Automatically RESTful Web Applications

Marking Modular Serializable Continuations

Jay McCarthy
Brigham Young University
jay@cs.byu.edu

Abstract

Continuation-based Web servers provide distinct advantages over traditional Web application development: expressive power and modularity. This power leads to fewer errors and more interesting applications. Furthermore, these Web servers are more than prototypes; they are used in some real commercial applications. Unfortunately, they pay a heavy price for the additional power in the form of lack of scalability.

We fix this key problem with a modular program transformation that produces scalable, continuation-based Web programs based on the REST architecture. Our programs use the same features as non-scalable, continuation-based Web programs, so we do not sacrifice expressive power for performance. In particular, we allow continuation marks in Web programs. Our system uses 10 percent (or less) of the memory required by previous approaches.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Control structures

General Terms Languages, Performance, Theory

Keywords Continuations, Stack Inspection, Web Applications

1. Introduction

The functional programming community has inspired Web application developers with the insight that Web interaction corresponds to continuation invocation (Hughes 2000; Queinnec 2000; Graham 2001). This insight helps to *explain* what Web applications do and when ad hoc continuation capture patterns are erroneous (Krishnamurthi et al. 2007). This understanding leads to more correct Web applications.

Many continuation-based Web development frameworks try to apply this insight directly by automatically capturing continuations for Web application authors. These frameworks are often frustrated because they limit modularity, are not scalable, or only achieve correct continuation capture without more expressive power.

Whole-program compilation systems are unusable by real-world developers because they sacrifice modularity and interaction with third-party libraries for performance and formal elegance (Matthews et al. 2004; Cooper et al. 2006). Modular compilation systems are unattractive to real world developers when they do not add expressive power (Pettyjohn et al. 2005; Thiemann 2006). Continuation-based Web servers are unusable by real world developers, though they add more expressive power, because they are inherently not scalable (Ducasse et al. 2004; Krishnamurthi et al. 2007).

This paper presents a modular program transformation that produces scalable Web applications and offers more expressive power than existing modular systems. Web applications written using our system can offload *all* state to clients—the gold standard of scalability—or, if necessary, keep state on the server and use *ten times* less memory.

2. Background

The largest problem Web developers solve is imposed by the statelessness of HTTP: when a server responds to a client's request, the connection is closed and the Web program on the server exits. When the client makes a subsequent request, the request delivered to the server must contain enough information to resume the computation. The insight of functional programmers is that this information *is* the continuation.

Traditional Web programmers call this representational state transfer (REST) (Fielding and Taylor 2002).[1] It is naturally scalable due to the lack of per-session state. Session state is poison to scalability because each session has an absolute claim on server resources. There is no sound way to reclaim space since dormant sessions may reactivate at any time. This is *clearly* inefficient. Consequently, unsafe and ad hoc resource policies, like timeouts, are used to restore some scalability. The scalability of REST also comes at a price in the form of programming difficulty. We will demonstrate this by porting a calculator to the Web.[2]

```
(define (calc)
  (display (+ (prompt "First:") (prompt "Second:"))))
```

We must break this single coherent function into three different functions to create a Web application. Each function represents a distinct part of the control-flow of the application: entering the first number, entering the second, and displaying their sum.

```
(define (calc) (web-prompt "First:" 'get-2nd #f))
(define (get-2nd first) (web-prompt "Second:" 'sum first))
(define (sum first second) (display/web (+ second first)))
```


ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.

[1] Unfortunately, modern Web programmers have forgotten this definition of REST. They use the acronym REST to refer to a resource-based URL structure where database operations (like create, replace, update, and delete) are mapped to suggestive combinations of URLs and HTTP request types (like PUT, POST, GET, and DELETE). We will *not* use REST in this way.

[2] All program examples are written in PLT Scheme.

```
(define (calc)
  ;; new-session allocates server-side storage
  (web-prompt "First:" 'get-2nd (new-session)))

(define (get-2nd session-id first)
  ;; session-set! modifies server-side storage
  (session-set! session-id 'first first)
  (web-prompt "Second:" 'sum session-id))

(define (sum session-id second)
  ;; session-lookup references server-side storage
  (define first (session-lookup session-id 'first))
  (display/web (+ second first)))
```

Figure 1. REST Without All the REST

The continuation is encoded by the developer in the second argument of *web-prompt* and the free variables of the continuation in the third argument.

Unfortunately, it is tiresome in most Web programming environments to marshal all data to the client and back but convenient to access the server-side store (through session objects and databases), so developers use this naturally RESTful style in an entirely non-RESTful way. (See Figure 1.)

The Web's RESTful style is a form of continuation-passing style (CPS) (Fischer 1972). There are well-known transformations from *direct style* code into CPS that allow Web applications to be written in the natural way but converted into the scalable style by the server before execution.

Matthews et al. (2004) gave an automatic translation from direct style programs into traditional Web applications. This tool performs a CPS transformation, λ-lifting, defunctionalization, and a store-passing style transformation (to capture the store as a cookie value). These direct style Web applications are entirely RESTful because the lexical environment and store are transferred to the user between interactions.

Unfortunately, the CPS transformation is not modular; the entire code-base, *including libraries*, must be transformed. Thus, this technique is not feasible in applications that rely on unmodifiable libraries or separate compilation.

The PLT Web Server by Krishnamurthi et al. (2007) does not have this problem. It enables direct style Web applications written in PLT Scheme through first-class continuations. These implicit continuations avoid the CPS transformation and thereby provide modularity. However, the PLT implementation technique sacrifices the REST architecture.

Continuations (and the environments they close over) in PLT Scheme cannot be serialized into an external format or transferred to the client. Thus, the PLT Web Server stores continuations in the server's memory and provides the client with a unique identifier for each continuation. These continuations are per-session server state, and their unique identifiers are new GC roots. Because there is no *sound* way to reclaim these continuations, they must be retained indefinitely or *unsoundly* deleted.

The memory problems associated with this un-RESTful policy are well known. For example, a recent ICFP experience report (Welsh and Gurnell 2007) concurs with our experience managing the CONTINUE service (Krishnamurthi 2003) by reporting unreasonable memory usage. CONTINUE is a Web application for paper submissions, reviews, and PC meetings, so there is no intrinsic reason for this memory usage. We have experimented with a number of stopgap strategies, such as explicit continuation management through the primitive *send/forward* (Krishnamurthi et al. 2007) and a Least Recently Used (LRU) continuation management strategy. While useful remedies for some symptoms, they are not solutions. In contrast, the work presented herein reduces memory consumption *by ten times* for these same Web applications.

```
(define (fact n)
  (if (zero? n)
    (begin (display (c-c-m 'fact))
           1)
    (w-c-m 'fact n (* n (fact (sub1 n))))))
(fact 3)
↦
console output: (1 2 3)
computed value: 6

(define (fact-tr n a)
  (if (zero? n)
    (begin (display (c-c-m 'fact))
           a)
    (w-c-m 'fact n (fact-tr (sub1 n) (* n a)))))
(fact 3)
↦
console output: (1)
computed value: 6
```

Figure 2. Factorial with Continuation Marks

Despite its memory problems, the PLT Web Server provides a valuable Web application framework, in part because of its expressive features, like continuation marks.

Many programming languages and environments allow access to the runtime stack in one way or another. Examples include Java security through stack inspection, privileged access for debuggers in .NET, and exception handlers in many languages. An abstraction of all these mechanisms is provided by PLT Scheme in continuation marks (Clements et al. 2001). Using the **with-continuation-mark** (**w-c-m**) language form, a developer can attach values to the control stack. Later, the stack-walking primitive *current-continuation-marks* (*c-c-m*) can retrieve those values from the stack. Continuation marks are parameterized by keys and do not interfere with Scheme's tail-calling requirements. These two mechanisms allow marks to be used without interfering with existing code.

A pedagogic example of continuation mark usage is presented in Figure 2. *fact* is the factorial function with instrumentation using continuation marks: **w-c-m** records function arguments on the stack and *c-c-m* collects them in the base case. *fact-tr* is a tail-recursive version of factorial that appears to be an identical usage of continuation marks, but because they preserve tail-calling space usage, the intermediate marks are overwritten, leaving only the final mark.

Continuation marks are useful in all programs, but are particularly useful on the Web. If the control stack is isomorphic to the user's position in the application, then continuation marks can be used to record information about where the user has gone and is going.

For example, in CONTINUE we use a continuation mark to hold the identity of the user. The essence of this technique is shown in Figure 3. This mark is stored on login and retrieved inside *display-site* for tasks like paper rating. This is more convenient than threading the state throughout the application and allows a trivial implementation of user masquerading, so an administrator can debug a user's problems, and delegation, so a reviewer can assign a sub-reviewer limited powers.

Web application developers are torn between the REST architecture, direct style code, modularity, and expressive features, like

```
(define (start-server ireq)
  (w-c-m current-user (show-login) (display-site)))

(define (who-am-i)
  (first (c-c-m current-user)))

(define (delegate email paper)
  (w-c-m current-user (list 'delegate paper (who-am-i))
    (email-continuation-url email)
    (display-paper paper)))

(define (masquerade user)
  (if (symbol=? (who-am-i) 'admin)
      (w-c-m current-user user (display-site))
      (access-denied)))
```

Figure 3. Marks in Web Applications

continuation marks. In this paper, we present a modular program transformation that automatically produces RESTful versions of direct style Web programs that utilize continuation marks.

3. Transformation Intuition

Our modular RESTful transformation is based on one from Pettyjohn et al. (2005). Unfortunately, their transformation does not support continuation marks in the input language, so it is not sufficient for our purposes. Our transformation is structurally similar to theirs, so we review their transformation before turning to our contribution.

The Pettyjohn et al. (2005) transformation relies on the modular Administrative Normal Form (ANF) transformation (Flanagan et al. 2004) and stack inspection to simulate *call/cc*.

ANF is a canonical form that requires all function arguments to be named. This has the implication that the entire program is a set of nested **let** expressions with simple function calls for bodies. If the **let**s are expanded into λs, then the continuation of every expression is syntactically obvious. Any expression can be modularly transformed into ANF without modifying the rest of the program.

The main insight of Pettyjohn et al. (2005) was that *c-c-m* can "capture" the continuation, just like *call/cc*, if the components of the continuation are installed via **w-c-m**. Their transformation does this by duplicating the continuation into marks. This is easy because ANF makes these continuations obvious, and the tail-calling property of marks mirrors that of continuations themselves, so the two stay synchronized.

Their work is a testament to the power of continuation marks; we will review the essence of the transformation.

Function applications, like (*k a*), are transformed as

```
(k (w-c-m □ k a))
```

where □ is a special key known only to the transformation. This effectively duplicates the continuation in a special mark. Then (*call/cc e*) is transformed as

```
(e (let ([ks (c-c-m □)])
     (λ (x) (abort (resume ks x)))))
```

where *resume* restores the continuation record from the □ marks into an actual control stack. *resume* must also reinstall the □ marks so subsequent invocations of *call/cc* are correct.

```
(define (resume l x)
  (if (empty? l) x
      (let ([k (first l)] [l (rest l)])
        (k (w-c-m □ k (resume l x))))))
```

This transformation produces RESTful Web applications, because standard modular λ-lifting and defunctionalization transformations encode all values into serializable representations that can be sent to the client.

The *great irony* of the Pettyjohn et al. (2005) transformation is that, while it shows the immense power of continuation marks, it *does not* support continuation marks in the input language; it cannot be used for Web applications that use marks themselves. Furthermore, it is not trivial to add support for continuation marks in the transformation: a semantic insight is necessary—this is our formal contribution, in addition to the other practical extensions we have made.

3.1 Capturing Marks

The most intuitive strategy for supporting marks is to simply capture "all the continuation marks" whenever the marks that record the continuation are captured. However, this is not possible.

Continuation marks are parameterized by a key. When a developer uses *c-c-m*, she or he must provide a key—and *only marks associated with that key* are returned. If a mark is added with a unique, or unknowable, key, such as a random value or uninterned symbol (a "*gensym*"), then it cannot be extracted by the context it wraps. This is an essential feature of continuation marks: they are invisible to the uninitiated. Without this property, the behavior of an expression could drastically change with a change to its context: new results could mysteriously return from *c-c-m* without any explanation. This would have a *dynamic scope* level of grotesqueness to it.

We must *record* the continuation marks, as we record the continuation components, so they can be *extracted* when performing continuation capture. A simple strategy is to transform all instances of

(**w-c-m** *k v e*) into (**w-c-m** *k v* (**w-c-m** ◇ (*cons k v*) *e*))

where ◇ is a key known only to the transformation. It seems straightforward to adapt the transformation so *call/cc* captures and restores these marks as well

```
(e (let ([ks (c-c-m □)] [cms (c-c-m ◇)])
     (λ (x) (abort (re-mark cms (λ () (resume ks x)))))))
```

where *re-mark* is similar to *resume*:

```
(define (re-mark l e)
  (if (empty? l) (e)
      (let* ([cm (first l)] [l (rest l)]
             [m (car cm)] [v (cdr cm)])
        (w-c-m m v (w-c-m ◇ (cons m v) (re-mark l e))))))
```

While simple and elegant, these strategies are incorrect.

3.2 Reinstalling Marks

The first problem is that *resume* and *re-mark* do not interact correctly. Consider the following program:

```
(f (w-c-m k1 v1 (g (call/cc e))))
```

This is transformed into

```
(f (w-c-m □ f
     (w-c-m k1 v1 (w-c-m ◇ (cons k1 v1)
       (g (w-c-m □ g
            (e (let ([ks (c-c-m □)] [cms (c-c-m ◇)])
```

(λ (*x*) (**abort** (*re-mark cms*
(λ () (*resume ks x*))))))))))))

If *e* calls the continuation with a value, *x*, it reduces to

(**w-c-m** *k1 v1* (**w-c-m** ⋄ (*cons k1 v1*)
(*f* (**w-c-m** □ *f* (*g* (**w-c-m** □ *g x*))))))

The mark for *k1* is lifted outside of *f*.

The problem is that even though the □ and ⋄ marks are recorded in the same stack frames, they are collected and installed separately: the ⋄s are put before the □s. We can collect these together by extracting multiple keys at once.

We correct the transformation of (*call/cc e*) as

(*e* (**let** ([*k*∗*cms* (*c-c-m* □ ⋄)])
(λ (*x*) (**abort** (*resume k*∗*cms x*)))))

When *c-c-m* is given n arguments, the marks are returned as a list of *frames*, where each frame is a list of length n where (*list-ref l i*) is the value of associated with the i argument or #f if none exists. Naturally, *resume* must combine the previous *resume* and *re-mark* operations.

(**define** (*resume l x*)
(**if** (*empty? l*) *x*
(**let**∗ ([*M* (*car l*)] [*l* (*cdr l*)] [*k* (*car M*)] [*cm* (*cadr M*)])
(**cond**
[(**and** *k* (*not cm*))
(*k* (**w-c-m** □ *k* (*resume l x*)))]
[(**and** (*not k*) *cm*)
(**let** ([*m* (*car cm*)] [*v* (*cdr cm*)])
(**w-c-m** *m v* (**w-c-m** ⋄ *cm* (*resume l x*))))]
[**else**
(*resume*
(*list*∗ (*list k* #f) (*list* #f *cm*) *l*) *x*)])))))

Even though the marks are now in the correct order, there is still an error in our transformation.

3.3 The Algebra of Marks

Consider the transformation of the following program:

(**w-c-m** *k1 v1* (**w-c-m** *k2 v2 e*))

This is transformed as

(**w-c-m** *k1 v1* (**w-c-m** ⋄ (*cons k1 v1*)
(**w-c-m** *k2 v2* (**w-c-m** ⋄ (*cons k2 v2*) *e*))))

Because continuation marks respect the tail-calling properties of Scheme, if a frame already contains a mark for a key, the mark is overwritten. Thus, the following are equivalent:

(**w-c-m** k v (**w-c-m** k u *e*)) and (**w-c-m** k u *e*)

Similarly, marks with different keys share the same frame. Therefore, the following are equivalent:

(**w-c-m** x v (**w-c-m** y u *e*)) and (**w-c-m** y u (**w-c-m** x v *e*))

Thus, the transformation is equivalent to

(**w-c-m** *k1 v1* (**w-c-m** *k2 v2*
(**w-c-m** ⋄ (*cons k1 v1*) (**w-c-m** ⋄ (*cons k2 v2*) *e*))))

which is equivalent to

(**w-c-m** *k1 v1* (**w-c-m** *k2 v2*
(**w-c-m** ⋄ (*cons k2 v2*) *e*)))

$$
\begin{array}{rcl}
e & ::= & a \\
 & | & (\overline{w}\ e) \\
 & | & (\textbf{letrec}\ (\overline{[\sigma\ v]})\ e) \\
 & | & (\text{w-c-m}\ a\ a\ e) \\
 & | & (\text{c-c-m}\ [\overline{a}]) \\
 & | & (\text{match}\ w\ \overline{l}) \\
 & | & (\text{abort}\ e) \\
 & | & (\text{call/cc}\ w) \\
l & ::= & [(K\ \overline{x})e] \\
a & ::= & w \mid (K\ \overline{a}) \\
w & ::= & v \mid x \\
v & ::= & (\lambda(\overline{x})\ e) \mid (K\ \overline{v}) \mid \sigma \mid \kappa.\mathcal{E} \\
 & & \\
x & \in & \text{Variables} \\
\sigma & \in & \text{References} \\
 & & \text{where Variables} \cap \text{References} = \emptyset \\
 & & \\
\mathcal{E}_\Delta & ::= & (\text{w-c-m}\ v\ v'\ \mathcal{E}_{v,\Delta}) \text{ where } v \notin \Delta \\
 & | & [\,] \mid (\overline{v}\ \mathcal{E}) \\
\Sigma & ::= & \emptyset \mid \Sigma[\sigma \mapsto v]
\end{array}
$$

Figure 4. SL Syntax

We've lost the record of the *k1* mark in the ⋄ mark.

One solution is to maintain a map from keys to values in ⋄ marks and explicitly update that map with the continuation mark transformation. For example, we will transform

(**w-c-m** *k v e*) into (**w-c-m** *k v* (*c-w-i-c-m* ⋄ (λ (*cms*)
(**w-c-m** ⋄ (*map-set cms k v*) *e*)) *empty*))

where (*c-w-i-c-m key-v proc default-v*) (*c-w-i-c-m = call-with-immediate-continuation-mark*) calls *proc* with the value associated with *key-v* in the first frame of the current continuation. This is the value that would be replaced if this call were replaced with a call to **w-c-m**. If no such value exists in the first frame, *default-v* is passed to *proc*. The call to *proc* is in tail position. This function can be implemented using just **w-c-m** and *c-c-m* (Clements et al. 2008).

After changing *resume* to operate on mark sets, we have a correct transformation of continuation marks in the input language. The rest of the transformation does not need to change dramatically for the entire PLT Scheme language. Now RESTful Web applications can be written in direct style.

4. Formal Treatment

Armed with intuition, we present the continuation (mark) reconstruction transformation formally.

4.1 The Source Language

The language in Figure 4, dubbed SL for *s*ource *l*anguage, is a modified version of A-Normal form (ANF) (Flanagan et al. 2004). It uses λ instead of **let**. Furthermore, we allow applications of arbitrary length. The language is extended with *call/cc*, **abort**, **letrec**, algebraic datatypes, and continuation marks. This is different from the source language of Pettyjohn et al. (2005) by including continuation marks and **abort**, which were included in the target language there. Algebraic datatypes are essential to using marks; **abort** is included for consistency with the target language. $\overline{X}$ denotes zero or more occurrences of X.

Instances of algebraic datatypes are created with constructors (K, K^m) and destructured with **match**. We leave the actual set of constructors unspecified, though we assume it contains the standard list constructors *cons* and *nil*.

$$\Sigma/\mathcal{E}[((\lambda(\overline{x})\ e)\ \overline{v})] \xrightarrow{(1)}_{SL} \Sigma/\mathcal{E}[e\overline{[x \mapsto v]}]$$

$$\Sigma/\mathcal{E}[(\text{match } (K\ \overline{v})\ \overline{l})] \xrightarrow{(2)}_{SL} \Sigma/\mathcal{E}[e\overline{[x \mapsto v]}] \quad \text{where } [(K\ \overline{x})e] \in \overline{l} \text{ and is unique}$$

$$\Sigma/\mathcal{E}[(\textbf{letrec } (\overline{[\sigma\ v]})\ e)] \xrightarrow{(3)}_{SL} \Sigma\overline{[\sigma \mapsto v]}/\mathcal{E}[e]$$

$$\Sigma/\mathcal{E}[(\sigma\ \overline{v})] \xrightarrow{(4)}_{SL} \Sigma/\mathcal{E}[e\overline{[x \mapsto v]}] \quad \text{where } \Sigma(\sigma) = (\lambda(\overline{x})\ e)$$

$$\Sigma/\mathcal{E}[(\text{match } \sigma\ \overline{l})] \xrightarrow{(5)}_{SL} \Sigma/\mathcal{E}[(\text{match } \Sigma(\sigma)\ \overline{l})]$$

$$\Sigma/\mathcal{E}[(\text{w-c-m } v_k\ v_1\ \mathcal{E}'_{v_k}[(\text{w-c-m } v_k\ v_2\ e)])] \xrightarrow{(6)}_{SL} \Sigma/\mathcal{E}[(\text{w-c-m } v_k\ v_2\ \mathcal{E}'_{v_k}[e])] \quad \text{where } \mathcal{E}'_{v_k} \text{ contains only w-c-ms}$$

$$\Sigma/\mathcal{E}[(\text{w-c-m } v_k\ v_1\ v_2)] \xrightarrow{(7)}_{SL} \Sigma/\mathcal{E}[v_2]$$

$$\Sigma/\mathcal{E}[(\text{c-c-m } [\overline{v}])] \xrightarrow{(8)}_{SL} \Sigma/\mathcal{E}[\chi_{\overline{v}}(\mathcal{E}, (\text{nil}))]$$

$$\Sigma/\mathcal{E}[(\text{abort } e)] \xrightarrow{(9)}_{SL} \Sigma/e$$

$$\Sigma/\mathcal{E}[(\text{call/cc } v)] \xrightarrow{(\star)}_{SL} \Sigma/\mathcal{E}[(v\ \kappa.\mathcal{E})]$$

$$\Sigma/\mathcal{E}[(\kappa.\mathcal{E}'\ v)] \xrightarrow{(\sharp)}_{SL} \Sigma/\mathcal{E}'[v]$$

$$\chi_{vs}(\mathcal{E}) = \chi_{vs}(\mathcal{E}, (\text{nil}))$$

$$\chi_{vs}([], v_l) = v_l$$

$$\chi_{vs}((\overline{v}\ \mathcal{E}), v_l) = (\text{cons } v_l\ \chi_{vs}(\mathcal{E}, (\text{nil}))$$

$$\chi_{vs}((\text{w-c-m } v_k\ v_v\ \mathcal{E}), v_l) = \chi_{vs}(\mathcal{E}, (\text{cons } (\text{cons } v_k\ v_v)\ v_l))) \quad \text{if } v_k \in vs$$

$$\chi_{vs}((\text{w-c-m } v_k\ v_v\ \mathcal{E}), v_l) = \chi_{vs}(\mathcal{E}, v_l) \quad \text{otherwise}$$

Figure 5. SL Semantics

The operational semantics is specified via the rewriting system in Figure 5. It is heavily based on target language semantics of Pettyjohn et al. (2005). The first rule is the standard β_v-rewriting rule for call-by-value languages (Plotkin 1975). The second handles algebraic datatypes.

Rules 3, 4, and 5 specify the semantics for **letrec**. Bindings established by **letrec** are maintained in a global store, Σ. For simplicity, store references (σ) are distinct from identifiers bound in lambda expressions (Felleisen and Hieb 1992). Furthermore, to simplify the syntax for evaluation contexts, store references are treated as values, and dereferencing is performed when a store reference appears in an application (rule 4) or in a match expression (rule 5).

The next six rules implement the continuation-related operators. Recall that continuation marks allow for the manipulation of contexts. Intuitively, (**w-c-m** *k v e*) installs a mark for the key k associated with the value v into the continuation of the expression e, while (*c-c-m* $[\overline{v}]$) recovers a list of all marks for the keys in $\overline{v}$ embedded in the current continuation. To preserve proper tail-call semantics, if a rewriting step results in more than one **w-c-m** of the same key, surrounding the same expression, the outermost mark is replaced by the inner one. Similarly, marks for different keys are considered to be a single location.

The mark interleaving requirement is enforced by a grammar for evaluation contexts that consists of one parameterized nonterminal. The parameter of $\mathcal{E}$ (Δ) represents the keys that are *not* allowed to appear in the context. Thus, multiple adjacent **w-c-m** expressions (of the same key) must be treated as a redex. When such a redex is encountered, the redundant marks are removed, starting with the outermost (rule 6). Marks that surround a value are discarded after the evaluation of the subterm (rule 7). The evaluation of *c-c-m* employs the function χ to extract the marks of the keys from the evaluation context (rule 8). Marks are extracted in order, such that *c-c-m* evaluates to a list of lists of pairs of keys and their value, starting with the oldest.

The evaluation rules for continuations (rules $\star$ and $\sharp$) are standard; abort abandons the context (rule 9) to facilitate reimplementing continuations.

$$\begin{array}{rcl}
e & ::= & a \\
 & | & (\overline{w}\ e) \\
 & | & (\textbf{letrec } (\overline{[\sigma\ v]})\ e) \\
 & | & (\text{w-c-m } a\ a\ e) \\
 & | & (\text{c-c-m } [\overline{a}]) \\
 & | & (\text{match } w\ \overline{l}) \\
 & | & (\text{abort } e) \\
 & & \\
l & ::= & [(K\ \overline{x})e] \\
a & ::= & w \mid (K\ \overline{a}) \\
w & ::= & v \mid x \\
v & ::= & (\lambda(\overline{x})\ e) \mid (K\ \overline{v}) \mid \sigma \\
 & & \\
x & \in & \text{Variables} \\
\sigma & \in & \text{References} \\
 & & \text{where Variables} \cap \text{References} = \emptyset \\
 & & \\
\mathcal{E}_\Delta & ::= & (\text{w-c-m } v\ v'\ \mathcal{E}_{v,\Delta}) \text{ where } v \notin \Delta \\
 & | & [\,] \mid (\overline{v}\ \mathcal{E}) \\
\Sigma & ::= & \emptyset \mid \Sigma[\sigma \mapsto v]
\end{array}$$

Figure 6. TL Syntax

4.2 The Target Language

The *t*arget *l*anguage (TL) in Figure 6 is identical to SL, except that *call/cc* has been removed along with the continuation values associated with it. The semantics (Figure 7) is also identical, except for the removal of rules $\star$ and $\sharp$ for continuation capture and application.

4.3 Replacing Continuation Capture

Following Pettyjohn et al. (2005), we define our translation (Figure 9) from SL to TL as $\mathcal{CMT}$, for *c*ontinuation *m*ark *t*ransform. The translation decomposes a term into a context and a redex by the grammar in Figure 8.

We prove that the decomposition is unique and thus the translation is well-defined.

Lemma 1 (Unique Decomposition)**.**
Let $e \in SL$. Either $e \in a$ or $e = \underline{\mathcal{E}}[r]$ for some redex r.

The translation rules are straightforward, except for application, continuation capture, values and marks. Continuation values are transformed using a variation of the rule for *call/cc*. *call/cc* uses c-c-m to reify the context and applies *resume* to reconstruct the context after a value is supplied. **abort** is used in the continuation-as-closure to escape from the calling context, as the SL semantics does.

Continuations rely on the insertion of marks to capture the continuation as it is built. This strategy employs the property of ANF that every continuation is obvious, in that it is the value in the function position of function applications. The translation marks each application, using the $\square$ mark to record the continuation.

$$
\begin{array}{rcl}
\Sigma/\mathcal{E}[((\lambda(\overline{x})\ e)\ \overline{v})] & \xrightarrow{(1)}_{TL} & \Sigma/\mathcal{E}[e[\overline{x \mapsto v}]] \\
\Sigma/\mathcal{E}[(\text{match}\ (K\ \overline{v})\ \overline{l})] & \xrightarrow{(2)}_{TL} & \Sigma/\mathcal{E}[e[\overline{x \mapsto v}]] \\
 & \text{where} & [(K\ \overline{x})e] \in \overline{l} \\
 & & \text{and is unique} \\
\Sigma/\mathcal{E}[(\textbf{letrec}\ (\overline{[\sigma\ v]})\ e)] & \xrightarrow{(3)}_{TL} & \Sigma\overline{[\sigma \mapsto v]}/\mathcal{E}[e] \\
\Sigma/\mathcal{E}[(\sigma\ \overline{v})] & \xrightarrow{(4)}_{TL} & \Sigma/\mathcal{E}[e[\overline{x \mapsto v}]] \\
 & \text{where} & \Sigma(\sigma) = (\lambda(\overline{x})\ e) \\
\Sigma/\mathcal{E}[(\text{match}\ \sigma\ \overline{l})] & \xrightarrow{(5)}_{TL} & \Sigma/\mathcal{E}[(\text{match}\ \Sigma(\sigma)\ \overline{l})]
\end{array}
$$

$$
\begin{array}{rcl}
\Sigma/\mathcal{E}[(\text{w-c-m}\ v_k\ v_1 & & \Sigma/\mathcal{E}[(\text{w-c-m}\ v_k\ v_2 \\
\mathcal{E}'_{v_k}[(\text{w-c-m}\ v_k\ v_2\ e)])] & \xrightarrow{(6)}_{TL} & \mathcal{E}'_{v_k}[e])] \\
\multicolumn{3}{c}{\text{where } \mathcal{E}'_{v_k} \text{ contains only w-c-ms}}
\end{array}
$$

$$
\begin{array}{rcl}
\Sigma/\mathcal{E}[(\text{w-c-m}\ v_k\ v_1\ v_2)] & \xrightarrow{(7)}_{TL} & \Sigma/\mathcal{E}[v_2] \\
\Sigma/\mathcal{E}[(\text{c-c-m}\ [\overline{v}])] & \xrightarrow{(8)}_{TL} & \Sigma/\mathcal{E}[\chi_{\overline{v}}(\mathcal{E}, (\text{nil}))] \\
\Sigma/\mathcal{E}[(\text{abort}\ e)] & \xrightarrow{(9)}_{TL} & \Sigma/e
\end{array}
$$

$$
\begin{array}{rl}
\chi_{vs}(\mathcal{E}) = & \chi_{vs}(\mathcal{E}, (\text{nil})) \\
\chi_{vs}([], v_l) = & v_l \\
\chi_{vs}((\overline{v}\ \mathcal{E}), v_l) = & (\text{cons}\ v_l\ \chi_{vs}(\mathcal{E}, (\text{nil})) \\
\chi_{vs}((\text{w-c-m}\ v_k\ v_v\ \mathcal{E}), v_l) = & \chi_{vs}(\mathcal{E}, (\text{cons}\ (\text{cons}\ v_k\ v_v)\ v_l))) \\
 & \text{if } v_k \in vs \\
\chi_{vs}((\text{w-c-m}\ v_k\ v_v\ \mathcal{E}), v_l) = & \chi_{vs}(\mathcal{E}, v_l) \\
 & \text{otherwise}
\end{array}
$$

Figure 7. TL Semantics

$$
\begin{array}{rcl}
r & ::= & (\overline{w}) \\
 & | & (\textbf{letrec}\ (\overline{[\sigma\ w]})\ e) \\
 & | & (\text{w-c-m}\ a\ a\ w) \\
 & | & (\text{c-c-m}\ [\overline{a}]) \\
 & | & (\text{match}\ w\ \overline{l}) \\
 & | & (\text{abort}\ e) \\
 & | & (\text{call/cc}\ e) \\
\underline{\mathcal{E}_\Delta} & ::= & (\text{w-c-m}\ v\ v'\ \underline{\mathcal{E}_{v,\Delta}}) \text{ where } v \notin \Delta \\
 & | & [\,] \mid (\overline{v}\ \underline{\mathcal{E}})
\end{array}
$$

Figure 8. Translation Decompositions

Similarly, all continuation marks are recorded with the $\diamond$ mark. Later, these marks will be collected by *c-c-m* and used to reproduce the context.

The *resume* function (Figure 10) is used by the translated program. *resume* faithfully reconstructs an evaluation context from a list of pairs of continuation functions and mark sets. It traverses the list and recursively applies the functions from the list and reinstalls the marks using *restore-marks*. It restores the original $\Box$ and $\diamond$ marks as well so that the context matches exactly and subsequent *call/cc* operations will succeed.

Variables and Values:

$$
\begin{array}{rl}
\mathcal{CMT}[x] = & x \\
\mathcal{CMT}[\sigma] = & \sigma \\
\mathcal{CMT}[(\lambda(\overline{x})\ e)] = & (\lambda(\overline{x})\ \mathcal{CMT}[e]) \\
\mathcal{CMT}[\kappa.\mathcal{E}] = & (\textbf{kont/ms}\ \chi_{\{\Box,\diamond\}}(\mathcal{CMT}[\mathcal{E}], (\text{nil}))) \\
\mathcal{CMT}[(K\ \overline{a})] = & (K\ \overline{\mathcal{CMT}[a]})
\end{array}
$$

Redexes:

$$
\begin{array}{rl}
\mathcal{CMT}[(\overline{w})] = & (\overline{\mathcal{CMT}[w]}) \\
\mathcal{CMT}[(\textbf{letrec}\ (\overline{[\sigma\ w]})\ e)] = & (\textbf{letrec}\ (\overline{[\sigma\ \mathcal{CMT}[w]]})\ \mathcal{CMT}[e]) \\
\mathcal{CMT}[(\text{w-c-m}\ \overline{e})] = & (\text{w-c-m}\ \overline{\mathcal{CMT}[e]}) \\
\mathcal{CMT}[(\text{c-c-m}\ [\overline{a}])] = & (\text{c-c-m}\ [\overline{\mathcal{CMT}[a]}]) \\
\mathcal{CMT}[(\text{,atch}\ w\ \overline{l})] = & (\text{match}\ \mathcal{CMT}[w]\ \overline{\mathcal{CMT}[l]}) \\
\mathcal{CMT}[[(K\ \overline{x})e]] = & [(K\ \overline{x})\mathcal{CMT}[e]] \\
\mathcal{CMT}[(\text{abort}\ e)] = & (\text{abort}\ \mathcal{CMT}[e]) \\
\mathcal{CMT}[(\text{call/cc}\ w)] = & (\mathcal{CMT}[w]\ \textbf{kont})
\end{array}
$$

$$
\begin{array}{rl}
\textbf{kont} = & (\textbf{kont/ms}\ (\text{c-c-m}\ [\Box\ \diamond])) \\
\textbf{kont/ms} = & (\lambda(m)\ (\lambda(x) \\
 & (\text{abort}\ (resume\ m\ x))))
\end{array}
$$

Contexts:

$$
\begin{array}{rl}
\mathcal{CMT}[[]] = & [] \\
\mathcal{CMT}[(\overline{w}\ \underline{\mathcal{E}})] = & (K\ (\text{w-c-m}\ \Box\ K\ \mathcal{CMT}[\underline{\mathcal{E}}])) \\
 & \text{where } K = (\lambda(x)\ (\overline{\mathcal{CMT}[w]}\ x)) \\
\mathcal{CMT}[(\text{w-c-m}\ v\ v'\ \underline{\mathcal{E}})] = & (\text{w-c-m}\ v\ v'\ (\text{c-w-i-c-m}\ \diamond\ (\lambda\ (cms) \\
 & (\text{w-c-m}\ \diamond\ (\text{map-set}\ cms\ v\ v') \\
 & \mathcal{CMT}[\underline{\mathcal{E}}])))))
\end{array}
$$

Compositions:

$$\mathcal{CMT}[\underline{\mathcal{E}}[r]] = \mathcal{CMT}[\underline{\mathcal{E}}][\mathcal{CMT}[r]]$$

Figure 9. Translation from SL to TL

4.4 Correctness

Let

$$\text{eval}_x(p) = \begin{cases} v & \text{if } \emptyset/p \rightarrow^* v \\ \bot & \emptyset/p \rightarrow^* \ldots \end{cases}$$

Theorem 1. $\mathcal{CMT}[eval_{SL}(p)] = eval_{TL}(\mathcal{CMT}[p])$

Overview. If a source term reduces in k steps, then its translation will reduce in at least k steps, such that the result of the translation's reduction is the translation of the source's result. This is proved by induction on k. The base case is obvious, but the inductive case must be shown by arguing that TL simulates each step of SL in a finite number of steps. This argument is captured in the next lemma. □

Lemma 2 (Simulation)**.**
If $\Sigma/\mathcal{E}[e] \rightarrow_{SL} \Sigma'/\mathcal{E}'[e']$ *then* $\mathcal{CMT}[\Sigma]/\mathcal{CMT}[\mathcal{E}[e]] \rightarrow^+_{TL} \mathcal{CMT}[\Sigma']/\mathcal{CMT}[\mathcal{E}'[e']]$

Overview. This is proved by a case analysis of the $\rightarrow_{SL}$ relation. It requires additional lemmas that cover the additional

```
(letrec
 ([resume
   (λ (l v)
    (match l
     [(nil) v]
     [(cons ms l)
      (match ms
       [(nil) (resume l v)]
       [(cons (cons □ k) nil)
        (k (w-c-m □ k (resume l v)))]
       [(cons (cons ◇ cms) nil)
        (restore-marks cms
         (λ () (w-c-m ◇ cms (resume l v))))]
       [(cons (cons □ k) (cons ◇ cms))
        (w-c-m □ k
         (restore-marks cms
          (λ () (w-c-m ◇ cms (resume l v)))))]
       [(cons (cons ◇ cms) (cons □ k))
        (w-c-m □ k
         (restore-marks cms
          (λ () (w-c-m ◇ cms (resume l v))))))]))]))]
  [restore-marks
   (λ (cms thnk)
    (match cms
     [(nil) (thnk)]
     [(cons (cons m v) cms)
      (w-c-m m v (restore-marks cms thnk))]))]
  [c-w-i-c-m (λ (k proc default-v) ...)]
  [map-set (λ (map k v) ...)])
 ...)
```

Figure 10. Necessary Definitions

steps that $\rightarrow_{TL}$ takes to reduce a program to images of subexpressions/contexts of the original. □

Lemma 3 (Compositionality)**.**
$\mathcal{CMT}[\Sigma]/\mathcal{CMT}[\mathcal{E}][\mathcal{CMT}[e]] \rightarrow^*_{TL} \mathcal{CMT}[\Sigma]/\mathcal{CMT}[\mathcal{E}][e]$

Sketch. $\mathcal{CMT}[]$ only introduces **w-c-m** into the context or abstracts the continuation of an argument to a function. These additional contexts are eventually erased as the argument is evaluated or the surrounding **w-c-m** is removed as a value is returned. □

Lemma 4 (Reconstitution)**.**

$$\begin{array}{ll} & \mathcal{CMT}[\Sigma]/(resume\ \chi_{\{\square,\diamond\}}(\mathcal{CMT}[\mathcal{E}'])\ \mathcal{CMT}[v]) \\ \rightarrow^+_{TL} & \mathcal{CMT}[\Sigma]/\mathcal{CMT}[\mathcal{E}'][\mathcal{CMT}[v]] \end{array}$$

Proof. We proceed by cases on the structure of $\mathcal{E}'$.

Suppose $\mathcal{E}' = []$, then $\mathcal{CMT}[\mathcal{E}'] = []$, so χ returns (nil) and $resume$ returns $\mathcal{CMT}[v]$, which is equal to $[][\mathcal{CMT}[v]]$.

Suppose $\mathcal{E}' = (\overline{w}\ \mathcal{E})$, then $\mathcal{CMT}[]$ expands to a mark that captures $\mathcal{E}'$ as a function abstracted over $\mathcal{E}$ in the □ mark, which is restored by $resume$. $\mathcal{E}$ is preserved by induction.

Suppose $\mathcal{E}' = (\text{w-c-m}\ v\ v'\ \mathcal{E})$, then $\mathcal{CMT}[]$ expands to a mark that captures v and v' in the ◇ mark, which is restored by $resume$. $\mathcal{E}$ is preserved by induction. □

Lemma 5 (Substitution)**.**

$$\mathcal{CMT}[e[\overline{x \mapsto v}]] = \mathcal{CMT}[e][\overline{x \mapsto \mathcal{CMT}[v]}]$$

Sketch. The $\mathcal{CMT}[]$ transformation is applied to every subexpression in the transformed expression. Thus, the vs substituted in will eventually have $\mathcal{CMT}[]$ performed on them if x appears in e. If an identifier appears in the argument to $\mathcal{CMT}[]$, it is not transformed, but left as is, so it could be substituted after the transformation with the $\mathcal{CMT}[]$ of the value v. □

4.5 Defunctionalization

We do not need to extend the defunctionalization defined by Pettyjohn et al. (2005) in any interesting way, but in our implementation we have extended it in the trivial way to keyed continuation marks and the numerous PLT λ forms.

5. Extensions

Continuation marks, however, are not the only expressive features of PLT Scheme we aim to support. We discuss how to support fluid state, Web cells, advanced control flow, and continuation management in turn.

5.1 Parameters

Web applications often use *fluid state*. State is fluid when it is bound only in a particular dynamic context. The continuation mark demonstration from the introduction (Figure 3) is an application of fluid state: when the user authenticates to CONTINUE, the "current user identity" is bound in the dynamic context of the *display-site* call. Every link presents the user with a new page using the same user identity. If another type of state were used, it would be more difficult to program or would prevent certain kinds of user behavior. For example, if the environment were used, then the user identity would need to be an argument to every function; if the store were used, then there would be only one user identity for all URLs associated with a session, thereby disallowing a "free" implementation of masquerading and delegation.

PLT Scheme provides a mechanism for fluidly bound state: parameters. Parameters are effectively a short-hand for continuation mark operations. (**parameterize** *p v e*) wraps *e* in a continuation mark for the key associated with *p* bound to *v*. The parameter *p* can then be referenced inside *e* and will evaluate to whatever the closest mark is.

Unfortunately, parameters are not implemented this way, because they also provide efficient lookup, thread-safety, and thread-local mutation. Instead, there is a single continuation mark key for all parameters. This key is not serializable, so our mark recording and reconstitution strategy fails. The key is included in the captured continuation structure but destroys its serializability. We compensate by providing an implementation of parameters using a distinct serializable key for each parameter. This way, Web servlets can effectively use fluid state, like parameters.

5.2 Web Cells

Sometimes fluid state (parameters), the environment (lexical variables), and the store (mutable structures) are all inappropriate for Web applications. A simple example is maintaining the preferred sort state of a list while the user is exploring the application, without forcing the user to have one sort state per session.

If fluid state is used, then the entire application must be written as tail calls to ensure that the dynamic extent of sort state modifications is the "rest" of the session. This means the program must be written in CPS.

If the lexical environment is used, then the sort state must be threaded throughout every part of the application, including those that are not related to the sorted list. This means the program must be written in store-passing style, an invasive and nonmodular global program transformation.

If the store is used, then a single sort state will be used for all browsers displaying the same list. This means the user will not be

able to use the Web interactions provided by the browser, such as Open in New Window, to compare different sorts of the same list.

Web cells (McCarthy and Krishnamurthi 2006) provide a kind of state appropriate for the sort state. Semantically, the set of Web cells is a store-like container that is part of the evaluation context; however, unlike the store, it is captured when continuations are captured and restored when they are invoked. Since continuation capture corresponds to Web interaction, this state is "fluid" over the Web interaction tree, rather than the dynamic call tree.

It is easy to add support for Web cells in our system. We have a thread-local box that stores the cells for the execution of a continuation. Whenever a continuation is captured, these are saved. Our serializable continuation data structure contains (a) the continuation components, (b) the continuation marks, and (c) the Web cell record. Each of these are restored when the continuation is called.

5.3 Request Handlers

send/suspend is the fundamental operator of the PLT Web Server. This function captures the current continuation, serializes it into a URL, and calls a display function with the URL. This works well for applications with a linear control-flow. However, most applications have many possible continuations (links) for each page, and therefore are difficult to encode with *send/suspend*.

We can simulate this control-flow by dispatching on some extra data attached to a single continuation captured by *send/suspend*. This dispatching pattern is abstracted into *send/suspend/dispatch* (*s/s/d*) (Hopkins 2003). This function allows request handling procedures to be embedded as links; when clicked, the request is given to the procedure, and the procedure returns to *s/s/d*'s continuation.

For example, consider the servlet

```
(define (show message)
 (send/suspend/dispatch
  (λ (embed/url)
   `(html (h1 ,message)
     (a ([href ,(embed/url (λ _ (show "One")))]) "One")
     (a ([href ,(embed/url (λ _ (show "Two")))]) "Two")))))
```

This servlet generates a page with two links: one for each call to *embed/url*. When clients click on either, they are sent to an identical page that contains a header with the link text.

s/s/d can either build a hash-table and perform dispatching with a single continuation, or it may be written (Krishnamurthi et al. 2007) succinctly as

```
(define (send/suspend/dispatch mk-page)
  (let/cc application-context
    (local
      [(define (embed/url handler)
         (let/ec mk-page-context
           (application-context
            (handler
             (send/suspend
              mk-page-context)))))]
      (send/back (mk-page embed/url)))))
```

This encoding employs a clever use of continuations to embed the handler in the continuation captured by *send/suspend*. When *embed/url* is called, it captures the continuation of *mk-page*, *mk-page-context*, which is in the process of constructing a link. *embed/url* provides this link by giving the continuation *mk-page-context* to *send/suspend*, which calls its argument with a link to *send/suspend*'s continuation. *embed/url* is arranged so that when *send/suspend* returns, its return value is given to the handler, whose return value is given to the caller of *send/suspend/dispatch*, via *application-context*.

This encoding generates one continuation per call to *s/s/d* (*application-context*) and one escape ("one-shot") continuation per call to the embedding procedure (*mk-page-context*).

We can do better with serializable continuations. Because everything is serializable and manipulable, we can implement *s/s/d* as

```
(define (send/suspend/dispatch mk-page)
 (call-with-serializable-current-continuation
  (λ (application-context)
   (define (embed/url handler)
    (define application+handler-context
     (kont-cons handler application-context))
    (kont→url application+handler-context))
   (send/back (mk-page embed/url)))))
```

Like before, this implementation first captures the continuation of *s/s/d*. *embed/url* accepts a procedure and returns a continuation serialized into a URL. This serialized continuation is the continuation of *s/s/d* with the procedure appended to the end. Since the components of the continuation are represented as a list, we can do this directly. However, the continuation components are stored in reverse order, so a logical append is a prepend on the representation.

In the program,

```
(f (g (h (s/s/d (λ (embed/url) (embed/url i))))))
```

application-context is (*list h g f*) and *application+handler-context* is (*list i h g f*).

This captures only a single continuation regardless of how many handler procedures are embedded. This improves our time *and* space efficiency.

5.4 Continuation Management

In our system, continuations are serialized and embedded in the URLs given to the client by default. However, there are some pragmatic reasons why this is not always a good idea.

First, there is, in principle, no limit to the size of a continuation. If the lexical environment contains the contents of a 100MB file, then the continuation will be at least 100MBs (modulo clever compression). Most Web clients and servers support URLs of arbitrary length, but some browsers and servers do not. In particular, Microsoft Internet Explorer (IE) limits URLs to 2,048 characters and Microsoft IIS limits them to 16,384 characters.

Second, if a continuation is embedded in a URL and given to the user, then it is possible to manipulate the continuation in its serialized form. Thus, the environment and Web cell contents are not "secure" when handled directly by users.

Providing security is not always appropriate, so we allow Web application developers to customize the *kont→url* function that is used to embed continuations in URLs with "stuffers." We provide a number of different stuffer algorithms and the ability to compose them. They compose because they produce and consume serializable values.

Plain	The value is serialized with no special considerations.
GZip	The value is compressed with the GZip algorithm.
Sign	The value is signed with a parameterized algorithm.
Crypt	The value is encrypted with a parameterized algorithm.
Hash	The value is hashed with the MD5 or SHA1 algorithm. The value is serialized into a database addressed by the hash and the hash is embedded in the URL.
Len(s)	Stuffer s is used if the URL would be too long when stuffed with the value.

These techniques can be combined in many ways. For example, an application with small continuations and no need for secrecy could just use the Plain algorithm. An application that had larger continuations might add the GZip algorithm. An application that needed to protect against changes could add the Sign algorithm, while one that needed to guarantee the values could not be inspected might add Crypt. Finally, an application that did not want the expense in either bandwidth or computational time could just use the Hash algorithm. Every URL would be the same length, and identical continuations would be stored only once.

Although the Hash method is not truly RESTful, it performs drastically better than the traditional method of storing the continuations in memory. It uses less space because the continuation representation is tailored to the particular application, in contrast to the standard C-stack copy. Furthermore, it takes less time to service requests. This might seem implausible since the operating system's virtual memory system seems morally equivalent to a continuation database because unused parts of memory are moved to disk. However, the VM considers memory unused only when it is not touched by the application. In PLT Scheme, the garbage collector stores a tag bit with objects. Thus, even though the collector doesn't need to walk all data, collection affects these tag bits, which causes the operating system to bring the entire page into main memory. This paging, which would not be present with a swap-sensitive garbage collector (Hertz et al. 2005), causes severe performance degradation.

The Hash method has the additional advantage of providing multi-server scalability easily, compared to other possible server-side continuation stores. Since the Hash method guarantees that two writes to the same key must contain the same data, because otherwise the hashing algorithm would not be secure, multiple Web servers do not need to coordinate their access to the database of serialized continuations. Therefore, replication can be done lazily and efficiently, avoiding many of the problems that many session object databases are fraught with.

5.4.1 Replay Attacks

Since the URLs of our application completely specify what the Web application will do in response to a request, it is natural to assume that our applications are particular susceptible to replay attacks. For example, suppose we build a stock-trading application and at some point a user sells 10 shares. An adversary could capture the continuation for "sell 10 shares" and replay it n times to sell $10n$ shares, even with encryption in place. This seems utterly unacceptable.

However, consider the same application on another platform where the continuation is specified through an ad-hoc combination of URL, form data, and cookies. In this case as well, a request may be replayed to perform this attack. On a traditional platform, this would be prevented by some server-side state. For example, each server response would include a unique identifier that would be sent back with requests; each identifier would be allowed to be received only once, and the identifier would be cryptographically tied to the incoming requests, so that new identifiers could not be used to "freshen" old requests to replay them. This same strategy can be implemented in our system as well, except perhaps better because the unique identifier can be combined with the *entire* continuation since it is explicitly represented, in one place, in our system.

As before with the various stuffer algorithms, it is not always appropriate to disallow replays. For example, it is useful to use the browser's Refresh button and to send links to colleagues. If we provided replay protection "for free," we would also disallow many useful Web applications.

5.4.2 Serialization Format

Each continuation record is scarcely more than 100 bytes. This is split between Web cells, the continuation marks, and the continuation function components. The cells and marks are comparable to the lexical values captured in the continuation. Each function is serialized as a unique identifier that refers to the compiled code and the captured values of free variables. The continuation record has a list of these functions. A sanitized, sample record is below.

```
(serialized ((web-server/lang/abort-resume . web:kont)
             (web-server/lang/web-cells . web:frame)
             (application/servlet . web:300))
 (web:kont
  (web:frame
   (list (cons web:23-1 (path #"static-path" . unix))
         (cons web:10-0 (path #"db-path" . unix))
         (cons web:36-2 "username")))
  (list (vector (web:300) #f))))
```

This can be seen as a short program that constructs the serialized value. The first part records what modules contain the definitions of data-structures that are created. The module path refer to code loaded into the PLT Scheme instance that is deserializing the continuation. If they are resolved to the wrong code, or if the module are simply not available, then deserialization will fail. This means any PLT Scheme instance with access to the same source can deserialize this continuation. Our system protects against certain errors by including in a hash of the module source in the names of continuation data structures. In the real record that this example corresponds to, the token 300 would include a hash of the source of *application/servlet* to result in a deserialization error if the code were changed, rather than the unsafe behavior that would result if a different kind of continuation were populated with erroneous data from this record.

The second part is an expression that creates the continuation record. Its first field contains the record of the Web cells. This is an association list from identifying symbols to values. In this example, two of the values are paths, while the other is a string. The second field of the continuation is the continuation record. This is the list that will be passed to *resume*. In the example, there is a single function, *web:300*, with no accompanying continuation mark recording.

6. Evaluation

The formal treatment of Section 4 can tell us if the transformation is correct and if it formally has the modularity properties we desire, but it cannot tell us if it is useful for producing scalable, RESTful, direct-style Web applications.

6.1 Scalability

We observe that Web applications in our system written in direct style *can* be entirely RESTful. Their usage of the lexical environment, fluid state, and Web cells are all contained in serializable structures. These can then be stored by the client in encrypted and compressed URLs. Cookies can easily capture store state, and since nearly all data structures are serializable, any value can be stored in cookies. Finally, our programs may *choose* to use server state where appropriate.

However, our system would not really be useful if it greatly slowed down the developer (with compilation lag) or the client (with execution lag), so we measure those.

Compilation takes, on average, twice the amount of time as compiling normal Scheme. This measurement was based on compiling a collection of 15 servlets. This is because our compiler is implemented as a PLT module language (Flatt 2002) that performs

```
#lang scheme
(require web-server/servlet)
(define (get-number which)
 (string->number
  (extract-binding/single
   'number
   (send/suspend
    (λ (k-url)
     `(html
       (body
        (form ([action ,k-url])
         ,which " number:"
         (input ([name "number"])))))))))
(define (start req)
 `(html
   (body
    ,(number->string
      (+ (get-number "First")
         (get-number "Second"))))))
```

Figure 11. Add-Two-Numbers (Before)

```
#lang web-server          ; ← different
                          ; ← different
(define (get-number which)
 (string->number
  (extract-binding/single
   'number
   (send/suspend
    (λ (k-url)
     `(html
       (body
        (form ([action ,k-url])
         ,which " Number:"
         (input ([name "number"])))))))))
(define (start req)
 `(html
   (body
    ,(number->string
      (+ (get-number "First")
         (get-number "Second"))))))
```

Figure 12. Add-Two-Numbers (After)

five passes over the source code before it produces normal Scheme. These five passes cause a delay that is noticeable to developers but not prohibitive.

There is no noticeable difference in the execution time of our servlets versus standard servlets. Although it is possible to cause slow down by serializing large objects.

We tested scalability by comparing the space usage of a typical Web application before and after conversion to our system. Before, the LRU manager kept memory usage scarcely below 128MB. This pattern of "hugging the edge" of the limit matches our experience with CONTINUE (Krishnamurthi 2003). After conversion, the server uses about 12MB, of which approximately 10MB is the bytecode for the application and its libraries.

We tested with multiple serialization regimes. When GZip is used, no continuation is larger than IE's limit, so there is no per-session state on the server. When we use Hash, the continuation store is about 24MB for approximately 150 users. This means that we use about 20 percent of the preconversion storage, while providing *more* service, because continuations are never revoked. But remember, we don't *need* to use that storage because the client can hold every continuation.

6.2 Modularity & Compatibility

The final way we evaluate our work is by its ability to run unmodified Scheme programs, in particular, Web applications. In most cases, there is no difficulty whatsoever; the user simply changes one line at the top of his or her program. Figure 11 presents a servlet and Figure 12 shows the same servlet using our transformation: the first line selects the compiler, and the second eliminates the unnecessary **require** specification.

There are two categories of programs that lead to errors when transformed. The first category is programs that include nonserializable data structures in the environment of their captured continuations. The second category is programs that use higher-order library procedures with arguments that capture continuations.

6.2.1 Non-serialized Data Structures

Our transformation implements continuations with closures and renders closures serializable through defunctionalization. However, other data structures remain unserializable: ports, foreign pointers, global and thread-local boxes, untransformed closures, parameters, etc. If a program includes these data structures in the environment of serialized continuations, then the continuation is not serializable. In most cases this is not problematic, because these data structures are often defined at the global level or used during a computation but not between Web interactions. For example, it is much more common for the function + to be invoked during a computation than for the function + to be stored in a list that it is in the environment of a continuation. Only the second prevents serialization. Since these practices are so uncommon, we have not found constraint to prevent compatibility in practice.

6.2.2 Higher-order Third-Party Library Procedures

Programs that use higher-order third-party library procedures cannot be used safely with our system. For example,

```
(map get-number (list "First" "Second"))
```

This does not work because the *send/suspend* inside of *get-number* relies on the □ mark to capture the continuation, but because *map* is not transformed, its part of the continuation is not recorded. We can detect this situation and signal a runtime error as described by Pettyjohn et al. (2005). However, it is *always* possible to recompile the necessary code (i.e., *map*) under our transformation.

7. Conclusion

We presented a modular program transformation that produces RESTful implementations of direct style Web programs that use expressive features, like continuation marks.

We have discussed how to extend this transformation into a deployable system. We have discussed the opportunities for continuation management this allows. We have evaluated the performance of our work and found that it meets the gold standard of scalability—no server-side session state—and can use as little as 10% of the memory when server-side state is desirable.

This work relies on continuation marks, so it is difficult to apply it to programming languages other than PLT Scheme. However, practitioners could apply our technique easily once continuation marks were available. Since continuation marks can be implemented for both C# (Pettyjohn et al. 2005) and JavaScript (Clements et al. 2008), it should be possible to automatically produce RESTful Web applications in those languages as well.

In the future, we will explore how to allow continuation capture in an untransformed context. We anticipate that the WASH approach (Thiemann 2006) of combining multiple continuation capture methods will be appropriate.

Acknowledgments We thank Matthew Flatt for his superlative work on PLT Scheme. We thank Greg Pettyjohn for his work on the prototype our system is based upon. We thank Matthias Felleisen, Matthew Flatt, Shriram Krishnamurthi, and the anonymous reviewers for their comments on this paper. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

References

John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, April 2001.

John Clements, Ayswarya Sundaram, and David Herman. Implementing continuation marks in JavaScript. In *Scheme and Functional Programming Workshop*, 2008.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, 2006.

Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside - a multiple control flow web application framework. In *European Smalltalk User Group - Research Track*, 2004.

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992.

Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, 2002.

M. J. Fischer. Lambda calculus schemata. *ACM SIGPLAN Notices*, 7(1):104–109, 1972. In the *ACM Conference on Proving Assertions about Programs*.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *SIGPLAN Notices*, 39(4):502–514, 2004.

Matthew Flatt. Composable and compilable macros. In *International Conference on Functional Programming*, 2002.

Paul Graham. Lisp for web-based applications, 2001. `http://www.paulgraham.com/lwba.html`.

Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Programming Language Design and Implementation*, pages 143–153, 2005.

Peter Walton Hopkins. Enabling complex UI in Web applications with send/suspend/dispatch. In *Scheme Workshop*, 2003.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000.

Shriram Krishnamurthi. The CONTINUE server. In *Practical Aspects of Declarative Langauges*, January 2003.

Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and Use of the PLT Scheme Web Server. *Higher-Order and Symbolic Computation*, 2007.

Jacob Matthews, Robert Bruce Findler, Paul T. Graunke, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the Web. *Automated Software Engineering*, 11(4):337–364, 2004.

Jay McCarthy and Shriram Krishnamurthi. Interaction-safe state for the Web. In *Scheme and Functional Programming*, September 2006.

Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *International Conference on Functional Programming*, September 2005.

Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1975.

Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *International Conference on Functional Programming*, pages 23–33, 2000.

Peter Thiemann. Wash server pages. *Functional and Logic Programming*, 2006.

Noel Welsh and David Gurnell. Experience report: Scheme in commercial web application development. In *International Conference on Functional Programming*, September 2007.