lambda_{cm}} v &\implies \bullet[p] \to^*_{\lambda_{cm}} \bullet[v] \\
&\implies \mathcal{C}_{\text{cps}}[\bullet[p]] \to^*_{\lambda_v} \mathcal{C}_{\text{cps}}[\bullet[v]] && \text{(by theorem 1)} \\
&\implies \mathcal{C}_{\text{cps}}[\bullet[p]] \to^*_{\lambda_v} \mathcal{C}'_{\text{cps}}[v] && \text{(by definition of } \mathcal{C}_{cps}) \\
&\implies (((\mathcal{C}_{\text{cps}}[p]\, \mathcal{C}_{\text{cps}}[\bullet])\, \xi(\bullet))\, \mathcal{C}'_{\text{cps}}[\chi(\bullet)]) \to^*_{\lambda_v} \mathcal{C}'_{\text{cps}}[v] && \text{(by definition of } \mathcal{C}_{cps}) \\
&\implies \hat{\mathcal{C}}_{\text{cps}}[p] \to^*_{\lambda_v} \mathcal{C}'_{\text{cps}}[v] && \text{(by definition of } \hat{\mathcal{C}}_{cps}) \\
&\implies \text{eval}_v(\hat{\mathcal{C}}_{\text{cps}}[p]) = \mathcal{C}'_{\text{cps}}[v] && \text{(by definition of eval}_v)
\end{aligned}
$$

Therefore, $\hat{\mathcal{C}}_{\text{cps}}[\text{eval}_{cm}(p)] \equiv \text{eval}_v(\hat{\mathcal{C}}_{\text{cps}}[p])$. □

Chapter 9

Conclusion

Continuation marks support a bevy of instrumentation tools and advanced language features in a generalized, portable way. Despite their demonstrated utility, they have not yet found their way into most languages. A verified characterization of continuation marks in a pure computational language provides implementors of higher-order languages a correct compiler for continuation marks which we have demonstrated for JavaScript.

Our macro-style approach would be more useful if deployed as a proper macro, but this requires a hygienic macro system which many languages lack. `sweet.js` is just such a system for JavaScript in early stages and is ideal for our transformation.

Although we used some level of mechanization in the proof of the theorem, a formal proof assistant, such as Coq [3], would increase confidence out of the gate that the transformation was correct.

References

[1] A.W. Appel. *Compiling with Continuations*. Cambridge University Press, 2007.

[2] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.

[3] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. *The Coq Proof Assistant Reference Manual: Version 6.1*. 1997.

[4] J. Clements. *Portable and High-Level Access to the Stack with Continuation Marks*. PhD thesis, Northeastern University Boston, 2006.

[5] J. Clements and M. Felleisen. A tail-recursive machine with stack inspection. *Transactions on Programming Languages and Systems*, 26(6):1029–1052, 2004.

[6] J. Clements, M. Flatt, and M. Felleisen. Modeling an algebraic stepper. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 320–334. Springer, 2001.

[7] J. Clements, A. Sundaram, and D. Herman. Implementing continuation marks in javascript. In *Proceedings of the Workshop on Scheme and Functional Programming*, pages 45–55. ACM, 2008.

[8] R.B. Findler and C. Klein. *Redex: Practical Semantics Engineering*. MIT Press, 2010.

[9] M.J. Fischer. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions About Programs*, volume 6, pages 104–109. ACM, 1972.

[10] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[11] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J.A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R.B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th Annual Symposium on Principles of Programming Languages*, pages 285–296. ACM, 2012.

[12] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE, 1989.

[13] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen. Continuations from generalized stack inspection. In *ACM SIGPLAN Notices*, volume 40, pages 216–227. ACM, 2005.

[14] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[15] A.A. Sabry. *The Formal Relationship Between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms*. PhD thesis, Rice University, 1994.

[16] D.B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 158–167. ACM, 2003.

[17] D.S. Wallach, A.W. Appel, and E.W. Felten. Safkasi: a security mechanism for language-based systems. *Transactions on Software Engineering and Methodology*, 9(4):341–378, 2000.

Appendix A

Proof of Lemma 1

A.1 Application Form

Proof. Case $e = (\textit{rator-expr}\ \textit{rand-expr})$

First, we have on the left side

$$\mathcal{C}_{\text{cps}}[(\textit{rator-expr}\ \textit{rand-expr})[x \leftarrow v]] = \mathcal{C}_{\text{cps}}[(\textit{rator-expr}[x \leftarrow v]\ \textit{rand-expr}[x \leftarrow v])]$$

We have on the right side

$$\mathcal{C}_{\text{cps}}[(\textit{rator-expr}\ \textit{rand-expr})][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$$

which, expanded, follows this sequence of equivalences:

$$\begin{array}{l}
(\lambda\ (\textit{kont}) \\
\quad (\lambda\ (\textit{flag}) \\
\quad\quad (\lambda\ (\textit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{rator-expr}] \\
\quad\quad\quad\quad (\lambda\ (\textit{rator-value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{rand-expr}] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\textit{rand-value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\textit{rator-value}\ \textit{rand-value})\ \textit{kont})\ \textit{flag})\ \textit{marks}))) \\
\quad\quad\quad\quad\quad\quad \textbf{false})\ \textit{marks}))) \\
\quad\quad\quad\quad \textbf{false})\ \textit{marks}))))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]
\end{array}$$

$$\begin{array}{l}
(\lambda\ (\textit{kont}) \\
\quad (\lambda\ (\textit{flag}) \\
\quad\quad (\lambda\ (\textit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{rator-expr}] \\
\quad\quad\quad\quad (\lambda\ (\textit{rator-value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{rand-expr}] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\textit{rand-value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\textit{rator-value}\ \textit{rand-value})\ \textit{kont})\ \textit{flag})\ \textit{marks}))) \\
\quad\quad\quad\quad\quad\quad \textbf{false})\ \textit{marks}))) \\
\quad\quad\quad\quad \textbf{false})\ \textit{marks}))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]))
\end{array}$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})\ \mathit{flag})\ \mathit{marks}))) \\
\quad\quad\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})\ \mathit{flag})\ \mathit{marks}))) \\
\quad\quad\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))) \\
\quad\quad\quad\quad \mathbf{false})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]\ \mathit{marks}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})\ \mathit{flag})\ \mathit{marks}))) \\
\quad\quad\quad\quad\quad\quad \mathbf{false})\ \mathit{marks})))[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad \mathbf{false}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})\ \mathit{flag})\ \mathit{marks}))) \\
\quad\quad\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})\ \mathit{flag})\ \mathit{marks}))) \\
\quad\quad\quad\quad\quad\quad \mathbf{false})\ \mathit{marks})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})\ \mathit{flag})\ \mathit{marks}))) \\
\quad\quad\quad\quad\quad\quad \mathbf{false})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]\ \mathit{marks}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]))) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})\ \mathit{flag})\ \mathit{marks})))[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad\quad\quad \mathbf{false}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])\ \mathit{marks}))) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})\ \mathit{flag})\ \mathit{marks}))[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]) \\
\quad\quad\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})\ \mathit{flag})\ \mathit{marks})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])) \\
\quad\quad\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})\ \mathit{flag})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]\ \mathit{marks}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]))) \\
\quad\quad\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]\ \mathit{flag}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])\ \mathit{marks}))) \\
\quad\quad\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]\ \mathit{kont}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])\ \mathit{flag})\ \mathit{marks}))) \\
\quad\quad\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]\ \mathit{rand\text{-}value}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])\ \mathit{kont})\ \mathit{flag})\ \mathit{marks}))) \\
\quad\quad\quad\quad\quad\quad \textbf{false})\ \mathit{marks}))) \\
\quad\quad\quad\quad \textbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})\ \mathit{flag})\ \mathit{marks}))) \\
\quad\quad\quad\quad\quad\quad \textbf{false})\ \mathit{marks}))) \\
\quad\quad\quad\quad \textbf{false})\ \mathit{marks}))))
\end{array}
$$

And finally, by induction, we have

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rator\text{-}expr}[x \leftarrow v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{rator\text{-}value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{rand\text{-}expr}[x \leftarrow v]] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\mathit{rand\text{-}value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\mathit{rator\text{-}value}\ \mathit{rand\text{-}value})\ \mathit{kont})\ \mathit{flag})\ \mathit{marks}))) \\
\quad\quad\quad\quad\quad\quad \textbf{false})\ \mathit{marks}))) \\
\quad\quad\quad\quad \textbf{false})\ \mathit{marks}))))
\end{array}
$$

which is equal to

$$\mathcal{C}_{\mathrm{cps}}[(\mathit{rator\text{-}expr}[x \leftarrow v]\ \mathit{rand\text{-}expr}[x \leftarrow v])]$$

Therefore,

$$\mathcal{C}_{\mathrm{cps}}[(\mathit{rator\text{-}expr}\ \mathit{rand\text{-}expr})[x \leftarrow v]] = \mathcal{C}_{\mathrm{cps}}[(\mathit{rator\text{-}expr}\ \mathit{rand\text{-}expr})][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]$$

□

A.2 wcm Form

Proof. Case $e = (\textbf{wcm}\ \textit{mark-expr}\ \textit{body-expr})$

First, we have on the left side

$$\mathcal{C}_{\text{cps}}[(\textbf{wcm}\ \textit{mark-expr}\ \textit{body-expr})[x \leftarrow v]] = \mathcal{C}_{\text{cps}}[(\textbf{wcm}\ \textit{mark-expr}[x \leftarrow v]\ \textit{body-expr}[x \leftarrow v])]$$

We have on the right side

$$\mathcal{C}_{\text{cps}}[(\textbf{wcm}\ \textit{mark-expr}\ \textit{body-expr})][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$$

which, expanded, follows this sequence of equivalences:

$$\begin{array}{l}
(\lambda\ (\textit{kont}) \\
\quad (\lambda\ (\textit{flag}) \\
\quad\quad (\lambda\ (\textit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{mark-expr}] \\
\quad\quad\quad\quad (\lambda\ (\textit{mark-value}) \\
\quad\quad\quad\quad\quad ((\lambda\ (\textit{rest-marks}) \\
\quad\quad\quad\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{body-expr}]\ \textit{kont})\ \textbf{true})\ \hat{\mathcal{C}}_{\text{cps}}[((\textbf{cons}\ \textit{mark-value})\ \textit{rest-marks})])) \\
\quad\quad\quad\quad\quad\quad ((\textit{flag}\ \hat{\mathcal{C}}_{\text{cps}}[(\textbf{snd}\ \textit{marks})])\ \textit{marks})))) \\
\quad\quad\quad\quad\quad \textbf{false})\ \textit{marks}))))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]
\end{array}$$

$$\begin{array}{l}
(\lambda\ (\textit{kont}) \\
\quad (\lambda\ (\textit{flag}) \\
\quad\quad (\lambda\ (\textit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{mark-expr}] \\
\quad\quad\quad\quad (\lambda\ (\textit{mark-value}) \\
\quad\quad\quad\quad\quad ((\lambda\ (\textit{rest-marks}) \\
\quad\quad\quad\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{body-expr}]\ \textit{kont})\ \textbf{true})\ \hat{\mathcal{C}}_{\text{cps}}[((\textbf{cons}\ \textit{mark-value})\ \textit{rest-marks})])) \\
\quad\quad\quad\quad\quad\quad ((\textit{flag}\ \hat{\mathcal{C}}_{\text{cps}}[(\textbf{snd}\ \textit{marks})])\ \textit{marks})))) \\
\quad\quad\quad\quad\quad \textbf{false})\ \textit{marks})))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])
\end{array}$$

$$\begin{array}{l}
(\lambda\ (\textit{kont}) \\
\quad (\lambda\ (\textit{flag}) \\
\quad\quad (\lambda\ (\textit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{mark-expr}] \\
\quad\quad\quad\quad (\lambda\ (\textit{mark-value}) \\
\quad\quad\quad\quad\quad ((\lambda\ (\textit{rest-marks}) \\
\quad\quad\quad\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{body-expr}]\ \textit{kont})\ \textbf{true})\ \hat{\mathcal{C}}_{\text{cps}}[((\textbf{cons}\ \textit{mark-value})\ \textit{rest-marks})])) \\
\quad\quad\quad\quad\quad\quad ((\textit{flag}\ \hat{\mathcal{C}}_{\text{cps}}[(\textbf{snd}\ \textit{marks})])\ \textit{marks})))) \\
\quad\quad\quad\quad\quad \textbf{false})\ \textit{marks}))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]))
\end{array}$$

$$\begin{array}{l}
(\lambda\ (\textit{kont}) \\
\quad (\lambda\ (\textit{flag})
\end{array}$$

$$
\begin{array}{l}
\quad (\lambda\ (\mathit{marks}) \\
\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{mark\text{-}expr}] \\
\quad\quad\quad\quad (\lambda\ (\mathit{mark\text{-}value}) \\
\quad\quad\quad\quad\quad ((\lambda\ (\mathit{rest\text{-}marks}) \\
\quad\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathit{kont})\ \mathbf{true})\ \hat{\mathcal{C}}_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})])) \\
\quad\quad\quad\quad\quad\quad ((\mathit{flag}\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \mathit{marks})])\ \mathit{marks})))) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{mark\text{-}expr}] \\
\quad\quad\quad\quad\quad (\lambda\ (\mathit{mark\text{-}value}) \\
\quad\quad\quad\quad\quad\quad ((\lambda\ (\mathit{rest\text{-}marks}) \\
\quad\quad\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathit{kont})\ \mathbf{true})\ \hat{\mathcal{C}}_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})])) \\
\quad\quad\quad\quad\quad\quad\quad ((\mathit{flag}\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \mathit{marks})])\ \mathit{marks})))) \\
\quad\quad\quad\quad\quad \mathbf{false})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]\ \mathit{marks}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{mark\text{-}expr}] \\
\quad\quad\quad\quad\quad (\lambda\ (\mathit{mark\text{-}value}) \\
\quad\quad\quad\quad\quad\quad ((\lambda\ (\mathit{rest\text{-}marks}) \\
\quad\quad\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathit{kont})\ \mathbf{true})\ \hat{\mathcal{C}}_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})])) \\
\quad\quad\quad\quad\quad\quad\quad ((\mathit{flag}\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \mathit{marks})])\ \mathit{marks})))))[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad\quad \mathbf{false}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{mark\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad\quad (\lambda\ (\mathit{mark\text{-}value}) \\
\quad\quad\quad\quad\quad\quad ((\lambda\ (\mathit{rest\text{-}marks}) \\
\quad\quad\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathit{kont})\ \mathbf{true})\ \hat{\mathcal{C}}_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})])) \\
\quad\quad\quad\quad\quad\quad\quad ((\mathit{flag}\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \mathit{marks})])\ \mathit{marks})))[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]) \\
\quad\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{mark\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad\quad (\lambda\ (\mathit{mark\text{-}value})
\end{array}
$$

$$
\begin{array}{l}
\quad\quad\quad\quad ((\lambda\ (\mathit{rest\text{-}marks}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathit{kont})\ \mathbf{true})\ \hat{\mathcal{C}}_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})])) \\
\quad\quad\quad\quad\quad ((\mathit{flag}\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \mathit{marks})])\ \mathit{marks}))[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{mark\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{mark\text{-}value}) \\
\quad\quad\quad\quad\quad ((\lambda\ (\mathit{rest\text{-}marks}) \\
\quad\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathit{kont})\ \mathbf{true})\ \hat{\mathcal{C}}_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})]))[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad\quad\quad ((\mathit{flag}\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \mathit{marks})])\ \mathit{marks})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]))) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{mark\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{mark\text{-}value}) \\
\quad\quad\quad\quad\quad ((\lambda\ (\mathit{rest\text{-}marks}) \\
\quad\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathit{kont})\ \mathbf{true})\ \hat{\mathcal{C}}_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})])[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]) \\
\quad\quad\quad\quad\quad\quad ((\mathit{flag}\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \mathit{marks})])[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]\ \mathit{marks}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]))))) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{mark\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{mark\text{-}value}) \\
\quad\quad\quad\quad\quad ((\lambda\ (\mathit{rest\text{-}marks}) \\
\quad\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathit{kont})\ \mathbf{true})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]\ \hat{\mathcal{C}}_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[\\
\quad\quad\quad\quad\quad\quad ((\mathit{flag}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \mathit{marks})][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])\ \mathit{marks})))) \\
\quad\quad\quad\quad \mathbf{false})\ \mathit{marks}))))
\end{array}
$$

$$
\begin{array}{l}
(\lambda\ (\mathit{kont}) \\
\quad (\lambda\ (\mathit{flag}) \\
\quad\quad (\lambda\ (\mathit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{mark\text{-}expr}][x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]] \\
\quad\quad\quad\quad (\lambda\ (\mathit{mark\text{-}value}) \\
\quad\quad\quad\quad\quad ((\lambda\ (\mathit{rest\text{-}marks}) \\
\quad\quad\quad\quad\quad\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathit{kont})[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]]\ \mathbf{true}[x \leftarrow \mathcal{C}'_{\mathrm{cps}}[v]])\ \hat{\mathcal{C}}_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}mark} \\
\quad\quad\quad\quad\quad\quad ((\mathit{flag}\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \mathit{marks})])\ \mathit{marks})))) \\
\end{array}
$$

false) *marks*))))

$(\lambda\ (kont)$
  $(\lambda\ (flag)$
    $(\lambda\ (marks)$
      $(((\mathcal{C}_{\text{cps}}[mark\text{-}expr][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$
        $(\lambda\ (mark\text{-}value)$
          $((\lambda\ (rest\text{-}marks)$
            $(((\mathcal{C}_{\text{cps}}[body\text{-}expr][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]\ kont[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])$ **true**$)\ \hat{\mathcal{C}}_{\text{cps}}[((\textbf{cons}\ mark\text{-}value)\ rest\text{-}mark$
           $((flag\ \hat{\mathcal{C}}_{\text{cps}}[(\textbf{snd}\ marks)])\ marks))))$
      false$)\ marks))))$

$(\lambda\ (kont)$
  $(\lambda\ (flag)$
    $(\lambda\ (marks)$
      $(((\mathcal{C}_{\text{cps}}[mark\text{-}expr][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$
        $(\lambda\ (mark\text{-}value)$
          $((\lambda\ (rest\text{-}marks)$
            $(((\mathcal{C}_{\text{cps}}[body\text{-}expr][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]\ kont)$ **true**$)\ \hat{\mathcal{C}}_{\text{cps}}[((\textbf{cons}\ mark\text{-}value)\ rest\text{-}marks)]))$
           $((flag\ \hat{\mathcal{C}}_{\text{cps}}[(\textbf{snd}\ marks)])\ marks))))$
      false$)\ marks))))$

And finally, by induction, we have

$(\lambda\ (kont)$
  $(\lambda\ (flag)$
    $(\lambda\ (marks)$
      $(((\mathcal{C}_{\text{cps}}[mark\text{-}expr[x \leftarrow v]]$
        $(\lambda\ (mark\text{-}value)$
          $((\lambda\ (rest\text{-}marks)$
            $(((\mathcal{C}_{\text{cps}}[body\text{-}expr[x \leftarrow v]]\ kont)$ **true**$)\ \hat{\mathcal{C}}_{\text{cps}}[((\textbf{cons}\ mark\text{-}value)\ rest\text{-}marks)]))$
           $((flag\ \hat{\mathcal{C}}_{\text{cps}}[(\textbf{snd}\ marks)])\ marks))))$
      false$)\ marks))))$

which is equal to

$\mathcal{C}_{\text{cps}}[(\textbf{wcm}\ mark\text{-}expr[x \leftarrow v]\ body\text{-}expr[x \leftarrow v])]$

Therefore,

$\mathcal{C}_{\text{cps}}[(\textbf{wcm}\ mark\text{-}expr\ body\text{-}expr)[x \leftarrow v]] = \mathcal{C}_{\text{cps}}[(\textbf{wcm}\ mark\text{-}expr\ body\text{-}expr)][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$

□

A.3 ccm Form

Proof. Case e=(**ccm**)

First, we have on the left side

$\mathcal{C}_{\text{cps}}[(\textbf{ccm})[x \leftarrow v]] = \mathcal{C}_{\text{cps}}[(\textbf{ccm})]$

We have on the right side

$\mathcal{C}_{\text{cps}}[(\textbf{ccm})][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$

which, expanded, follows this sequence of equivalences:

$(\lambda\ (kont)$
$\quad(\lambda\ (\mathit{flag})$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ marks))))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$

$(\lambda\ (kont)$
$\quad(\lambda\ (\mathit{flag})$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ marks)))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])$

$(\lambda\ (kont)$
$\quad(\lambda\ (\mathit{flag})$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ marks))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]))$

$(\lambda\ (kont)$
$\quad(\lambda\ (\mathit{flag})$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ marks)[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])))$

$(\lambda\ (kont)$
$\quad(\lambda\ (\mathit{flag})$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]\ marks[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]))))$

$(\lambda\ (kont)$
$\quad(\lambda\ (\mathit{flag})$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ marks))))$

which is equal to

$\mathcal{C}_{\text{cps}}[(\textbf{ccm})]$

Therefore, $\mathcal{C}_{\text{cps}}[(\mathbf{ccm})[x \leftarrow v]] = Ccps[(\mathbf{ccm})][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$. □

A.4 Value Form

Proof. Case $e = (\lambda\ (x)\ e')$

On the left side, we have

$$\mathcal{C}_{\text{cps}}[(\lambda\ (x)\ e')[x \leftarrow v]] = \mathcal{C}_{\text{cps}}[(\lambda\ (x)\ e')]$$

On the right side, we have

$$\mathcal{C}_{\text{cps}}[(\lambda\ (x)\ e')][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$$

which follows

$$\begin{array}{l}(\lambda\ (\mathit{kont}) \\ \quad (\lambda\ (\mathit{flag}) \\ \quad\quad (\lambda\ (\mathit{marks}) \\ \quad\quad\quad (\mathit{kont}\ (\lambda\ (x)\ \mathcal{C}_{\text{cps}}[e'])))))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]\end{array}$$

$$\begin{array}{l}(\lambda\ (\mathit{kont}) \\ \quad (\lambda\ (\mathit{flag}) \\ \quad\quad (\lambda\ (\mathit{marks}) \\ \quad\quad\quad (\mathit{kont}\ (\lambda\ (x)\ \mathcal{C}_{\text{cps}}[e'])))))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])\end{array}$$

$$\begin{array}{l}(\lambda\ (\mathit{kont}) \\ \quad (\lambda\ (\mathit{flag}) \\ \quad\quad (\lambda\ (\mathit{marks}) \\ \quad\quad\quad (\mathit{kont}\ (\lambda\ (x)\ \mathcal{C}_{\text{cps}}[e'])))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]))\end{array}$$

$$\begin{array}{l}(\lambda\ (\mathit{kont}) \\ \quad (\lambda\ (\mathit{flag}) \\ \quad\quad (\lambda\ (\mathit{marks}) \\ \quad\quad\quad (\mathit{kont}\ (\lambda\ (x)\ \mathcal{C}_{\text{cps}}[e']))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])))\end{array}$$

$$\begin{array}{l}(\lambda\ (\mathit{kont}) \\ \quad (\lambda\ (\mathit{flag}) \\ \quad\quad (\lambda\ (\mathit{marks}) \\ \quad\quad\quad (\mathit{kont}[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]\ (\lambda\ (x)\ \mathcal{C}_{\text{cps}}[e'])[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])))))\end{array}$$

$$\begin{array}{l}(\lambda\ (\mathit{kont}) \\ \quad (\lambda\ (\mathit{flag}) \\ \quad\quad (\lambda\ (\mathit{marks}) \\ \quad\quad\quad (\mathit{kont}\ (\lambda\ (x)\ \mathcal{C}_{\text{cps}}[e'])))))\end{array}$$

$\mathcal{C}_{\text{cps}}[(\lambda\ (x)\ e')]$

Therefore, $\mathcal{C}_{\text{cps}}[(\lambda\ (x)\ e')[x \leftarrow v]] = \mathcal{C}_{\text{cps}}[(\lambda\ (x)\ e')][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$. □

Proof. Case $e = (\lambda\ (x')\ e')$ where $x' \neq x$

On the left side, we have

$\mathcal{C}_{\text{cps}}[(\lambda\ (x')\ e')[x \leftarrow v]] = \mathcal{C}_{\text{cps}}[(\lambda\ (x')\ e')]$

On the right side, we have

$\mathcal{C}_{\text{cps}}[(\lambda\ (x')\ e')][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$

which follows

$(\lambda\ (kont)$
$\quad(\lambda\ (flag)$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ (\lambda\ (x')\ \mathcal{C}_{\text{cps}}[e'])))))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$

$(\lambda\ (kont)$
$\quad(\lambda\ (flag)$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ (\lambda\ (x')\ \mathcal{C}_{\text{cps}}[e'])))))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])$

$(\lambda\ (kont)$
$\quad(\lambda\ (flag)$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ (\lambda\ (x')\ \mathcal{C}_{\text{cps}}[e'])))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]))$

$(\lambda\ (kont)$
$\quad(\lambda\ (flag)$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ (\lambda\ (x')\ \mathcal{C}_{\text{cps}}[e']))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])))$

$(\lambda\ (kont)$
$\quad(\lambda\ (flag)$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]\ (\lambda\ (x')\ \mathcal{C}_{\text{cps}}[e'])[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])))))$

$(\lambda\ (kont)$
$\quad(\lambda\ (flag)$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ (\lambda\ (x')\ \mathcal{C}_{\text{cps}}[e'][x \leftarrow \mathcal{C}'_{\text{cps}}[v]])))))$

By induction, this is

$(\lambda\ (kont)$
$\quad(\lambda\ (flag)$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ (\lambda\ (x')\ \mathcal{C}_{\text{cps}}[e'[x \leftarrow v]])))))$

which equals

$\mathcal{C}_{\text{cps}}[(\lambda\ (x')\ e'[x \leftarrow v])]$

Therefore, $\mathcal{C}_{\text{cps}}[(\lambda\ (x')\ e')[x \leftarrow v]] = \mathcal{C}_{\text{cps}}[(\lambda\ (x')\ e')][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$. □

A.5 Variable Form

Proof. Case $e = x$

On the left, we have $\mathcal{C}_{\text{cps}}[x[x \leftarrow v]] = \mathcal{C}_{\text{cps}}[v]$.

On the right, we have $\mathcal{C}_{\text{cps}}[x][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$ which follows

$(\lambda\ (kont)$
$\quad(\lambda\ (flag)$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ x))))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$

$(\lambda\ (kont)$
$\quad(\lambda\ (flag)$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ x)))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])$

$(\lambda\ (kont)$
$\quad(\lambda\ (flag)$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ x))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]))$

$(\lambda\ (kont)$
$\quad(\lambda\ (flag)$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont\ x)[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])))$

$(\lambda\ (kont)$
$\quad(\lambda\ (flag)$
$\quad\quad(\lambda\ (marks)$
$\quad\quad\quad(kont[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]\ x[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])))))$

$(\lambda\ (kont)$
  $(\lambda\ (\mathit{flag})$
    $(\lambda\ (marks)$
      $(kont\ \mathcal{C}'_{\text{cps}}[v])))))$

$\mathcal{C}_{\text{cps}}[v]$

Therefore, $\mathcal{C}_{\text{cps}}[x[x \leftarrow v]] = \mathcal{C}_{\text{cps}}[x][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$. □

Proof. Case $e = x'$ where $x' \neq x$

On the left, we have $\mathcal{C}_{\text{cps}}[x'[x \leftarrow v]]$=$\mathcal{C}_{\text{cps}}[x']$.

On the right, we have $\mathcal{C}_{\text{cps}}[x'][x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$, which follows

$(\lambda\ (kont)$
  $(\lambda\ (\mathit{flag})$
    $(\lambda\ (marks)$
      $(kont\ x'))))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]$

$(\lambda\ (kont)$
  $(\lambda\ (\mathit{flag})$
    $(\lambda\ (marks)$
      $(kont\ x')))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])$

$(\lambda\ (kont)$
  $(\lambda\ (\mathit{flag})$
    $(\lambda\ (marks)$
      $(kont\ x'))[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]))$

$(\lambda\ (kont)$
  $(\lambda\ (\mathit{flag})$
    $(\lambda\ (marks)$
      $(kont\ x')[x \leftarrow \mathcal{C}'_{\text{cps}}[v]])))$

$(\lambda\ (kont)$
  $(\lambda\ (\mathit{flag})$
    $(\lambda\ (marks)$
      $(kont[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]\ x'[x \leftarrow \mathcal{C}'_{\text{cps}}[v]]))))$

$(\lambda\ (kont)$
  $(\lambda\ (\mathit{flag})$
    $(\lambda\ (marks)$
      $(kont\ x'))))$

$\mathcal{C}_{\text{cps}}[x']$

Therefore, $\mathcal{C}_{\text{cps}}[x'[x \leftarrow v]] = \mathcal{C}_{\text{cps}}[x'][x \leftarrow \mathcal{C}'_{\text{cps}}[v]$. $\square$

Appendix B

Reductions

B.1 Application Form

$\mathcal{C}_{\text{cps}}[E[(\textit{rator-expr}\ \textit{rand-expr})]]$

$$
\begin{array}{l}
((((\lambda\ (\textit{kont}) \\
\quad (\lambda\ (\textit{flag}) \\
\quad\quad (\lambda\ (\textit{marks}) \\
\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{rator-expr}] \\
\quad\quad\quad\quad\quad (\lambda\ (\textit{rator-value}) \\
\quad\quad\quad\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{rand-expr}] \\
\quad\quad\quad\quad\quad\quad\quad\quad (\lambda\ (\textit{rand-value}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad ((((\textit{rator-value}\ \textit{rand-value})\ \textit{kont})\ \textit{flag})\ \textit{marks}))) \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{false})\ \textit{marks}))) \\
\quad\quad\quad\quad\quad \textbf{false})\ \textit{marks})))) \\
\quad\quad \mathcal{C}[E])\ \xi(E))\ \mathcal{C}'_{\text{cps}}[\chi(E)])
\end{array}
$$

app1

$$
\begin{array}{l}
(((\lambda\ (\textit{flag}) \\
\quad (\lambda\ (\textit{marks}) \\
\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{rator-expr}] \\
\quad\quad\quad\quad (\lambda\ (\textit{rator-value}) \\
\quad\quad\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{rand-expr}] \\
\quad\quad\quad\quad\quad\quad\quad (\lambda\ (\textit{rand-value}) \\
\quad\quad\quad\quad\quad\quad\quad\quad ((((\textit{rator-value}\ \textit{rand-value})\ \mathcal{C}[E])\ \textit{flag})\ \textit{marks}))) \\
\quad\quad\quad\quad\quad\quad \textbf{false})\ \textit{marks}))) \\
\quad\quad\quad\quad \textbf{false})\ \textit{marks}))) \\
\quad\quad \xi(E))\ \mathcal{C}'_{\text{cps}}[\chi(E)])
\end{array}
$$

app2

$$
\begin{array}{l}
((\lambda\ (\textit{marks}) \\
\quad (((\mathcal{C}_{\text{cps}}[\textit{rator-expr}] \\
\quad\quad\quad (\lambda\ (\textit{rator-value}) \\
\quad\quad\quad\quad (((\mathcal{C}_{\text{cps}}[\textit{rand-expr}] \\
\quad\quad\quad\quad\quad\quad (\lambda\ (\textit{rand-value}) \\
\quad\quad\quad\quad\quad\quad\quad ((((\textit{rator-value}\ \textit{rand-value})\ \mathcal{C}[E])\ \xi(E))\ \textit{marks})))
\end{array}
$$

false) *marks*)))
false) *marks*)) $\mathcal{C}'_{\text{cps}}[\chi(E)]$)

app3

(((($\mathcal{C}_{\text{cps}}$[*rator-expr*]
(λ (*rator-value*)
(((($\mathcal{C}_{\text{cps}}$[*rand-expr*]
(λ (*rand-value*)
((((*rator-value rand-value*) $\mathcal{C}[E]$) $\xi(E)$) $\mathcal{C}'_{\text{cps}}[\chi(E)]$))))
false) $\mathcal{C}'_{\text{cps}}[\chi(E)]$))))
false) $\mathcal{C}'_{\text{cps}}[\chi(E)]$)

app4

(((($\mathcal{C}_{\text{cps}}$[*rand-expr*]
(λ (*rand-value*)
((((($\mathcal{C}'_{\text{cps}}[v_0]$ *rand-value*) $\mathcal{C}[E]$) $\xi(E)$) $\mathcal{C}'_{\text{cps}}[\chi(E)]$))))
false) $\mathcal{C}'_{\text{cps}}[\chi(E)]$)

app5

((((($\mathcal{C}'_{\text{cps}}[v_0]$ $\mathcal{C}'_{\text{cps}}[v_1]$) $\mathcal{C}[E]$) $\xi(E)$) $\mathcal{C}'_{\text{cps}}[\chi(E)]$)

((((($\mathcal{C}'_{\text{cps}}[(\lambda\ (x)\ e_0)]$ $\mathcal{C}'_{\text{cps}}[v_1]$) $\mathcal{C}[E]$) $\xi(E)$) $\mathcal{C}'_{\text{cps}}[\chi(E)]$)

app6

(((($\mathcal{C}_{\text{cps}}[e_0][x \leftarrow \mathcal{C}'_{\text{cps}}[v_1]]$ $\mathcal{C}[E]$) $\xi(E)$) $\mathcal{C}'_{\text{cps}}[\chi(E)]$)

B.2 wcm form

$\mathcal{C}_{\text{cps}}[E[($**wcm** *mark-expr body-expr*$)]]$

((((λ (*kont*)
(λ (*flag*)
(λ (*marks*)
(((($\mathcal{C}_{\text{cps}}$[*mark-expr*]
(λ (*mark-value*)
((λ (*rest-marks*)
((((($\mathcal{C}_{\text{cps}}$[*body-expr*] *kont*) **true**) $\mathcal{C}'_{\text{cps}}$[((**cons** *mark-value*) *rest-marks*)]))
((*flag* $\hat{\mathcal{C}}_{\text{cps}}$[(**snd** *marks*)]) *marks*))))
false) *marks*))))
$\mathcal{C}[E]$) $\xi(E)$) $\mathcal{C}'_{\text{cps}}[\chi(E)]$)

wcm1

(((λ (*flag*)

$$
\begin{array}{l}
\quad (\lambda\ (\mathit{marks}) \\
\quad\ \ (((\mathcal{C}_{\mathrm{cps}}[\mathit{mark\text{-}expr}] \\
\qquad\quad (\lambda\ (\mathit{mark\text{-}value}) \\
\qquad\quad\ \ ((\lambda\ (\mathit{rest\text{-}marks}) \\
\qquad\quad\quad\ \ (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathcal{C}[E])\ \mathbf{true})\ \mathcal{C}'_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})])) \\
\qquad\quad\ \ ((\mathit{flag}\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \mathit{marks})])\ \mathit{marks})))) \\
\qquad\ \ \mathbf{false})\ \mathit{marks}))) \\
\ \ \xi(E))\ \mathcal{C}'_{\mathrm{cps}}[\chi(E)])
\end{array}
$$

wcm2

$$
\begin{array}{l}
((\lambda\ (\mathit{marks}) \\
\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{mark\text{-}expr}] \\
\qquad (\lambda\ (\mathit{mark\text{-}value}) \\
\qquad\ \ ((\lambda\ (\mathit{rest\text{-}marks}) \\
\qquad\quad\ \ (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathcal{C}[E])\ \mathbf{true})\ \mathcal{C}'_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})])) \\
\qquad\ \ ((\xi(E)\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \mathit{marks})])\ \mathit{marks})))) \\
\quad\ \ \mathbf{false})\ \mathit{marks})) \\
\mathcal{C}'_{\mathrm{cps}}[\chi(E)])
\end{array}
$$

wcm3

$$
\begin{array}{l}
(((\mathcal{C}_{\mathrm{cps}}[\mathit{mark\text{-}expr}] \\
\qquad (\lambda\ (\mathit{mark\text{-}value}) \\
\qquad\ \ ((\lambda\ (\mathit{rest\text{-}marks}) \\
\qquad\quad\ \ (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathcal{C}[E])\ \mathbf{true})\ \mathcal{C}'_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})])) \\
\qquad\ \ ((\xi(E)\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \chi(E))])\ \mathcal{C}'_{\mathrm{cps}}[\chi(E)]))))) \\
\ \ \mathbf{false})\ \mathcal{C}'_{\mathrm{cps}}[\chi(E)])
\end{array}
$$

wcm4tail

$$
\begin{array}{l}
((\lambda\ (\mathit{rest\text{-}marks}) \\
\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathcal{C}[E])\ \mathbf{true})\ \mathcal{C}'_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})])) \\
\ ((\mathbf{true}\ \hat{\mathcal{C}}_{\mathrm{cps}}[(\mathbf{snd}\ \chi(E))])\ \mathcal{C}'_{\mathrm{cps}}[\chi(E)]))
\end{array}
$$

wcm5tail

$$
\begin{array}{l}
((\lambda\ (\mathit{rest\text{-}marks}) \\
\quad (((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathcal{C}[E])\ \mathbf{true})\ \mathcal{C}'_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ \mathit{rest\text{-}marks})])) \\
\ \mathcal{C}'_{\mathrm{cps}}[(\mathbf{snd}\ \chi(E))])
\end{array}
$$

wcm6tail

$$
(((\mathcal{C}_{\mathrm{cps}}[\mathit{body\text{-}expr}]\ \mathcal{C}[E])\ \mathbf{true})\ \mathcal{C}'_{\mathrm{cps}}[((\mathbf{cons}\ \mathit{mark\text{-}value})\ (\mathbf{snd}\ \chi(E)))])
$$

wcm4nontail

$$
((\lambda\ (\mathit{rest\text{-}marks})
$$

$(((\mathcal{C}_{\text{cps}}[\textit{body-expr}]\ \mathcal{C}[E])\ \textbf{true})\ \mathcal{C}'_{\text{cps}}[((\textbf{cons}\ \textit{mark-value})\ \textit{rest-marks})]))$
$((\textbf{false}\ \hat{\mathcal{C}}_{\text{cps}}[(\textbf{snd}\ \chi(E))])\ \mathcal{C}'_{\text{cps}}[\chi(E)]))$

wcm5nontail

$((\lambda\ (\textit{rest-marks})$
$(((\mathcal{C}_{\text{cps}}[\textit{body-expr}]\ \mathcal{C}[E])\ \textbf{true})\ \mathcal{C}'_{\text{cps}}[((\textbf{cons}\ \textit{mark-value})\ \textit{rest-marks})]))$
$\mathcal{C}'_{\text{cps}}[\chi(E)])$

wcm6nontail

$(((\mathcal{C}_{\text{cps}}[\textit{body-expr}]\ \mathcal{C}[E])\ \textbf{true})\ \mathcal{C}'_{\text{cps}}[((\textbf{cons}\ \textit{mark-value})\ \chi(E))])$

B.3 ccm form

$\mathcal{C}_{\text{cps}}[E[(\textbf{ccm})]]$

$((((\lambda\ (\textit{kont})$
$(\lambda\ (\textit{flag})$
$(\lambda\ (\textit{marks})$
$(\textit{kont}\ \textit{marks}))))$
$\mathcal{C}[E])\ \xi(E))\ \mathcal{C}'_{\text{cps}}[\chi(E)])$

ccm1

$(((\lambda\ (\textit{flag})$
$(\lambda\ (\textit{marks})$
$(\mathcal{C}[E]\ \textit{marks})))$
$\xi(E))\ \mathcal{C}'_{\text{cps}}[\chi(E)])$

ccm2

$((\lambda\ (\textit{marks})$
$(\mathcal{C}[E]\ \textit{marks}))$
$\mathcal{C}'_{\text{cps}}[\chi(E)])$

ccm3

$(\mathcal{C}[E]\ \mathcal{C}'_{\text{cps}}[\chi(E)])$

B.4 Value Form

$\mathcal{C}_{\text{cps}}[E[v]]$=$\mathcal{C}_{\text{cps}}[E[(\lambda\ (x)\ e)]]$

$((((\lambda\ (\textit{kont})$
$(\lambda\ (\textit{flag})$
$(\lambda\ (\textit{marks})$
$(\textit{kont}\ (\lambda\ (x)\ \mathcal{C}_{\text{cps}}[e])))))$
$\mathcal{C}[E])\ \xi(E))\ \mathcal{C}'_{\text{cps}}[\chi(E)])$

value1

$$(((\lambda\ (\mathit{flag})\\ \quad (\lambda\ (\mathit{marks})\\ \quad\quad (\mathcal{C}[E]\ (\lambda\ (x)\ \mathcal{C}_{\mathrm{cps}}[e])))))\\ \ \xi(E))\ \mathcal{C}'_{\mathrm{cps}}[\chi(E)])$$

value2

$$((\lambda\ (\mathit{marks})\\ \quad (\mathcal{C}[E]\ (\lambda\ (x)\ \mathcal{C}_{\mathrm{cps}}[e])))\\ \mathcal{C}'_{\mathrm{cps}}[\chi(E)])$$

value3

$$(\mathcal{C}[E]\ (\lambda\ (x)\ \mathcal{C}_{\mathrm{cps}}[e]))$$

B.5 Variable Form

$\mathcal{C}_{\mathrm{cps}}[E[x]]$

$$((((\lambda\ (\mathit{kont})\\ \quad (\lambda\ (\mathit{flag})\\ \quad\quad (\lambda\ (\mathit{marks})\\ \quad\quad\quad (\mathit{kont}\ x))))\\ \quad \mathcal{C}[E])\ \xi(E))\ \mathcal{C}'_{\mathrm{cps}}[\chi(E)])$$

x1

$$(((\lambda\ (\mathit{flag})\\ \quad (\lambda\ (\mathit{marks})\\ \quad\quad (\mathcal{C}[E]\ x)))\\ \ \xi(E))\ \mathcal{C}'_{\mathrm{cps}}[\chi(E)])$$

x2

$$((\lambda\ (\mathit{marks})\\ \quad (\mathcal{C}[E]\ x))\\ \mathcal{C}'_{\mathrm{cps}}[\chi(E)])$$

x3

$$(\mathcal{C}[E]\ x)$$

x4

$$(\mathcal{C}[E]\ \mathsf{error})$$

x5

error