Pure Continuation Marks

Kimball Germane and Jay McCarthy

Brigham Young University, Provo, Utah

Abstract. Continuation marks are a programming language feature which generalizes 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 transformation from the call-by-value λ-calculus augmented with continuation marks to the pure call-by-value λ-calculus. We discuss how such transformations simplify the construction of compilers which treat continuation marks correctly. We document how Redex, a domain-specific language for exploring language semantics, aided the discovery of correct transformations. We offer the sketch proof of a meaning-preservation theorem. Finally, we apply the transformation to JavaScript.

1 Introduction

Numerous program 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. In each of these scenarios, the instrument relies on implementation-specific information and must be maintained as the instrumented language undergoes optimizations and ventures across platforms. This makes these instruments brittle and increases the porting cost of the language ecosystem. Each 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 [3] are a programming language feature which generalizes stack inspection. Not only do they dramatically simplify correct instrumentation [5], they have been used to allow inspection-based dynamic security checks in the presence of tail-call optimization [4], to express aspect-oriented programming in higher-order languages [14], and to implement a form of dynamic binding in parameters.

Continuation marks originated in PLT Scheme (now Racket [9]). However, 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 [6], but it remains a proof-of-concept.)

To avoid this roadblock, we instead enhance a core language with facilities to manipulate continuation marks and desugar the enhanced language back to the core. To make our desugaring transformation portable to other 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).

We begin by further explaining continuation marks in section 2. We then formalize them via the introduction of a core and enhanced language in section 3. We discuss the necessary properties of a meaningful transformation between these languages. We present a transformation of continuation marks to the call-by-value λ-calculus in section 4. This is followed by randomized testing using a lightweight mechanized model in section 5 and a sketch proof of a meaning-preservation theorem in section 6. We finally apply it to JavaScript in section 7.

2 Continuation Marks

Continuation marks allow the program to annotate and observe the stack (or continuation). This is accomplished via two surface-level syntactic forms in the language: **with-continuation-mark**, abbreviated **wcm**; and **current-continuation-marks**, abbreviated **ccm**.

A **wcm** expression of the form (**wcm** *mark-expr body-expr*) annotates the youngest portion of the continuation with *mark-value*, the evaluation of *mark-expr*, before evaluating *body-expr* to *body-value*. If an annotation, or *mark*, already exists on the youngest portion, it is replaced by the new mark. This newly-added mark is available within *body-expr* via **ccm** during its evaluation. Once complete, the entire **wcm** expression takes on *body-value*.

A **ccm** expression of the form (**ccm**) traverses the continuation accumulating a list of annotations ascending in age.

2.1 Example

The traditional, properly-recursive factorial function can be expressed with continuation marks as follows:

```
(define (fac n)
  (if (zero? n)
      (begin
        (print (ccm))
        1)
      (wcm n (* n (fac (- n 1))))))
```

In this definition, a multiplication pends following the recursive call, so the continuation grows with each.

The effect of evaluating the factorial of 2, expressed as (**fac** 2), is

```
> (fac 2)
(1 2)
2
```

In a tail-call optimized language, a call made in tail position does not enlarge the continuation. Instead, the portion of the evaluation context dedicated to the calling function is repurposed for the called function. Continuation marks are also subject to this optimization; if the continuation is marked in tail position, the previous mark there is replaced, if it exists.

This behavior is apparent in the tail-recursive definition of the factorial function, which follows.

```
(define (fac n acc)
  (if (zero? n)
      (begin
        (print (ccm))
        acc)
      (wcm n (fac (- n 1) (* n acc)))))
```

In this definition, the multiplication is performed before the recursive call and an accumulated value is passed. With no pending computation, the evaluation context devoted to this function is no longer necessary, and is subsumed by the continuation mark directive.

The effect of evaluating the factorial of 2, here expressed by (**fac** 2 1), is

```
> (fac 2 1)
(1)
2
```

from which the overwriting which occurred is evidenced the singleton list obtained by (**ccm**).

Using this mechanism, the principal onus of a statistical profiler, algebraic stepper, or debugger can be met by a straightforward language transformation which recursively wraps each term in a **wcm** directive annotating the continuation with a description of the wrapped term.

With this understanding of continuation marks, we can now formalize them behaviorally.

3 Language Core and Extension

In order to formalize continuation marks, we introduce an extension of the λ-calculus with facilities to manipulate continuation marks and present its semantics in the style of Felleisen and Hieb [7]. Because our ultimate goal is a desugaring transform, we first introduce the target language of the transform, the λ-calculus, in the same way.

3.1 $\boldsymbol{\lambda_v}$

The target language of the transform is Plotkin's call-by-value λ-calculus, λ_v [13], augmented with natural numbers.

Terms e in λ_v are the familiar terms of the λ-calculus, defined by

$$e = (e\,e) \mid v \mid x \tag{1}$$

where

$$v = (\lambda\,(x)\,e) \mid n \tag{2}$$

with $n \in \mathbb{N}$.

The evaluation model of λ_v requires a definition of evaluation contexts. We define evaluation contexts E by

$$E = (E\,e) \mid (v\,E) \mid \bullet \tag{3}$$

where $\bullet$ denotes a "hole" in the evaluation context, the ultimate destination of the evaluation of the expression that previously resided there.

The reduction relation of λ_v is defined simply by

$$E[((\lambda\,(x)\,e)\,v)] \rightarrow E[e[x \leftarrow v]] \tag{4}$$

where $e[x \leftarrow v]$ denotes a capture-avoiding substitution of every free occurrence of x in e with v.

3.2 $\boldsymbol{\lambda_{cm}}$

The source language of the transformation is an extension of λ_v with facilities for continuation marks which we term λ_{cm}.

Because λ_{cm} is an extension of λ_v, the definition of terms e in λ_{cm}

$$e = (\mathrm{wcm}\,e\,e) \mid (\mathrm{ccm}) \mid (e\,e) \mid v \mid x \tag{5}$$

is identical to that of λ_v with the addition of two forms for manipulating continuation marks: **wcm**, short for **with-continuation-mark**, which annotates the continuation with the evaluation of the given mark expression; and **ccm**, short for **current-continuation-marks**, which retrieves the current continuation marks.

The sole novelty of λ_{cm} above the plain λ_v is the ability to annotate and observe the continuation. Our evaluation model of λ_{cm}, given by Pettyjohn et al. [12], utilizes the current evaluation context for that purpose. Evaluation contexts E are defined by

$$E = (\mathrm{wcm}\, v\, F) \mid F \tag{6}$$
$$F = (E\, e) \mid (v\, E) \mid (\mathrm{wcm}\, E\, e) \mid \bullet \tag{7}$$

The definition of E is crafted to prevent directly nested **wcm** directives from occurring in valid evaluation contexts.

The reduction relation of λ_{cm} is defined by

$$E[((\lambda\,(x)\, e)\, v)] \rightarrow E[e[x \leftarrow v]] \tag{8}$$
$$E[(\mathrm{wcm}\, v\, (\mathrm{wcm}\, v'\, e))] \rightarrow E[(\mathrm{wcm}\, v'\, e)] \tag{9}$$
$$E[(\mathrm{wcm}\, v\, v')] \rightarrow E[v'] \tag{10}$$
$$E[(\mathrm{ccm})] \rightarrow E[\chi(E)] \tag{11}$$

As an extension of λ_v, λ_{cm} inherits rule 8 with substitution carried through **wcm** terms.

If directly-nested **wcm** directives are introduced into the evaluation context, invalidating it (as in the tail-recursive factorial example), rule 9 collapses the outer into the inner.

Rule 10 defines the value of a **wcm** expression to take on the value of the body, once obtained.

The definition of rule 11 is given in terms of a metafunction χ defined by

$$\chi(E) = \chi'(E, (\lambda\,(x)\,(\lambda\,(y)\, y))) \tag{12}$$

where χ' is defined by

$$\chi'((\mathrm{wcm}\, v\, F), vs) = \chi'(F, (\lambda\,(z)\,((z\, v)\, vs))) \tag{13}$$
$$\chi'((E\, e), vs) = \chi'(E, vs) \tag{14}$$
$$\chi'((v\, E), vs) = \chi'(E, vs) \tag{15}$$
$$\chi'((\mathrm{wcm}\, E\, e), vs) = \chi'(E, vs) \tag{16}$$
$$\chi'(\bullet, vs) = vs \tag{17}$$

This definition formalizes the intuition given earlier regarding the behavior of **ccm**. Previously, we saw that the value of a **ccm** directive was a list; however, lists do not strictly exist as values in the λ-calculus, nor in our extension. Instead of adding lists as primitive values to λ_{cm}, we employ Church encodings to represent them natively [2]. To keep things clear, we preserve the intention of Church-encoded terms by representing them with the useful shorthand of fig. 3.2.

Let $\rightarrow_v$ denote the reduction relation for λ_v and $\rightarrow_v^*$ its transitive closure. Similarly, let $\rightarrow_{cm}$ denote the reduction relation for λ_{cm} and $\rightarrow_{cm}^*$ its transitive closure. We will use the more specific notation when $\rightarrow$ is potentially ambiguous.

With the source and target languages formally specified, we can now examine a language transformation in earnest.

$$\begin{aligned}
\textbf{true} &= (\lambda\,(x)\,(\lambda\,(y)\,x)) \\
\textbf{false} &= (\lambda\,(x)\,(\lambda\,(y)\,y)) \\
\textbf{cons} &= (\lambda\,(a)\,(\lambda\,(b)\,(\lambda\,(z)\,((z\,a)\,b)))) \\
\textbf{snd} &= (\lambda\,(p)\,(p\,\textbf{false})) \\
\textbf{nil} &= \textbf{false}
\end{aligned}$$

Fig. 1. Church encodings of booleans and lists

4 Transformation

As transformations, desugarings put the meaning of one construct in terms of another. If the meaning of the former was given only informally, the desugaring offers a new definition as formal as the definition of the latter. In this way, desugarings offer a convenient way to provide meaning to a construct and avoid enlarging the language. If, however, the sugared language already has a formal meaning, as does λ_{cm}, the desugaring must preserve it.

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, for programs $p \in \lambda_{cm}$,

$$\begin{tikzcd}
p \arrow[r, "*_{cm}"] \arrow[d, "\mathcal{C}"] & v \arrow[d, "\mathcal{C}"] \\
\mathcal{C}[p] \arrow[r, "*_{v}"] & \mathcal{C}[v]
\end{tikzcd}$$

should hold. If we define

$$\text{eval}_{cm}(p) = \begin{cases} v & \text{if } p \rightarrow^*_{cm} v \\ \perp & \text{if } p \rightarrow^*_{cm} \cdots \end{cases} \tag{18}$$

and

$$\text{eval}_{v}(p) = \begin{cases} v & \text{if } p \rightarrow^*_{v} v \\ \perp & \text{if } p \rightarrow^*_{v} \cdots \end{cases} \tag{19}$$

we can state this more concisely by

$$\mathcal{C}[\text{eval}_{cm}(p)] = \text{eval}_v(\mathcal{C}[p]) \tag{20}$$

4.1 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 9. 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.

4.2 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, $\mathcal{C}[(\textit{rator-expr rand-expr})]$, 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 $\mathcal{C}[\textit{rator-expr}]$ and $\mathcal{C}[\textit{rand-expr}]$ 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, $\mathcal{C}[(\textbf{wcm}\ \textit{mark-expr body-expr})]$, 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 $\mathcal{C}[\textit{body-expr}]$ with the correct tail-position flag and current continuation marks.

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

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

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 similar.

Now, to address the absence of **let**, **if**, **cons**, etc. from λ_v: We can express the **let** construct in λ_v with application. To achieve **if** and conditionals as well as list primitives **cons**, **snd**, and **nil**, we use the Church encodings of fig. 1.

4.3 Initiation

In our call-by-value language, abstracting terms has the effect of suspending evaluation. When an entire program is transformed, all evaluation is suspended, awaiting 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

$$\hat{\mathcal{C}}[p] = ((\mathcal{C}[p]\,\mathbf{false})\,\mathbf{nil}) \tag{21}$$

This changes our commutativity property somewhat, since applying $\hat{\mathcal{C}}$ to a value–say, the result of eval_{cm}–will instigate further, if benign, reduction. In light of this, we must alter our statement about commutativity to reflect that we no longer guarantee term equality, but term equivalence, in the sense that they share a normal form (or lack of one). We can state this modified property as

$$\hat{\mathcal{C}}[\text{eval}_{cm}(p)] \equiv \text{eval}_v(\hat{\mathcal{C}}[p]) \tag{22}$$

which, expanded, is

$$((\mathcal{C}[\text{eval}_{cm}(p)]\,\mathbf{false})\,\mathbf{nil}) \equiv \text{eval}_v(((\mathcal{C}[p]\,\mathbf{false})\,\mathbf{nil})) \tag{23}$$

4.4 Some Final Subtleties

We have not added lists as primitive values to λ_{cm}but instead encode lists with other terms in the language. By not relying on a particular characterization of lists in the target language, the dependencies of the transformation remain few, and thus widen its possible application. However, this makes lists themselves subject to the transformation which complicates their manifestation in the core language. The effect of this is that the transformation must deal with the list of continuation marks 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

$$\mathcal{C}'[(\lambda\,(x)\,e)] = (\lambda\,(x)\,\mathcal{C}[e]) \tag{24}$$

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

$$\hat{\mathcal{C}}[p] = ((\mathcal{C}[p]\,\mathbf{false})\,\mathcal{C}'[\mathbf{nil}]) \tag{25}$$

4.5 Definition of $\mathcal{C}$

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

Definition 1. $\mathcal{C}[(\mathit{rator\text{-}expr}\ \mathit{rand\text{-}expr})]$

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

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

Definition 2. $\mathcal{C}[(\mathbf{wcm}\ \mathit{mark\text{-}expr}\ \mathit{body\text{-}expr})]$

The formal transformation of a **wcm** directive is also extremely similar to the **let** version. The Church-encoded conditional eagerly evaluates both branches, but this still achieves correct behavior, as *marks* is already a value and (**snd** *marks*) is benign, even if *marks* is **nil**.

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

Definition 3. $\mathcal{C}[(\mathbf{ccm})]$

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

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

Definition 4. $\mathcal{C}[v]$=$\mathcal{C}[(\lambda\ (x)\ e)]$

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

$(\lambda\ (\mathit{flag})$
$\quad(\lambda\ (\mathit{marks})$
$\qquad(\lambda\ (x)\ \mathcal{C}[e])))$

Definition 5. $\mathcal{C}[x]$

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 $\mathcal{C}[x][x \leftarrow \mathcal{C}'[(\lambda\,(y)\,y)]] = \mathcal{C}[x[x \leftarrow (\lambda\,(y)\,y)] = \mathcal{C}[(\lambda\,(y)\,y)]$.

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

4.6 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 $(\mathbf{wcm}\ 0\ ((\lambda\ (x)\ (\mathbf{wcm}\ x\ (\mathbf{ccm})))\ 1))$. It reduces according to λ_{cm} semantics as

$(\mathbf{wcm}\ 0\ ((\lambda\ (x)\ (\mathbf{wcm}\ x\ (\mathbf{ccm})))\ 1))$
$(\mathbf{wcm}\ 0\ (\mathbf{wcm}\ 1\ (\mathbf{ccm})))$
$(\mathbf{wcm}\ 1\ (\mathbf{ccm}))$
$(\mathbf{wcm}\ 1\ (\lambda\ (z)\ ((z\ 1)\ (\lambda\ (x)\ (\lambda\ (y)\ y)))))$
$(\lambda\ (z)\ ((z\ 1)\ (\lambda\ (x)\ (\lambda\ (y)\ y))))$

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.

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

By definition this is

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

which explodes upon expansion to

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

After the application of contextual information, we reach

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

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:

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

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

$$\begin{array}{l}
(((\lambda\ (\mathit{flag})\\
\qquad (\lambda\ (\mathit{marks})\\
\qquad\quad (((((\mathcal{C}[(\lambda\ (x)\ (\mathbf{wcm}\ x\ (\mathbf{ccm})))]\ \mathbf{false})\ \mathit{marks})\\
\qquad\qquad ((\mathcal{C}[1]\ \mathbf{false})\ \mathit{marks}))\\
\qquad\qquad \mathit{flag})\\
\qquad\quad\ \ \mathit{marks})))\ \mathbf{true})\ \mathcal{C}'[((\mathbf{cons}\ 0)\ \mathbf{nil})])
\end{array}$$

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:

$$\begin{array}{l}
((((\lambda\ (x)\ \mathcal{C}[(\mathbf{wcm}\ x\ (\mathbf{ccm}))])\\
\quad 1)\ \mathbf{true})\ \mathcal{C}'[((\mathbf{cons}\ 0)\ \mathbf{nil})])
\end{array}$$

reduces to

$$\begin{array}{l}
((\mathcal{C}[(\mathbf{wcm}\ 1\ (\mathbf{ccm}))]\\
\quad \mathbf{true})\ \mathcal{C}'[((\mathbf{cons}\ 0)\ \mathbf{nil})])
\end{array}$$

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

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

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

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

the list is beheaded to simulate mark overwriting:

$((\lambda\ (\mathit{marks})\ \mathit{marks})$
$\ ((\lambda\ (\mathit{rest\text{-}marks})\ \mathcal{C}'[((\mathbf{cons}\ 1)\ \mathit{rest\text{-}marks})])$
$\ \ \hat{\mathcal{C}}[(\mathbf{snd}\ ((\mathbf{cons}\ 0)\ \mathbf{nil}))]))$

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

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

reduces to

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

and we are left with just what we hoped for.

5 Testing

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. [10] 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 [8], a domain-specific language for exploring language semantics, to provide feedback, thoroughly exercise candidates, and perform exploratory analysis.

The correctness of the transform lies in the property that it commutes with evaluation. In order to test for this property, we must construct evaluation models for the source language λ_{cm} and the target language λ_v. Since λ_{cm} is an extension of λ_v, many of its forms and semantics are inherited. Redex allows us to exploit this fact by defining a model for λ_v first and then extending that model with the enhancements of λ_{cm}.

After constructing the models, we are prepared to test various semantic properties of these two languages. Of course, we are particularly interested in testing the commutativity of our transformation.

We can test that the property described by equation 22 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 term 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 the transformation to random testing. We used the specific counterexamples to guide modifications to the transform until it withstood 10,000 random tests. Interestingly, no incorrect transformation withstood more than 500 random tests before failing.

6 Proof

With a reasonable intuition and definitions hardened by random testing, we lightly stetch a proof of the correctness of $\mathcal{C}$.

We begin by expressing the commutativity property we have sought to preserve as a theorem:

Theorem 1. *For all programs $p \in \lambda_{cm}$, $\hat{\mathcal{C}}[\mathrm{eval}_{cm}(p)] = \mathrm{eval}_v(\hat{\mathcal{C}}[p])$.*

We prove this by overloading $\mathcal{C}$ to accommodate first evaluation contexts and then context-term pairs (where $E[e]$ represents (E, e)) by

Definition 6. $\mathcal{C}[E[e]]$

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

which allows us to formally relate $E[e]$ and $\mathcal{C}[E[e]]$.

We then prove a lemma

Lemma 1 (Simulation). *For all contexts $E \in \lambda_{cm}$ and expressions $e \in \lambda_{cm}$, $E[e] \rightarrow_{cm} E'[e'] \implies \mathcal{C}[E[e]] \rightarrow_v^* \mathcal{C}[E'[e']]$*

by induction over contexts E and terms e. This lemma justifies the evolution of the evaluation context and each rule in the reduction relation except rule 8. We preserve this rule with

Lemma 2 (Substitution). *For all $e, x, v \in \lambda_{cm}$, $\mathcal{C}[e[x \leftarrow v]] = \mathcal{C}[e][x \leftarrow \mathcal{C}'[v]]$.*

With these lemmas, the theorem follows quickly.

7 Back to JavaScript

With a correct transformation of continuation marks defined over the λ-calculus, little effort is required to add continuation marks to an eager, higher-order language such as JavaScript.

We add two keywords to manipulate continuation marks: the familiar `wcm` and `ccm`. We chose block syntax for `wcm` to convey that it is a special form, like a conditional, instead of a function-like interface. A lone `ccm` evaluates to the current continuation marks.

Tail-call elimination is not part of the JavaScript specification which complicates the treatment of the tail-call behavior. We simulate proper tail-call behavior by using the flag to encode whether a function call is in tail position.

A properly-recursive factorial can be implemented in this JavaScript extension as

```
var fac = function( n ) {
  if( n == 0 ) {
    console.log( ccm );
    return 1;
  }
  else {
    return wcm { n }
               { n * fac( n - 1 ) };
  }
}
```

A direct-style transformation of this function desugars these constructs into vanilla JavaScript, leveraging native arrays for lists of continuation marks.

```
var fac = function( flag, marks ) {
  return function( n ) {
    if( n == 0 ) {
      console.log( (function( flag, marks ) { return marks; })( false, marks ) );
      return 1;
    }
    else {
      return (function( flag, marks ) {
        return (function( mark_value, rest_marks ) {
          return n * fac( false, [ mark_value ].concat( rest_marks ) )( n - 1 );
        })( n, flag ? marks.slice( 1 ) : marks );
      })( flag, marks );
    }
  }
}
```

As expected, the output of a manual initiation of this function is

```
> fac( false, [] )( 5 )
[ 1, 2, 3, 4, 5 ]
120
```

A tail-recursive factorial is expressible in this extension as

```
var fac = function( n, acc ) {
  if( n == 0 ) {
    console.log( ccm );
    return acc;
  }
  else {
    return wcm { n }
               { fac( n - 1, n * acc ) };
  }
}
```

and transforms to

```
var fac = function( flag, marks ) {
  return function( n, acc ) {
    if( n == 0 ) {
      console.log( (function( flag, marks ) { return marks; })( false, marks ) );
      return acc;
    }
    else {
      return (function( flag, marks ) {
        return (function( mark_value, rest_marks ) {
          return fac( true, [ mark_value ].concat( rest_marks ) )( n - 1, n * acc );
        })( n, flag ? marks.slice( 1 ) : marks );
      })( flag, marks );
    }
  }
}
```

The output of a manual initiation of this function is

```
> fac( false, [] )( 5, 1 )
[ 1 ]
120
```

In this example, the API to compute factorials has changed, but the change is localized. With top-level control of a program, we can transform the relevant part of the system, leaving the rest the same. This preserves interoperability with foreign functions, i.e., third-party JavaScript libraries.

8 Related Work

Monads, introduced by Moggi [11], are another way to model semantics. Their appeal is significant: they allow one to implement a particular model of computation with a pure computational logic, such as the λ-calculus.

Expressing continuation marks in a monadic way is simple enough. In Haskell,

```
data CM m a = CM ((Bool,[m]) -> a)

instance Monad (CM m) where
  return x = CM (\_ -> x)
  (CM m) >>= f = CM (\(flag,vs) ->
      let (CM m') = f (m (False,vs)) in m' (flag,vs))

wcm :: m -> CM m a -> CM m a
wcm v (CM m) = CM (\(f,vs) -> m (True, v:(if f then (tail vs) else vs)))

ccm :: CM m [m]
ccm = CM (\(_,vs) -> vs)

runCM :: CM m a -> a
runCM (CM m) = m (False,[])
```

is one way, using our approach. The factorial functions leveraging this implementation exhibit the correct continuation mark behavior but this is *not* a monad.

Specifically, the right identity law, `m >>= return = m`, does not hold since `CM` conceptually adds a stack frame in bind (`>>=`). Ager et al. [1] implemented a similar stack-inspection monad in terms of a lifted state monad, but resorted to manual management of the stack frames.

9 Conclusion and Future Work

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.

References

1. M.S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Principles and practice of declarative programming*, 2003.
2. H.P. Barendregt. *The lambda calculus: Its syntax and semantics*. 1984.
3. J. Clements. *Portable and high-level access to the stack with Continuation Marks*. PhD thesis, Northeastern University, 2006.
4. J. Clements and M. Felleisen. A tail-recursive machine with stack inspection. *Transactions on Programming Languages and Systems*, 2004.
5. J. Clements, M. Flatt, and M. Felleisen. Modeling an algebraic stepper. *Programming Languages and Systems*, 2001.
6. J. Clements, A. Sundaram, and D. Herman. Implementing continuation marks in javascript. *Computer Science and Software Engineering*, 2008.
7. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, 1992.
8. R.B. Findler and C. Klein. Redex: Practical semantics engineering. 2010.
9. Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. http://racket-lang.org/tr1/.
10. 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 *Principles of programming languages*, 2012.
11. E. Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science*, 1989.
12. G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen. Continuations from generalized stack inspection. In *ACM SIGPLAN Notices*, 2005.
13. G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, 1975.
14. D.B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Aspect-oriented software development*, 2003.