Continuations from Generalized Stack Inspection

Greg Pettyjohn
Northeastern University

John Clements
Northeastern University

Joe Marshall
Northeastern University

Shriram Krishnamurthi
Brown Univerity

Matthias Felleisen
Northeastern University

Motivation: Continuations, VMs, and the Web

We present a translation for eliminating call/cc using PLT Scheme's Continuation Marks and prove correctness of the translation.

Motivation: Continuations, VMs, and the Web

We present a translation for eliminating call/cc using PLT Scheme's Continuation Marks and prove correctness of the translation.

- First consider two situations that expose the requirements for the translation:

Motivation: Continuations, VMs, and the Web

We present a translation for eliminating call/cc using PLT Scheme's Continuation Marks and prove correctness of the translation.

- First consider two situations that expose the requirements for the translation:

    1. Web programming with continuations

Motivation: Continuations, VMs, and the Web

We present a translation for eliminating call/cc using PLT Scheme's Continuation Marks and prove correctness of the translation.

- First consider two situations that expose the requirements for the translation:
  1. Web programming with continuations
  2. Implementing call/cc on standard VMs

Continuations and the Web

When an interactive Web program issues a Web response, the client may decide to answer the response zero or more times, thus re-launching the rest of the servlet's computation zero or more times.

Continuations and the Web

When an interactive Web program issues a Web response, the client may decide to answer the response zero or more times, thus re-launching the rest of the servlet's computation zero or more times.

- The "rest of the servlet's computation" is essentially a *continuation* that must be stored and used possibly several times.

Continuations and the Web

When an interactive Web program issues a Web response, the client may decide to answer the response zero or more times, thus re-launching the rest of the servlet’s computation zero or more times.

- The “rest of the servlet’s computation” is essentially a *continuation* that must be stored and used possibly several times.
- This has lead many to believe that a servlet language that supports first-class continuations is a natural choice for the Web.

Two Approaches to Servlets

We have explored two approaches to continuation based Web programming:

Two Approaches to Servlets

We have explored two approaches to continuation based Web programming:

1. Start with a language that already has native support for continuations and add Web programming capabilities via a custom Web server.

Two Approaches to Servlets

We have explored two approaches to continuation based Web programming:

1. Start with a language that already has native support for continuations and add Web programming capabilities via a custom Web server.

2. Start with a Web programming language that also has continuations and automatically restructure Web programs to run on a standard framework.

Native Continuations

Native Continuations

- Requires support from a custom Web server.

Native Continuations

- Requires support from a custom Web server.
- Continuations are separated from Web Responses.

Native Continuations

- Requires support from a custom Web server.
- Continuations are separated from Web Responses.
- Storing continuations uses resources on the server.

Native Continuations

- Requires support from a custom Web server.
- Continuations are separated from Web Responses.
- Storing continuations uses resources on the server.
- Must essentially *guess* the lifetime of a continuation.

Automatic Restructuring of Web Programs

Automatic Restructuring of Web Programs

+ The translation has control over the representation of continuaions.

Automatic Restructuring of Web Programs

- + The translation has control over the representation of continuaions.
- + No longer need a custom Web server.

Automatic Restructuring of Web Programs

+ The translation has control over the representation of continuaions.
+ No longer need a custom Web server.
+ Continuations can be encoded in the Web Response, perhaps even in the URL. Can therefore avoid storing extra resources on the server and can support bookmarking.

Automatic Restructuring of Web Programs

- The translation has control over the representation of continuaions.
- No longer need a custom Web server.
- Continuations can be encoded in the Web Response, perhaps even in the URL. Can therefore avoid storing extra resources on the server and can support bookmarking.
- Continuation expires exactly when the Response goes out of existence. Perfect!

Implementing Scheme on Standard VMs

Standard VMs do not provide direct support for installing and saving the runtime stack.

Implementing Scheme on Standard VMs

Standard VMs do not provide direct support for installing and saving the runtime stack.

- Give up on call/cc

Implementing Scheme on Standard VMs

Standard VMs do not provide direct support for installing and saving the runtime stack.

- Give up on call/cc
- Translate programs to a form that does not rely on direct support for call/cc

call/cc – CPS Approach

call/cc – CPS Approach

- CPS is a whole program transformation that changes the calling signature for each function in the program.

call/cc – CPS Approach

- CPS is a whole program transformation that changes the calling signature for each function in the program.
- CPS programs essentially manage their own stack on the heap.

call/cc – CPS Approach

- CPS is a whole program transformation that changes the calling signature for each function in the program.
- CPS programs essentially manage their own stack on the heap.
- Need proper tail calls or else use a trampoline.

call/cc – CPS Approach

- CPS is a whole program transformation that changes the calling signature for each function in the program.
- CPS programs essentially manage their own stack on the heap.
- Need proper tail calls or else use a trampoline.
- Implementing the stack on the heap precludes using standard tools and runtime optimizations.

call/cc – CMT Approach

We offer an alternative: *C*ontinuation *M*ark *T*ransform.

call/cc – CMT Approach

We offer an alternative: *C*ontinuation *M*ark *T*ransform.

+ Does not rely on any special support from the VM.

call/cc – CMT Approach

We offer an alternative: *C*ontinuation *M*ark *T*ransform.

+ Does not rely on any special support from the VM.
+ Does not change calling signature for functions.

call/cc – CMT Approach

We offer an alternative: *C*ontinuation *M*ark *T*ransform.

- \+ Does not rely on any special support from the VM.
- \+ Does not change calling signature for functions.
- \+ Permits conventional use of the stack and so does not interfere with standard tools or optimizations.

call/cc – CMT Approach

We offer an alternative: *C*ontinuation *M*ark *T*ransform.

+ Does not rely on any special support from the VM.
+ Does not change calling signature for functions.
+ Permits conventional use of the stack and so does not interfere with standard tools or optimizations.
+ Transformation can be applied locally without disrupting “most” function calls.

Let’s Translate

$$\textbf{(define}\ (f\ l)$$
$$\text{(case}\ l$$
$$\text{(cons}\ a\ l') \Rightarrow \text{(cons}\ (g\ a)\ (f\ l'))$$
$$\text{(nil)} \Rightarrow \text{(nil)))}$$

A-Normalize

```
(define (f l)
  (case l
   (cons a l') ⇒ (let (x (g a))
                   (let (l'' (f l'))
                    (cons x l'')))
   (nil) ⇒ (nil)))
```

A-Normalize

(**define** (f l)
 (case l
  (cons a l') $\Rightarrow$ (**let** (x (g a))
   (**let** (l'' (f l'))
    (cons x l'')))
  (nil) $\Rightarrow$ (nil)))

- A-Normal form names each intermediate value.

A-Normalize

```
(define (f l)
  (case l
    (cons a l') ⇒ (let (x (g a))
                    (let (l'' (f l'))
                      (cons x l'')))
    (nil) ⇒ (nil)))
```

- A-Normal form names each intermediate value.
- What we really want is the continuation of each intermediate computation.

Eliminate Let

```
(define (f l)
  (case l
    (cons a l') ⇒ ((λ (x)
                      ((λ (l'') (cons x l''))
                       (f l')))
                    (g a))
    (nil) ⇒ (nil)))
```

Eliminate Let

(**define** (f l)
  (case l
   (cons a l') $\Rightarrow$ ((λ (x)
                 ((λ (l'') (cons x l''))
                  (f l')))
                (g a))
   (nil) $\Rightarrow$ (nil)))

- Now each fragment of the continuation is explicitly represented as a lambda expression.

Evaluation Context

$$
\begin{array}{l}
((\lambda\ (l'')\ (\text{cons}\ B_0\ l''))\\
\quad \ldots\\
\qquad\quad ((\lambda\ (l'')\ (\text{cons}\ B_{n-1}\ l''))\\
\qquad\qquad ((\lambda\ (\mathit{x})\\
\qquad\qquad\quad ((\lambda\ (l'')\ (\text{cons}\ \mathit{x}\ l''))\\
\qquad\qquad\qquad (\mathit{f}\ L_n)))\\
\qquad\qquad [\ ]))\ \cdots)
\end{array}
$$

Evaluation Context

$$((\lambda\ (l'')\ (\text{cons}\ B_0\ l''))$$
$$\ldots$$
$$((\lambda\ (l'')\ (\text{cons}\ B_{n-1}\ l''))$$
$$((\lambda\ (x)$$
$$((\lambda\ (l'')\ (\text{cons}\ x\ l''))$$
$$(f\ L_n)))$$
$$[\ ]))\cdots)$$

- What do the evaluation contexts look like?

Evaluation Context

$((\lambda\ (l'')\ (\text{cons}\ B0\ l''))$

$\ldots$

$((\lambda\ (l'')\ (\text{cons}\ Bn\text{-}1\ l''))$

$((\lambda\ (x)$

$((\lambda\ (l'')\ (\text{cons}\ x\ l''))$

$(f\ L_n)))$

$[\ ]))\ \cdots)$

Evaluation Context

$((\lambda\ (l'')\ (\text{cons}\ B0\ l''))$

$\ldots$

$((\lambda\ (l'')\ (\text{cons}\ Bn\text{-}1\ l''))$

$((\lambda\ (x)$

$((\lambda\ (l'')\ (\text{cons}\ x\ l''))$

$(f\ L_n)))$

$[\ ]))\cdots)$

- $\mathcal{E} = [\ ] \mid ((\lambda \cdots)\ \mathcal{E})$

Evaluation Context

$((\lambda\ (l'')\ (\text{cons}\ B\text{0}\ l''))$

$\ldots$

$((\lambda\ (l'')\ (\text{cons}\ B\textit{n-1}\ l''))$

$((\lambda\ (x)$

$((\lambda\ (l'')\ (\text{cons}\ x\ l''))$

$(f\ L_n)))$

$[\ ]))\cdots)$

- $\mathcal{E} = [\ ] \mid ((\lambda \cdots)\ \mathcal{E})$
- Evaluation contexts are just chains of lambda applications.

Continuation Marks and CMT

Continuation Marks and CMT

- In the model langauge evaluation contexts are completely determined as sequences of lambda expressions.

Continuation Marks Basics

Continuation marks allow you to store extra information in the continuation of an expression and possibly retrieve it later.

Continuation Marks Basics

Continuation marks allow you to store extra information in the continuation of an expression and possibly retrieve it later.

- Values are embedded in the continuation using w-c-m

Continuation Marks Basics

Continuation marks allow you to store extra information in the continuation of an expression and possibly retrieve it later.

- Values are embedded in the continuation using w-c-m
- All such values embedded in the current continuation are recovered using c-c-m

Continuation Marks Example

```
(define (f l)
  (case l
    (nil) ⇒ (nil)
    (cons x l') ⇒ (cons (g x) (f l'))))
```

Continuation Marks Example

```
(define (f l)
  (case l
    (nil) ⇒ (begin
              (display (c-c-m))
              (nil))
    (cons x l') ⇒ (w-c-m x
                     (cons (g x) (f l')))))
```

Continuation Marks Example

```
(define (f l)
  (case l
    (nil) ⇒ (begin
              (display (c-c-m))
              (nil))
    (cons x l') ⇒ (w-c-m x
                    (cons (g x) (f l')))))

(f (cons 0 (cons 1 (cons 2 (nil)))))
```

Continuation Marks Example

```
(define (f l)
  (case l
    (nil) ⇒ (begin
               (display (c-c-m))
               (nil))
    (cons x l') ⇒ (w-c-m x
                     (cons (g x) (f l')))))
```

(*f* (cons 0 (cons 1 (cons 2 (nil)))))

$\xrightarrow{\text{eval}^*}$

(cons (*g* 0) (cons (*g* 1) (cons (*g* 2) (nil))))

Console Output: (*list* 0 1 2 3)

Where the Marks Go

- Recall that evaluation contexts are just appplications of lambdas

Where the Marks Go

- Recall that evaluation contexts are just appplications of lambdas

$$((\lambda\ (x_0)\ \cdots)$$
$$\quad((\lambda\ (x_1)\ \cdots)$$
$$\quad\ \cdots))$$

Where the Marks Go

- Recall that evaluation contexts are just appplications of lambdas

$$((\lambda\ (x_0)\ \cdots)$$
$$((\lambda\ (x_1)\ \cdots)$$
$$\cdots))$$

$$(\text{w-c-m}\ (\lambda\ \ (x0)\ \cdots\ )$$
$$((\lambda\ (x_0)\ \cdots)$$
$$(\text{w-c-m}\ (\lambda\ (x1)\ \cdots\ )$$
$$((\lambda\ (x_1)\ \cdots)$$
$$\cdots))))$$

Where the Marks Go

- Recall that evaluation contexts are just appplications of lambdas

$((\lambda\ (x_0)\ \cdots)$
$\quad((\lambda\ (x_1)\ \cdots)$
$\quad\ \cdots))$

(w-c-m $(\lambda\ \ (x0)\ \cdots\ )$
$\ ((\lambda\ (x_0)\ \cdots)$
$\quad$ (w-c-m $(\lambda\ (x1)\ \cdots\ )$
$\qquad((\lambda\ (x_1)\ \cdots)$
$\qquad\ \cdots))))$

- $\mathcal{E} = [\ ] \mid$ (w-c-m $(\lambda\ \cdots)\ ((\lambda\ \cdots)\ \mathcal{E}))$

Recovering the Marks

c-c-m can then be used to tease out the continuation.

Recovering the Marks

c-c-m can then be used to tease out the continuation.

```
(w-c-m (λ (x₀) ···)
  ((λ (x₀) ···)
   (w-c-m (λ (x₁) ···)
     ((λ (x₁) ···)
      (λ (m)
        (λ (v)
          (abort (resume m v))))
      (c-c-m))))))
```

Recovering the Marks

c-c-m can then be used to tease out the continuation.

```
(w-c-m (λ (x₀) ···)
  ((λ (x₀) ···)
   (w-c-m (λ (x₁) ···)
     ((λ (x₁) ···)
      (λ (m)
        (λ (v)
          (abort (resume m v))))
       (c-c-m))))))
```

$\xrightarrow{\text{eval}^*}$

(λ (*v*)
  (abort (*resume* (*list* (λ (x_0) ···) (λ (x_1) ···)) *v*)))

Resume

We need a helper function to reconstitute the stack using the results of a `c-c-m`

Resume

We need a helper function to reconstitute the stack using the results of a c-c-m

(**define** (*resume* l v)
  (case *l*
   (nil) $\Rightarrow v$
   (cons f l') $\Rightarrow$ (f (w-c-m f (*resume* l' v)))))

Reconstitution Theorem

$$\mathcal{CMT}[\![\Sigma]\!]/(\textit{resume}\ \mathcal{X}(\mathcal{CMT}[\![\mathcal{E}']\!])\ \mathcal{CMT}[\![v]\!])$$

$$\rightarrow^{+}_{\mathsf{TL}}\ \mathcal{CMT}[\![\Sigma]\!]/\mathcal{CMT}[\![\mathcal{E}']\!][\mathcal{CMT}[\![v]\!]]$$

Reconstitution Theorem

$$\mathcal{CMT}[\![\Sigma]\!]/(\textit{resume}\ \mathcal{X}(\mathcal{CMT}[\![\mathcal{E}']\!])\ \mathcal{CMT}[\![v]\!])$$
$$\rightarrow^{+}_{\text{TL}}\ \mathcal{CMT}[\![\Sigma]\!]/\mathcal{CMT}[\![\mathcal{E}']\!][\mathcal{CMT}[\![v]\!]]$$

- *resume* **faithfully** reconstitutes the stack

Putting It All Together

$((\lambda\ (x_0)\ \cdots)$

$\ldots$

$((\lambda\ (x_n)\ \cdots)\ [\ ]))$

Putting It All Together

$((\lambda\ (x_0)\ \cdots)$

$\cdots$

$((\lambda\ (x_n)\ \cdots)\ [\ ]))$

$\xrightarrow{\mathcal{C}MT}$

$(\lambda\ (\mathsf{v})$

(abort (*resume* (*list* $(\lambda\ (x_0)\ \cdots)\ \cdots\ (\lambda\ (x_n)\ \cdots)$) v)))

Putting It All Together

$((\lambda\ (x_0)\ \cdots)$

$\cdots$

$((\lambda\ (x_n)\ \cdots)$ [ ]))

$\xrightarrow{\mathcal{C}MT}$

(λ (v)

(abort (*resume* (*list* $(\lambda\ (x_0)\ \cdots)\ \cdots\ (\lambda\ (x_n)\ \cdots)$) v)))

((λ (*v*)

(abort (*resume* (*list* $(\lambda\ (x_0)\ \cdots)\ \cdots\ (\lambda\ (x_n)\ \cdots)$) *v*)))

7)

Putting It All Together

$((\lambda\ (x_0)\ \cdots)$

$\ldots$

$((\lambda\ (x_n)\ \cdots)\ [\ ]))$

$\xrightarrow{\mathcal{C}MT}$

$(\lambda\ (\mathsf{v})$

$(\text{abort}\ (\textit{resume}\ (\textit{list}\ (\lambda\ (x_0)\ \cdots)\ \cdots\ (\lambda\ (x_n)\ \cdots))\ \mathsf{v})))$

$((\lambda\ (v)$

$(\text{abort}\ (\textit{resume}\ (\textit{list}\ (\lambda\ (x_0)\ \cdots)\ \cdots\ (\lambda\ (x_n)\ \cdots))\ v)))$

$7)$

$\xrightarrow{\text{eval}^*}$

$((\lambda\ (x_0)\ \cdots)$

$\ldots$

$((\lambda\ (x_n)\ \cdots)\ [7]))$

Evaluation Theorem

$$\mathcal{CMT}[\![\textit{eval}_{\mathsf{SL}}(p)]\!] = \textit{eval}_{\mathsf{TL}}(\mathcal{CMT}[\![p]\!])$$

Evaluation Theorem

$$\mathcal{CMT}[\![\textit{eval}_{\mathsf{SL}}(p)]\!] = \textit{eval}_{\mathsf{TL}}(\mathcal{CMT}[\![p]\!])$$

$$\begin{array}{ccc}
\mathcal{E}[e] & \xrightarrow{\text{eval}} & \mathcal{E}'[e'] \\
| & & | \\
\mathcal{CMT} & & \mathcal{CMT} \\
\downarrow & & \downarrow \\
[\![\mathcal{E}[e]]\!] & \xrightarrow{\text{eval}^{+}} & [\![\mathcal{E}'[e']]\!]
\end{array}$$

Defunctionalization

Defunctionalization

- Still need to make continuations *serializable*.

Defunctionalization

- Still need to make continuations *serializable*.
- Continuation values are now just lists of lambdas.

Defunctionalization

- Still need to make continuations *serializable*.
- Continuation values are now just lists of lambdas.
- Use standard defunctionalization to replace lambda constructed values with serializable data structures.

Pragmatics

Recall that we wanted to avoid whole-program transformation

Pragmatics

Recall that we wanted to avoid whole-program transformation

- CMT does not change the calling signature of functions, so it can be applied locally.

Pragmatics

Recall that we wanted to avoid whole-program transformation

- CMT does not change the calling signature of functions, so it can be applied locally.
- The translated code will work fine in *most* contexts.

Pragmatics

Recall that we wanted to avoid whole-program transformation

- CMT does not change the calling signature of functions, so it can be applied locally.
- The translated code will work fine in *most* contexts.
- There's a problem when an *un*translated function calls a translated function that then attempts to capture a continuation.

Problem Details

Problem Details

- In our model language stack frames *are* lambda applications.

Problem Details

- In our model language stack frames *are* lambda applications.
- Translated frames are explicitly marked with their lambdas, while *un*translated frames are not marked.

Problem Details

- In our model language stack frames *are* lambda applications.
- Translated frames are explicitly marked with their lambdas, while *un*translated frames are not marked.
- If a continuation capture is attempted while the stack contains unmarked frames, then the resulting continuation value will have bits missing.

Problem Details

- In our model language stack frames *are* lambda applications.
- Translated frames are explicitly marked with their lambdas, while *un*translated frames are not marked.
- If a continuation capture is attempted while the stack contains unmarked frames, then the resulting continuation value will have bits missing.
- This will cause undefined behavior.

Solution Strategy

Use continuation marks to delimit untranslated portions of the stack.

Solution Strategy

Use continuation marks to delimit untranslated portions of the stack.

- Every function application is given a special “safety” mark.

Solution Strategy

Use continuation marks to delimit untranslated portions of the stack.

- Every function application is given a special “safety” mark.
- Whenever a continuation is captured inspect the list of safety marks.

Solution Strategy

Use continuation marks to delimit untranslated portions of the stack.

- Every function application is given a special “safety” mark.
- Whenever a continuation is captured inspect the list of safety marks.
- Undefined behavior is avoided by signalling an error.

Solution Strategy

Use continuation marks to delimit untranslated portions of the stack.

- Every function application is given a special “safety” mark.
- Whenever a continuation is captured inspect the list of safety marks.
- Undefined behavior is avoided by signalling an error.
- Need to take a closer look at Continuation marks.

Continuation Marks and Tail-Calls

A Continuation Mark that is in Tail Position with respect to an enclosing Continuation Mark will overwrite the value of the enclosing Continuation Mark.

Continuation Marks and Tail-Calls

A Continuation Mark that is in Tail Position with respect to an enclosing Continuation Mark will overwrite the value of the enclosing Continuation Mark.

- (w-c-m 7 (w-c-m 8 (f $\cdots$)))

Continuation Marks and Tail-Calls

A Continuation Mark that is in Tail Position with respect to an enclosing Continuation Mark will overwrite the value of the enclosing Continuation Mark.

- (w-c-m 7 (w-c-m 8 (f $\cdots$)))
- (w-c-m 8 (f $\cdots$)) is in Tail Position w.r.t. (w-c-m 7 $\cdots$)

Continuation Marks and Tail-Calls

A Continuation Mark that is in Tail Position with respect to an enclosing Continuation Mark will overwrite the value of the enclosing Continuation Mark.

- (w-c-m 7 (w-c-m 8 (f $\cdots$)))
- (w-c-m 8 (f $\cdots$)) is in Tail Position w.r.t. (w-c-m 7 $\cdots$)
- (w-c-m 7 $\cdots$) encloses (w-c-m 8 (f $\cdots$))

Continuation Marks and Tail-Calls

A Continuation Mark that is in Tail Position with respect to an enclosing Continuation Mark will overwrite the value of the enclosing Continuation Mark.

- (w-c-m 7 (w-c-m 8 (f $\cdots$)))
- (w-c-m 8 (f $\cdots$)) is in Tail Position w.r.t. (w-c-m 7 $\cdots$)
- (w-c-m 7 $\cdots$) encloses (w-c-m 8 (f $\cdots$))
- Simplifies to: (w-c-m 8 (f $\cdots$))

Example with Tail Calls

(**define** (*f-cps* *k* l)
  (case l
    (nil) $\Rightarrow$ (*k* (nil))
    (cons x l') $\Rightarrow$ (*f-cps* (λ (l'') (*g* (λ (x') (*k* (cons x' l'')) x)))
                                         l')))

Example with Tail Calls

```
(define (f-cps k l)
  (case l
    (nil) ⇒ (begin
               (display (c-c-m))
               (k (nil)))
    (cons x l′) ⇒ (w-c-m x
                      (f-cps (λ (l″) (g (λ (x′) (k (cons x′ l″))) x))
                        l′))))
```

Example with Tail Calls

```
(define (f-cps k l)
  (case l
    (nil) ⇒ (begin
              (display (c-c-m))
              (k (nil)))
    (cons x l′) ⇒ (w-c-m x
                     (f-cps (λ (l″) (g (λ (x′) (k (cons x′ l″))) x))
                        l′))))
```

```
(f (λ (x) x) (cons 0 (cons 1 (cons 2 (nil)))))
```

Example with Tail Calls

(**define** (*f-cps* *k* l)
  (case l
    (nil) $\Rightarrow$ (**begin**
                (*display* (c-c-m))
                (*k* (nil)))
    (cons x l') $\Rightarrow$ (w-c-m x
                (*f-cps* (λ (l'') (*g* (λ (x') (*k* (cons x' l''))) x))
                    l'))))

(*f* (λ (*x*) *x*) (cons 0 (cons 1 (cons 2 (nil)))))

$\xrightarrow{\text{eval}^*}$

(cons (*g* 0) (cons (*g* 1) (cons (*g* 2) (nil))))

Console Output: (*list* 3)

Continuation Marks – Extended Interface

The interface for Continuation Marks can be extended to allow for multiple disjoint sets of Continuation Marks.

Continuation Marks – Extended Interface

The interface for Continuation Marks can be extended to allow for multiple disjoint sets of Continuation Marks.

- w-c-m accepts an additional value that acts as a key identifying to which set the mark belongs.

Continuation Marks – Extended Interface

The interface for Continuation Marks can be extended to allow for multiple disjoint sets of Continuation Marks.

- w-c-m accepts an additional value that acts as a key identifying to which set the mark belongs.
- c-c-m accepts a key argument identifying which set of marks to recover.

Extended Interface Example

```
(w-c-m "Fred" 0
  (f_0 (w-c-m "Barney" 1
          (f_1 (w-c-m "Fred" 2
                  (f_2 (w-c-m "Barney" 3
                          (begin
                            (printf "Fred: ~a~n" (c-c-m "Fred"))
                            (printf "Barney: ~a~n" (c-c-m "Barney"))
                            19))))))))
```

Extended Interface Example

(w-c-m "Fred" 0
 (f_0 (w-c-m "Barney" 1
  (f_1 (w-c-m "Fred" 2
   (f_2 (w-c-m "Barney" 3
    (**begin**
     (*printf* "Fred: *~a~n*" (c-c-m "Fred"))
     (*printf* "Barney: *~a~n*" (c-c-m "Barney"))
     19))))))))

$\xrightarrow{\text{eval}^*}$

(f_0 (f_1 (f_2 19)))

Console Output:

Fred: (*list* 0 2)

Barney: (*list* 1 3)

Example – *map*

(**define** (*map f* l)
  (case l
    (cons *x* l') $\Rightarrow$ (cons (*f* *x*) (*map f* l'))
    (nil) $\Rightarrow$ (nil)))

Example – *map*

(**define** (*map f* l)
(case l
(cons *x* l') $\Rightarrow$ (cons (*f x*) (*map f* l'))
(nil) $\Rightarrow$ (nil)))

$\cdots$ (*map f l*) $\cdots$

Example – *map*

(**define** (*map* *f* l)
  (case l
    (cons *x* l') $\Rightarrow$ (cons (*f* *x*) (*map* *f* l'))
    (nil) $\Rightarrow$ (nil)))

$\cdots$ (*map f l*) $\cdots$

$\cdots$ (w-c-m “safe” false (*map f l*)) $\cdots$

Example – *map*

$$(\textbf{define}\ (\textit{map}\ f\ l)$$
$$(\text{case}\ l$$
$$(\text{cons}\ x\ l') \Rightarrow (\text{cons}\ (f\ x)\ (\textit{map}\ f\ l'))$$
$$(\text{nil}) \Rightarrow (\text{nil})))$$

$$\cdots (\textit{map}\ f\ l) \cdots$$

$$\cdots (\text{w-c-m “safe” false}\ (\textit{map}\ f\ l)) \cdots$$

$$\xrightarrow{\text{eval}^*}$$

$$\cdots (\text{w-c-m “safe” } \color{magenta}{\text{false}}\ \text{(cons (f x)} \cdots)) \cdots$$

Example – *safe-map*

(**define** (*safe-map f l*)
 (w-c-m "safe" true
  · · ·))

Example – *safe-map*

(**define** (*safe-map f l*)
 (w-c-m "safe" true
  ⋯))

⋯ (*safe-map f l*) ⋯

Example – *safe-map*

(**define** (*safe-map f l*)
(w-c-m “safe” true
· · ·))

· · · (*safe-map f l*) · · ·

· · · (w-c-m “safe” false (*safe-map f l*)) · · ·

Example – *safe-map*

(**define** (*safe-map f l*)
 (w-c-m “safe” true
  · · ·))

· · · (*safe-map f l*) · · ·

· · · (w-c-m “safe” false (*safe-map f l*)) · · ·

$\xrightarrow{\text{eval}^*}$

· · · (w-c-m “safe” false (w-c-m “safe” true · · ·)) · · ·

Example – *safe-map*

(**define** (*safe-map f l*)
 (w-c-m "safe" true
  ⋯))

⋯ (*safe-map f l*) ⋯

⋯ (w-c-m "safe" false (*safe-map f l*)) ⋯

$\xrightarrow{\text{eval}^*}$

⋯ (w-c-m "safe" false (w-c-m "safe" true ⋯)) ⋯

$\xrightarrow{\text{eval}^*}$

⋯ (w-c-m "safe" true ⋯) ⋯

Partial Solution

Continuation Marks allow us to detect when a continuation capture could lead to undefined behavior and instead signal an error.

Partial Solution

Continuation Marks allow us to detect when a continuation capture could lead to undefined behavior and instead signal an error.

- Translation can be applied locally.

Partial Solution

Continuation Marks allow us to detect when a continuation capture could lead to undefined behavior and instead signal an error.

- Translation can be applied locally.
- Special cases will lead lead to a run-time error.

Partial Solution

Continuation Marks allow us to detect when a continuation capture could lead to undefined behavior and instead signal an error.

- Translation can be applied locally.
- Special cases will lead lead to a run-time error.
- Some higher-order functions will have to be translated.

Partial Solution

Continuation Marks allow us to detect when a continuation capture could lead to undefined behavior and instead signal an error.

- Translation can be applied locally.
- Special cases will lead lead to a run-time error.
- Some higher-order functions will have to be translated.

Exceptions as Continuation Marks

Our translation offers an alternative to CPS for implementing first-class continuations on traditional VMs.

Exceptions as Continuation Marks

Our translation offers an alternative to CPS for implementing first-class continuations on traditional VMs.

- Conventional VMs don’t implement continuation marks but do implement exceptions.

Exceptions as Continuation Marks

Our translation offers an alternative to CPS for implementing first-class continuations on traditional VMs.

- Conventional VMs don’t implement continuation marks but do implement exceptions.
- We show that exceptions can simulate continuation marks.

Exceptions as Continuation Marks

Our translation offers an alternative to CPS for implementing first-class continuations on traditional VMs.

- Conventional VMs don't implement continuation marks but do implement exceptions.
- We show that exceptions can simulate continuation marks.
- (w-c-m) corresponds to intalling an exception handler.

Exceptions as Continuation Marks

Our translation offers an alternative to CPS for implementing first-class continuations on traditional VMs.

- Conventional VMs don't implement continuation marks but do implement exceptions.
- We show that exceptions can simulate continuation marks.
- (w-c-m) corresponds to intalling an exception handler.
- (c-c-m) corresponds to throwing an exception.

Exceptions as Continuation Marks

$\cdots$ (c-c-m) $\cdots$

Exceptions as Continuation Marks

$\cdots$ (c-c-m) $\cdots$

$\longmapsto$

$\cdots$ (w-c-m “safe” false (w-c-m “safe” true $\cdots$)) $\cdots$

Exceptions as Continuation Marks

$\cdots$ (c-c-m) $\cdots$

$\longmapsto$

$\cdots$ (w-c-m “safe” false (w-c-m “safe” true $\cdots$)) $\cdots$

(w-c-m (λ $\cdots$)
 ((λ $\cdots$) $\cdots$))

Exceptions as Continuation Marks

$\cdots$ (c-c-m) $\cdots$

$\longmapsto$

$\cdots$ (w-c-m “safe” false (w-c-m “safe” true $\cdots$)) $\cdots$

(w-c-m (λ $\cdots$)
  ((λ $\cdots$) $\cdots$))

$\longmapsto$

(try ((lambda $\cdots$) $\cdots$) (catch exn (throw (cons (lambda

Exceptions as Continuation Marks

$\cdots$ (c-c-m) $\cdots$

$\longmapsto$

$\cdots$ (w-c-m “safe” false (w-c-m “safe” true $\cdots$)) $\cdots$

(w-c-m (λ $\cdots$)
 ((λ $\cdots$) $\cdots$))

$\longmapsto$

(try ((lambda $\cdots$) $\cdots$) (catch exn (throw (cons (lambda