The Two-State Solution

Native and Serializable Continuations Accord

Jay A. McCarthy

PLT
Computer Science Department
Brigham Young University
Provo, UT, USA
jay@cs.byu.edu

Abstract

Continuation-based Web servers provide advantages over traditional Web application development through the increase of expressive power they allow. This leads to fewer errors and more productivity for the programmers that adopt them. Unfortunately, existing implementation techniques force a hard choice between scalability and expressiveness.

Our technique allows a smoother path to scalable, continuation-based Web programs. We present a modular program transformation that allows scalable Web applications to use third-party, higher-order libraries with higher-order arguments that cause Web interaction. Consequently, our system provides existing Web applications with more scalability through significantly less memory use than the traditional technique.

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

General Terms Languages, Performance, Theory

Keywords Delimited Continuations, Stack Inspection, Web Applications

1. Introduction

Web computations are described by the capture and resumption of continuations on the Web server. This is accepted wisdom now in the functional programming community (Hughes 2000; Queinnec 2000; Graham 2001). Furthermore, it is the theoretical basis for many practical Web application development frameworks (Matthews et al. 2004; Ducasse et al. 2004; Pettyjohn et al. 2005; Thiemann 2006; Cooper et al. 2006; Krishnamurthi et al. 2007; McCarthy 2009).

Unfortunately, these frameworks rely on techniques that force users to make all-or-nothing trade-offs between program expressiveness and industrial scalability. Whole program compilers achieve scalability but sacrifice interaction with third-party libraries (Matthews et al. 2004; Cooper et al. 2006). Modular compilation techniques (Pettyjohn et al. 2005; McCarthy 2009) achieve scalability but sacrifice higher-order interaction with third-party libraries. First-class continuation-based Web servers (Ducasse et al. 2004; Krishnamurthi et al. 2007) do *not* achieve scalability (Welsh and Gurnell 2007), but do *not* sacrifice any expressiveness. Worse still, each of these techniques must essentially be used in isolation; it is not possible to gradually control the trade-off between expressiveness and scalability.

We present an implementation technique, and its formal model, that allows controlled scalability for Web applications that use higher-order third-party libraries and total scalability for those that do not use these libraries. This allows Web applications written using our system to use drastically less memory than before without sacrificing expressiveness.

2. Background

The central problem of Web application implementation is caused by the statelessness of HTTP: when the server responds to a client's request, the connection is closed and the server program exits. If the client needs to communicate with the server program, its next request must contain enough information to resume the computation.

All Web programmers understand this problem, but functional programmers understand that this resumption information *is* the continuation. This is easily demonstrated by porting a small application from the command-line to the Web.[1]

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

[1] All program examples are written in Racket (Flatt and PLT June 7, 2010).

```
(define (get-number i)
  (prompt "Please provide number #~a:\n" (add1 i)))

(define (sum how-many)
  (define the-sum (num-sum (build-list how-many get-number)))
  (printf "Sum of ~a is ~a.\n" how-many the-sum))
```

The program *sum*, when given an integer, requests that many numbers from the user and returns their sum. When the user is providing the third of four numbers, the continuation is

```
(define the-sum
 (num-sum
  (list* 42 1830 [] (build-list 1 get-number))))
(printf "Sum of ~a is ~a.\n" 4 the-sum)
```

where [] represents the hole. *This* is the continuation that must be captured and given to the client in a Web application version to resume the computation appropriately.

Traditional Web programmers manually produce code that makes this continuation explicit, and many whole program transformation-based Web application frameworks (Matthews et al. 2004) automatically compile code that represents this continuation. Both the programmers and the tools produce code like:

```
(define (get-number/k i k)
  (web-prompt
   k "Please provide number #~a:\n" (add1 i)))

(define (got-numbers how-many l)
  (num-sum/k l
   (make-kont got-sum
    (list (cons 'how-many how-many)))))

(define (got-sum how-many sum)
  (format "Sum of ~a is ~a.\n" how-many sum))

(define (sum how-many)
  (build-list/k how-many get-number/k
    (make-kont got-numbers
     (list (cons 'how-many how-many)))))
```

This code is striking because the third-party libraries *num-sum* and *build-list* must be rewritten to be in continuation-passing style (CPS) (Fischer 1972) in the form of the new functions *num-sum/k* and *build-list/k*. This is only acceptable when all the source code for the application is available for programmer rewriting or compiler transformation.

In contrast, most first-class continuation-based Web application frameworks support the original program with only small modifications. The Racket Web Server (Krishnamurthi et al. 2007) version is almost identical to the original:

```
(define (get-number i)
  (web-prompt "Please provide number #~a:\n" (add1 i)))

(define (sum how-many)
  (define the-sum
   (num-sum (build-list how-many get-number)))
  (format "Sum of ~a is ~a.\n" how-many the-sum))
```

This is possible because *web-prompt* captures the precise continuation that resumes the computation when the user's input is available. Unfortunately, first-class continuation-based Web application frameworks have much worse scalability properties than other approaches (see Welsh and Gurnell (2007) for an anecdote.)

The heart of the problem is that in many languages, especially those that mix a high-level language with a C-based infrastructure, continuation serialization is simply not expressible[2]. Thus native continuations are typically stored in the server's memory and the client is provided 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.

Luckily, there is at least one more functional implementation strategy for continuation-based Web applications: modular program transformation based on continuation marks (Pettyjohn et al. 2005; McCarthy 2009). These work by transforming part of the code to duplicate the continuation components (stack frames) into continuation marks (Clements et al. 2001). Continuation marks are a generalization of the stack inspection found in other languages (for example, Java security through stack inspection or exception handling). By copying the continuation into the marks, the continuation can be extracted when user interaction is required *and* there is representation freedom, so they can be serialized to the client.

Unfortunately, this strategy relies on the insertion of marks during the entire context up to continuation capture. The implication of this is that continuations can only be captured in transformed contexts, such as during the evaluation of higher-order arguments to higher-order third-party library functions. Our example does this in the call to *build-list*, so if we use the mark-based serialization strategy we would need to reimplement *build-list* to expose it to the transformation. Our code is close to the original version:

```
(define (get-number i)
  (web-prompt "Please provide number #~a:\n" (add1 i)))

(define (sum how-many)
  (define the-sum
   (num-sum (my-build-list how-many get-number)))
  (format "Sum of ~a is ~a.\n" how-many the-sum))
```

[2] Of course, some systems (Cejtin et al. 1995) do support native continuation serialization. Given that so many system do not allow this, our work is widely applicable.

Strategy	Scalable	Automated	*num-sum* Unchanged	*build-list* Unchanged
Manual	Yes		Yes	
CPS	Yes	Yes		
First-Class		Yes	Yes	Yes
Modular	Yes	Yes	Yes	
Two-State	Yes	Yes	Yes	Yes

Table 1. Web Application Implementation Options

Compared to whole-program transformations, this strategy is a win because only *build-list* needs a reimplementation (*my-build-list*), not *num-sum*.

At this point we have exhausted existing implementation choices. Our options are depicted in Table 1. Obviously we need a strategy that is scalable and automated, and does not require us to rewrite any third-party libraries. We present such an implementation technique. In the end, our program will look like:

```
(define (get-number i)
  (native→serial
   (web-prompt
    "Please provide number #~a:\n" (add1 i))))

(define (sum how-many)
  (define the-sum
    (num-sum
     (serial→native (build-list how-many get-number))))
  (format "Sum of ~a is ~a.\n" how-many the-sum))
```

Intuitively, we use delimited continuations to capture the native continuations that the mark-based serializable continuations cannot capture. The functions serial→native and native→serial correspond, respectively, to *with-continuation-promp* and *call-with-delimited-continuation*.

Thus, we use two different kinds of state: server-side state for the small native delimited continuations and client-side state for the larger serializable continuations. Our "two-state" solution is much more scalable than the first-class continuation approach and only slightly worse than the modular approach. It provides an essential way point between native (first-class) continuations and serializable (modular) continuations for developers to use when reimplementation and recompilation is not possible or is otherwise prohibitive.

3. Intuition

Our implementation can be seen as an extension to our prior work on modular serializable continuations (McCarthy 2009); however, the essence is captured by an extension to the work that our work was based on: the Pettyjohn et al. (2005) transformation. Therefore, we will present our intuition and formalism in that context, even though the actual implementation supports our continuation mark extensions.

The Pettyjohn et al. (2005) transformation relies on the modular A-Normal Form (ANF) transformation to name continuation components (Flanagan et al. 2004) and stack inspection through continuation marks to provide the "capturing" part of *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 in contrast to naïve CPS.

Many programming languages and environments allow access to the run-time 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 Racket 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 Racket's tail-call optimization. These two mechanisms allow marks to be used without interfering with existing code.

A pedagogic example of continuation mark usage is presented in Figure 1. *fact* is the factorial function with instrumentation using continuation marks: **w-c-m** records function arguments on the stack with the 'fact mark, even function arguments on the stack with the 'even mark, and *c-c-m* collects both mark values in the base case. In the result of *c-c-m* stack frames that have no 'even mark are recorded as #f to ensure that the relation on the stack between the different marks is discernible. *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 for 'fact and the only mark for 'even.

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

```
(define-syntax-rule (mark-if-even n e)
 (if (even? n)
     (w-c-m 'even n e)
     e))

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

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

Figure 1. Factorial with Continuation Marks

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.

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

```
(k (w-c-m SQUARE k a))
```

where SQUARE is a special key known only to the transformation. This effectively duplicates the continuation in a special mark. Then *call/cc* is defined as

```
(define (call/cc e)
 (define ks (c-c-m SQUARE))
 (e (λ (x) (abort (resume ks x)))))
```

where $\underline{\lambda}$ constructs a serializable closure by generating a fresh structure that runs its λ-lifted (Johnsson 1985) body when applied; and *resume* restores the continuation record from the SQUARE marks into an actual control stack. *resume* must also reinstall the SQUARE marks so subsequent invocations of *call/cc* are correct.

```
(define (resume l x)
 (match l
  [(list) x]
  [(cons k l)
   (k (w-c-m SQUARE 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 main deficiency of this transformation is that *call/cc* is useless when called from an untransformed context. For example, in the program

```
(+ 1 (build-list how-many (λ (i) (call/cc (λ (k) ...)))))
```

where *build-list* is not available to the transformation, the continuation *k* is generated from the SQUARE marks and therefore does not contain the code for *build-list*, because it was never transformed to create SQUARE marks. That is, the continuation is recreated as (+ 1 []) rather than the true context inside *build-list*.

The first step will be to formalize the implementation Pettyjohn et al. (2005) used to detect such problematic uses of *call/cc*.[3] The second step will be to use this *detection* technique as a hook to *eliminate* the problem and provide access to the actual part of the native continuation that was not captured by the mark-based transformation.

3.1 Third-Party Higher-Order Libraries with Higher-Order Arguments That Capture Continuations

The essence of the problem is that when untransformed code is called, it is unsafe to try and capture the continuation in its dynamic context, because the SQUARE marks are not available. Therefore, we need to learn when a *call/cc* attempt occurs in the dynamic context of untransformed code. The real question, then, is how do we observe our dynamic context?

This, of course, is the purpose of continuation marks. We can create a continuation mark key called SAFE? that marks whether the context is safe for continuation capture. Our implementation of *call/cc* will extract these marks and ensure that each is true:

```
(define (call/cc e)
 (define ks (c-c-m SQUARE))
 (define safe? (c-c-m SAFE?))
 (if (safe-context? safe?)
  (e (λ (x) (abort (resume ks x))))
  (error 'call/cc "Unsafe context")))

(define (safe-context? safe?-marks)
 (andmap identity safe?-marks))
```

This, however, assumes that we mark unsafe contexts by wrapping them in SAFE? continuation marks. But how do we know if a context is unsafe before entering it?

In an implementation there are many options, such as looking at the exporting module of a function's definition and a flow-analysis to determine if lexical identifiers are

[3] The intuitive description of this implementation is found in item 2 of the list at the end of section 4.1 of their paper, roughly on page 8.

bound to third-party functions. But in our theory, we can be conservative and assume that all contexts are unsafe, but explicitly mark contexts that we know are safe.

We wrap all function application in a continuation mark that signifies its unsafety: the application (*k a*) is transformed as

```
(w-c-m SAFE? false (k (w-c-m SQUARE k a)))
```

This mark will be available in the dynamic context of *k* to detect that the context is unsafe. Of course, not all contexts are unsafe so we must explicitly mark the safe context as such. The safe contexts are inside safe functions; and the safe functions are the ones that are transformed, so if we transform all function definitions, we will track safety.

The function (λ (*x* ...) *expr* ...) will be transformed to

```
(λ (x ...)
 (w-c-m SAFE? true
  expr ...))
```

When a transformed function is called, it adds a safety mark to its context. If the function is called from a safe (transformed) context, then there is already an unsafety mark on the context; but the tail call property of continuation marks ensures that the *safety* mark will overwrite the *unsafety* mark, so *call/cc* will succeed.

In contrast, when a transformed function is called from an unsafe (third-party) context, unless the call is in tail position, the continuation frame does *not* contain an unsafety mark, so the safety mark does not override any marks. However, *call/cc* will fail because when it captures the SAFE? marks with *c-c-m*, it will extract a list like (*list* ... *false true* ...) where the frames between *false* and *true* were the third-party context.

An interesting subtle point is that continuation capture is safe in the higher-order arguments to third-party higher-order functions if they are always called from tail-position.

At this point, we have an account of how to detect and avoid continuation capture when the context is not transformed and does not contain the SQUARE marks that make capture safe and sound. In the next section, we will transform unsafe contexts into safe contexts.

3.2 Delimiting the Native Continuation

In the previous section, we effectively used the SAFE? mark to *delimit* the part of the continuation where the transformation was not run and thus where the SQUARE marks are not available. This allows us to detect that the user-level *call/cc*, implemented by extracting SQUARE marks, would fail because it misses part of the continuation during reconstruction. In the context of Web applications all is not lost, because we can always capture a native continuation and store it on the server. However, that severely limits our scalability.

If we were to capture a complete native continuation, we'd be doing something even worse: capturing native versions of continuation components that are serializable. For instance, in this program

```
(+ 5 (map (λ (x) (native-call/cc ...)) (list 1)))
```

if *map* were not transformed but the rest of the program were, then the native continuation captured by *native-call/cc* would be:

```
(+ 5 (w-c-m SQUARE (λ (x) (+ 5 x))
       (w-c-m SAFE? false (list []))))
```

Even though we have a serializable representation of the application of +, we would still capture that part of the continuation natively.

Fortunately for us, practical implementations of *delimited* continuations are already well developed (Gasbichler and Sperber 2002; Dyvbig et al. 2007; Flatt et al. 2007). While a full continuation captures the entire program context, delimited continuations provide two operations: *with-continuation-prompt*, which installs a "prompt" that serves as the upper bound of delimited capture, and *call-with-delimited-continuation*, which captures the context up to the nearest dynamic prompt.[4]

What we need is not just to mark delimited contexts as safe, but rather capture those contexts for later use. We require that programmers explicitly specify that they are willing to pay for this behavior by annotating the entry into untransformed code with serial→native and the return to transformed code with native→serial. For example,

```
(+ 5 (serial→native
       (map (λ (x) (native→serial (call/cc ...))) (list 1))))
```

Intuitively, serial→native is *with-continuation-prompt* and native→serial is *call-with-delimited-continuation*, but there are some subtleties. Here is a first attempt at the transformation:

```
(+ 5 (w-c-m SAFE? false
       (with-continuation-prompt
        (map (λ (x)
          (w-c-m SAFE? true
            (call-with-delimited-continuation
              (λ (dc) (call/cc ...)))) (list 1))))
```

But, of course, we need to communicate the delimited continuation (*dc*) to the serializable *call/cc* implementation:

```
(+ 5 (w-c-m SAFE? false
       (with-continuation-prompt
        (map (λ (x)
          (w-c-m SAFE? true
            (call-with-delimited-continuation
              (λ (dc)
                (w-c-m UNSAFE-PART dc
                  (call/cc ...)))) (list 1))))
```

[4] In most implementations, including Racket's, continuation prompts are "tagged" to support multiple distinct prompts.

This sets things up so that *call/cc* can extract the UNSAFE-PART marks in addition to the SQUARE marks. These native continuations can then be stored on the server in a global hash table, like normal native continuations are; synthetic SQUARE marks can be generated that look up and execute these native continuations:

```
(define (call/cc e)
 (define ks+unsafe (c-c-m SQUARE UNSAFE-PART))
 (define ks (store-unsafe-on-server ks+unsafe))
 (define safe? (c-c-m SAFE?))
 (if (andmap identity safe?)
  (e (λ (x) (abort (resume ks x))))
  (error 'call/cc "Unsafe context")))

(define store-unsafe-on-server
 (match-lambda
  [(list) (list)]
  [(list (vector k #f) l)
   (cons k (store-unsafe-on-server l))]
  [(list (vector #f unsafe) l)
   (define cont-id (store-on-server! unsafe))
   (cons (λ args
           (apply (lookup-on-server cont-id) args))
    (store-unsafe-on-server l))]))
```

where *store-on-server!* stores a value on the server and returns a serializable value that can be used to fetch the value later.

Our implementation of *resume* does not need to change because *store-unsafe-on-server* perfectly prepares the native continuations for use as if they are serializable, as it expects. However, there is still one problem: we are still refusing to construct the continuation, because all the SAFE? marks are not *true*. We must adapt so that false SAFE? marks only matter if they are not immediately followed by UNSAFE-PART marks.

We redefine *safe?* in *call/cc* as:

```
(define safe? (c-c-m SAFE? UNSAFE-PART))
```

and rewrite *safe-context?* as:

```
(define (safe-context? safe?+unsafe-part-marks)
 ; We start off in a non-native context
 (safe-context?/native-context
  safe?+unsafe-part-marks
  false))

(define (safe-context?/native-context
         safe?+unsafe-part-marks
         in-native?)
 (match safe?+unsafe-part-marks
  ; We cannot end in a non-native context
  [(list) (not in-native?)]
  [(cons (vector safe? unsafe-part) rest)
   (and
    ; If we are in a native context, then we must have
    ; captured a native continuation. Otherwise, this
    ; context is unsafe.
    (if in-native?
     unsafe-part
     true)
    ; If this part of the context is safe, we recur; safe?
    ; tells us if the context is native or transformed.
    (safe-context?/native-context rest (not safe?)))]))
```

We now have all the machinery in place to allow continuation capture in the higher-order arguments to third-party higher-order libraries. The key is a *two-state* solution, where some parts of the state stay on the server, while other parts are serialized to the client. Delimited continuations allow us to simply specify the part of the context that must remain on the server. Existing machinery for managing server-sided resources can be applied to these delimited native continuations.

4. Formal Treatment

In this section, we formalize the transformation where capturing continuations in unsafe contexts is detectable.

4.1 Source Language

Figure 2 presents grammar for the source language (SL). It is a modified version of A-Normal form (ANF) (Flanagan et al. 2004) because the continuation is always syntactically obvious (in the first w of applications.) It uses λ rather than let and has applications of arbitrary length. The language is extended with $call/cc$, pattern matching on algebraic data types, and $letrec$ for recursive binding.

Identifiers bound by $letrec$ (σ, e.g. **map**) are typeset differently than normal identifiers (x, e.g. map) to easily distinguish them.

Instances of algebraic data types are created with constructors (K) and destructured with $match$. Constructors (e.g. CONS) are typeset differently than identifiers (e.g. $cons$) to easily distinguish them.

The most significant non-standard aspect of the language is the presence of $\overline{\lambda}$. These represent functions that are from third-party libraries where the source code is unavailable. They behave identically to normal functions, but provide a cue to the transformation. When we discuss the transformation (Section 4.3), we will point out that the transformation does not apply to their bodies.

The operational semantics is specified via the rewriting system in Figure 3 (top and middle.) The $\rightarrow$ is used to denote reduction; the $\Rightarrow$ is used to denote reduction that is invariant on the store; and the $\rightsquigarrow$ is used to denote reduction in a context ($\mathcal{E}$) that is invariant on the store.

The semantics is heavily based on source language semantics of Pettyjohn et al. (2005). The [beta] and [beta (unsafe)] rules are the standard β_v-rewriting rule for call-by-value languages (Plotkin 1975). The [match] rule handles pattern

$$\frac{e \rightsquigarrow e'}{\mathcal{E}[e] \Rightarrow \mathcal{E}[e']} \qquad \frac{e \Rightarrow e'}{\Sigma, e \rightarrow \Sigma, e'}$$

Shared reductions

$$((\lambda\ (x\ \ldots)\ e)\ v\ \ldots) \rightsquigarrow e[x \leftarrow v]\ \ldots \quad \text{[beta]}$$

$$(match\ (K\ v\ \ldots)\ [(K\ x\ \ldots)\ \Rightarrow e]\ l\ \ldots) \rightsquigarrow e[x \leftarrow v]\ \ldots \quad \text{[match]}$$

$$(match\ (K_1\ v\ \ldots)\ [(K_2\ x\ \ldots)\ \Rightarrow e]\ l\ \ldots) \rightsquigarrow (match\ (K_1\ v\ \ldots)\ l\ \ldots) \quad \text{[match (next)]}$$

where $K_1 \neq K_2$

$$(match\ v\ [else\ \Rightarrow e]\ l\ \ldots) \rightsquigarrow e \quad \text{[match (else)]}$$

$$\Sigma,\ \mathcal{E}[(letrec\ ([\sigma\ v]\ \ldots)\ e)] \rightarrow \Sigma[\sigma\ \mapsto\ v]\ldots, \mathcal{E}[e] \quad \text{[letrec]}$$

$$\Sigma,\ \mathcal{E}[(\sigma\ v\ \ldots)] \rightarrow \Sigma,\ \mathcal{E}[(\Sigma(\sigma)\ v\ \ldots)] \quad [\sigma \text{ + apply}]$$

$$\Sigma,\ \mathcal{E}[(match\ \sigma\ l\ \ldots)] \rightarrow \Sigma,\ \mathcal{E}[(match\ \Sigma(\sigma)\ l\ \ldots)] \quad [\sigma \text{ + match}]$$

SL reductions

$$((\overline{\lambda}\ (x\ \ldots)\ e)\ v\ \ldots) \rightsquigarrow e[x \leftarrow v]\ \ldots \quad \text{[beta (unsafe)]}$$

$$\mathcal{E}[(call/cc\ v)] \Rightarrow \mathcal{E}[(v\ \kappa.\mathcal{E})] \quad \text{[call/cc]}$$

$$\mathcal{E}_1[(\kappa.\mathcal{E}_2\ v)] \Rightarrow \mathcal{E}_2[v] \quad \text{[cont invoke]}$$

TL reductions

$$\mathcal{E}[(abort\ e)] \Rightarrow e \quad \text{[abort]}$$

$$\mathcal{E}[(wcm\ ([v_{k1}\ v_{v1}]\ \ldots)\ (wcm\ ([v_{k2}\ v_{v2}])\ e))] \Rightarrow \mathcal{E}[(wcm\ \mathsf{merge}[([v_{k1}\ v_{v1}\ ]\ \ldots\ [v_{k2}\ v_{v2}\ ]\ \ldots)]\ e)] \quad \text{[wcm (merge)]}$$

where $\mathcal{E}\ \neq\ \mathcal{E}'[(wcm\ ([v'_k\ v'_v]\ \ldots)\ [])]$

$$\mathcal{E}[(wcm\ ([v_k\ v_v]\ \ldots)\ v_{ret})] \Rightarrow v_{ret} \quad \text{[wcm (return)]}$$

where $\mathcal{E}\ \neq\ \mathcal{E}'[(wcm\ ([v'_k\ v'_v]\ \ldots)\ [])]$

$$\mathcal{E}[(ccm\ v_k\ \ldots)] \Rightarrow \mathcal{E}[\mathsf{extract}[\mathcal{E}, v_k\ \ldots]] \quad \text{[ccm]}$$

$$\mathsf{merge}[([v_{ki}\ v_{vi}]\ \ldots)] = ([v_k\ \omega(v_k)]\ \ldots)$$

where $\omega = \emptyset[v_{ki}\ \mapsto\ v_{vi}]\ \ldots$

$v_k = dom(\omega)$

$$\mathsf{extract}[[], v_t\ \ldots] = (\text{NIL})$$

$$\mathsf{extract}[(v\ \ldots \mathcal{E}), v_t\ \ldots] = \mathsf{extract}[\mathcal{E}, v_t\ \ldots]$$

$$\mathsf{extract}[(wcm\ ([v_k\ v_v]\ \ldots)\ \mathcal{F}), v_t\ \ldots] = (\text{CONS}\ (\text{VECTOR}\ v_i\ \ldots)\ \mathsf{extract}[\mathcal{F}, v_t\ \ldots])$$

where $v_i = (\text{SOME}\ v_v)$ for each $v_k = v_t$ and (NONE) otherwise

Figure 3. Reductions

$$
\begin{array}{rcl}
e & ::= & a \\
& | & (w\ \ldots e) \\
& | & (letrec\ ([\sigma\ v])\ e) \\
& | & (match\ w\ l\ \ldots) \\
& | & (call/cc\ w) \\
l & ::= & [(K\ x\ \ldots)\ \Rightarrow\ e] \\
& | & [else\ \Rightarrow\ e] \\
a & ::= & w \\
& | & (K\ a\ \ldots) \\
w & ::= & v \\
& | & x \\
v & ::= & \sigma \\
& | & (\lambda\ (x\ \ldots)\ e) \\
& | & (\overline{\lambda}\ (x\ \ldots)\ e) \\
& | & (K\ v\ \ldots) \\
& | & \kappa.E \\
\Sigma & ::= & \emptyset \\
& | & \Sigma[\sigma\ \mapsto\ v] \\
\mathcal{E} & ::= & [] \\
& | & (v\ \ldots\ \mathcal{E})
\end{array}
$$

Figure 2. SL grammar

matching; [match (next)] handles unused cases; [match (else)] handles the default case of a pattern matching. The [letrec], [σ + apply], and [σ + match] rules specify the semantics of $letrec$. 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 de-referencing is performed when a store reference appears in an application ([σ + apply]) or in a match expression ([σ + match]). The final rules for continuations are standard.

4.2 Target Language

The target language (Figure 4) is similar to the source language, except that $\overline{\lambda}$ and $call/cc$ are removed, while wcm, ccm, and $abort$ have been added.

The semantics (top and bottom of Figure 3) is similar to the source language's as well, except that because of continuation marks the evaluation contexts are structured to avoid adjacent marks.

There are four new reduction rules that make use of a few meta-functions (Figure 3, bottom). The first rule implements $abort$ by abandoning the context. The second implements

$$
\begin{array}{rcl}
e & ::= & a \\
& | & (w\ \ldots e) \\
& | & (letrec\ ([\sigma\ v])\ e) \\
& | & (match\ w\ l\ \ldots) \\
& | & (wcm\ ([w\ w]\ \ldots)\ e) \\
& | & (ccm\ w\ \ldots) \\
& | & (abort\ e) \\
l & ::= & [(K\ x\ \ldots)\ \Rightarrow\ e] \\
& | & [else\ \Rightarrow\ e] \\
a & ::= & w \\
& | & (K\ a\ \ldots) \\
w & ::= & v \\
& | & x \\
v & ::= & \sigma \\
& | & (\lambda\ (x\ \ldots)\ e) \\
& | & (K\ v\ \ldots) \\
\Sigma & ::= & \emptyset \\
& | & \Sigma[\sigma\ \mapsto\ v] \\
\mathcal{E} & ::= & (wcm\ ([v\ v]\ \ldots)\ \mathcal{F}) \\
& | & \mathcal{F} \\
\mathcal{F} & ::= & [] \\
& | & (v\ \ldots\ \mathcal{E})
\end{array}
$$

Figure 4. TL grammar

the tail-calling semantics of wcm, where adjacent marks are collapsed and overridden. The meta-function merge replaces outer marks by inner marks when their keys are equal. The rule is applied whenever the structured evaluation context forces adjacent wcms to be treated as a redex. The third new rule returns final values from wcm bodies. These two rules are deterministic because the side-condition on $\mathcal{E}$ forces [wcm (merge)] to be applied from outside to inside. The rule for ccm uses the meta-function extract to extract the marks from the context in the format:

$$
\begin{array}{rcl}
marks & ::= & (\text{NIL}) \\
& | & (\text{CONS}\ markset\ marks) \\
markset & ::= & (\text{VECTOR}\ maybe\ \ldots) \\
maybe & ::= & (\text{NONE}) \\
& | & (\text{SOME}\ v)
\end{array}
$$

4.3 Continuation Mark Transformation

Translating from SL to TL follows Pettyjohn et al. (2005) with the *CMT* (*C*ontinuation *M*ark *T*ransformation) shown

in Figure 5. The translation when applied to full expressions ($\mathsf{CMT}_e[]$) first decomposes a term into a context and a redex by the grammar

$$
\begin{aligned}
r &::= (w\ \ldots\ a) \\
&\mid (letrec\ ([\sigma\ w])\ e) \\
&\mid (match\ w\ l\ \ldots) \\
&\mid (call/cc\ w) \\
\underline{\mathcal{E}} &::= [] \\
&\mid (w\ \ldots\ \underline{\mathcal{E}})
\end{aligned}
$$

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 a single redex r and context $\underline{\mathcal{E}}$.*

The translation relies on particular definitions for **resume** and **call/cc**, given in Figure 6. The translation rules are mostly straightforward. Continuation values are converted directly into functions that apply **resume**, but the captured SL context must be translated into a set of marks that match the format expected by **resume**. This is handled by the $\mathcal{E}$ case of the $\mathsf{CMT}[]$ meta-function. Safe λs are translated by translating their bodies and annotating the body context with a safety mark, as discussed in Section 3.1. In contrast, the bodies of $\overline{\lambda}$s are **not** translated. (This makes the translation ill-defined on $\overline{\lambda}$s that embed other $\overline{\lambda}$s, continuation values, or $call/cc$. We discuss this below.)

The next interesting rule is for function application ($(w...a)$); in this rule, the call has to be explicitly marked as unsafe in case the function applied is an $\overline{\lambda}$. Direct calls to $call/cc$ are replaced with calls to the TL implementation of **call/cc**. Finally, context compositions ($(w...\underline{\mathcal{E}})$) are replaced with applications where the syntactic continuations are duplicated into the (SQUARE) marks to communicate with the implementation of **call/cc**.

We now turn our attention to the definitions of **resume** and **call/cc** (Figure 6). **resume** works by reconstructing the evaluation context from the list of (SQUARE) marks, including restoring the marks themselves. **call/cc** works by extracting the (SAFE?) marks and calls **all-safe?** with that list. **all-safe?** ensures that every frame is marked as safe. **call/cc** then extracts the (SQUARE) marks and constructs a continuation value (a λ that calls **resume**) before giving it to its argument, f.

4.4 Correctness

We define "reasonable SL programs" as those where the bodies of $\overline{\lambda}$s are syntactically TL expressions and do not override the standard library functions **resume** and **call/cc**, or the functions they rely on. This is a low standard of reasonableness; we are not hiding triviality behind this definition. Furthermore, these constraints ensure that SL programs more closely model the reality that unsafe programs will not include fragments of safe programs.

```
(letrec
 ([resume
   (λ (l v)
    (match l
     [(NIL) ⇒ v]
     [(CONS cm l) ⇒
      (match cm
       [(VECTOR mk) ⇒
        (match mk
         [(SOME k) ⇒
          (k (wcm ([(SQUARE) k]) (resume l v)))])])])))]
  [all-safe?
   (λ (marks)
    (match marks
     [(NIL) ⇒ (TRUE)]
     [(CONS ml) ⇒
      (match m
       [(VECTOR s) ⇒
        (match s
         [(SOME v) ⇒
          (match v
           [(TRUE) ⇒ (all-safe? l)]
           [(FALSE) ⇒ (FALSE)])])])])))]
  [call/cc
   (λ (f)
    ((λ (is-safe?)
      (match is-safe?
       [(TRUE) ⇒
        ((λ (k) (f k))
         ((λ (m)
            (λ (x) (abort (resume m x))))
          (ccm (SQUARE)))]
       [(FALSE) ⇒
        (abort (UNSAFE CONTEXT))]))
     (all-safe? (ccm (SAFE?)))))])
 ...)
```

Figure 6. CMT library

Theorem 1. *For every reasonable SL program e, if $\emptyset, e$ reduces to a value, v, then $\mathsf{CMT}_e[e]$ reduces to $\mathsf{CMT}_v[v]$ or* (UNSAFE CONTEXT).

This theorem states that the transformation preserves the meaning of the program or errors because the program captures a continuation in an unsafe context.

We define "pure SL programs" as those that do not contain $\overline{\lambda}$s and that do not override the standard library functions **resume**, **call/cc**, or **all-safe?**. This corresponds to real world programs that do not interact with third-party libraries.

$$
\begin{aligned}
\mathsf{CMT}_v[\kappa.\mathcal{E}] &= (\lambda\ (x)\ (abort\ (\textbf{resume}\ \mathsf{CMT}_{\mathcal{E}}[\mathcal{E}]\ x)) \\
\mathsf{CMT}_v[(\lambda\ (x\ \ldots)\ e)] &= (\lambda\ (x\ \ldots)\ (wcm\ ([(\textsc{safe?})\ (\textsc{true})])\ \mathsf{CMT}_e[e])) \\
\mathsf{CMT}_v[(\overline{\lambda}\ (x\ \ldots)\ e)] &= (\lambda\ (x\ \ldots)\ e) \\
\mathsf{CMT}_v[\sigma] &= \sigma \\
\mathsf{CMT}_v[(K\ v\ \ldots)] &= (K\ \mathsf{CMT}_v[v]\ \ldots)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{CMT}_w[x] &= x \\
\mathsf{CMT}_w[v] &= \mathsf{CMT}_v[v]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{CMT}_a[w] &= \mathsf{CMT}_w[w] \\
\mathsf{CMT}_a[(K\ a\ \ldots)] &= (K\ \mathsf{CMT}_a[a]\ \ldots)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{CMT}_l[[(K\ x\ \ldots)\ \Rightarrow\ e]] &= [(K\ x\ \ldots)\ \Rightarrow\ \mathsf{CMT}_e[e]] \\
\mathsf{CMT}_l[[else\ \Rightarrow\ e]] &= [else\ \Rightarrow\ \mathsf{CMT}_e[e]]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{CMT}_r[(w\ \ldots\ a)] &= (wcm\ ([(\textsc{safe?})\ (\textsc{false})])\ (\mathsf{CMT}_w[w]\ \ldots\ \mathsf{CMT}_a[a])) \\
\mathsf{CMT}_r[(letrec\ ([\sigma\ w]\ \ldots)\ e)] &= (letrec\ ([\sigma\ \mathsf{CMT}_w[w]]\ \ldots)\ \mathsf{CMT}_e[e]) \\
\mathsf{CMT}_r[(call/cc\ w)] &= (\textbf{call/cc}\ \mathsf{CMT}_w[w]) \\
\mathsf{CMT}_r[(match\ w\ l\ \ldots)] &= (match\ \mathsf{CMT}_w[w]\ \mathsf{CMT}_l[l]\ \ldots)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{CMT}_{\underline{\mathcal{E}}}[[]] &= [] \\
\mathsf{CMT}_{\underline{\mathcal{E}}}[(w\ \ldots\ \underline{\mathcal{E}})] &= (\kappa\ (wcm\ ([(\textsc{square})\ \kappa])\ \mathsf{CMT}_{\underline{\mathcal{E}}}[\underline{\mathcal{E}}])) \\
&\quad \text{where } \kappa\ = (\lambda\ (x)\ (\mathsf{CMT}_w[w]\ \ldots\ x))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{CMT}_e[a] &= \mathsf{CMT}_a[a] \\
\mathsf{CMT}_e[\underline{\mathcal{E}}[r]] &= \mathsf{CMT}_{\underline{\mathcal{E}}}[\underline{\mathcal{E}}][\mathsf{CMT}_r[r]]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{CMT}_{\mathcal{E}}[[]] &= (\textsc{nil}) \\
\mathsf{CMT}_{\mathcal{E}}[(v\ \ldots\ \mathcal{E})] &= (\textsc{cons}\ (\textsc{vector}\ (\textsc{some}\ (\lambda\ (x)\ (\mathsf{CMT}_v[v]\ \ldots\ x))))\ \mathsf{CMT}_{\mathcal{E}}[\mathcal{E}])
\end{aligned}
$$

Figure 5. CMT definition

Theorem 2. *For every pure SL program e, if $\emptyset, e$ reduces to a value, v, then $\mathsf{CMT}_e[e]$ reduces to $\mathsf{CMT}_v[v]$.*

5. Real World Issues

Once third-party higher-order library functions can be called with arguments that capture continuations, there are still a few practical problems and extensions that can be made to make real Web programming easier and scalable. In this section, we discuss a few.

5.1 Native Function Interfaces

The two-state solution allows us to write code like

```
(serial→native
 (build-list
  how-many
  (λ (i)
   (native→serial
    (web-prompt
     "Please provide number #~a:\n" (add1 i))))))
```

where the native part of the continuation is captured separately and stored on the server. This increases expressivity, but it is overly verbose and forces programmers to think about how functions are implemented whenever they are used.

This additional cognitive task is valuable, because server resources are scarce, but it is often too inconvenient for the prototyping stage. For that reason, the Racket Web framework supports the *define-native* form for creating a wrapper. *define-native* is given a new identifier to bind, a specification of which higher-order arguments may capture continuations, and a native implementation. For example, the wrapper for *build-list* is defined:

```
(define-native (build-list/safe _ ho) build-list)
```

Programmers can then write:

```
(build-list/safe
 how-many
 (λ (i)
  (web-prompt
   "Please provide number #~a:\n" (add1 i))))
```

This simple macro is incredibly effective at reducing the burden on Web programmers that seek scalability and convenience.

5.2 Managing Native Continuations

The fundamental problem for scalability with native continuations is that they cannot be removed from memory or reclaimed because references to them may be stored by users in untraceable ways. For example, users can bookmark a page or otherwise remember a URL indefinitely, yet expect the page to be available whenever they request it. This creates a serious resource management problem for continuation-based Web applications that mirrors the "session state management" problem of traditional Web applications. The standard scalability technique is to time out sessions after periods of inactivity and provide seamless ways for users to authenticate and restore their session.

The two-state solution does not remove the server-state management problem, but it does minimize it by storing only the smallest necessary delimited continuations. Therefore, we have to have some policy for managing the server resources consumed by native components in the two-state solution.

The Racket Web framework associates a *manager* with each Web program. This manager is essentially a hash table mapping unique, serializable identifiers with opaque, non-serializable values, like continuations. When each value is stored, it is given an *expiration-handler* that is returned when the manager reclaims the value's resources. One other complicating factor is that the values are stored indexed by *instance* (distinct invocation of the application); this is useful because it allows an instance to explicitly remove values from the managers (such as on logout).

Programmers often write their own custom managers, but there are a few standard managers.

The *null* manager never uses server resources and always returns the expiration handler. This manager is the default for scalable Web applications and must be explicitly removed. It is useful to ensure that native continuations are not accidentally stored.

The *timeout* manager implements the standard timeout policy: instances and values are reclaimed after a configurable number of seconds have elapsed since their last access. This is very problematic for large deployments because the behavior does not change in response to observed use.

The *LRU* manager implements a more sophisticated technique: each value has a "life count" that is initialized to $start$, if it ever reaches 0, then the value is reclaimed. The life count is decremented by one whenever the reclamation routine is run. The reclamation routine runs every $collect$ seconds and every $check$ seconds when $check?$ returns true. These four parameters ($start$, $collect$, $check$, and $check?$) allow a wide array of management policies. A common policy configuration is the "threshold" policy: $start$ is 24, $collect$ is 10 minutes, $check$ is 5 seconds, and $check?$ determines if the memory used by the manager is greater than $threshold$ megabytes. This ensures that values are available for at most four hours and at least two minutes. It also ensures that a particular memory limit is never exceeded for long, because once it is, old values are rapidly reclaimed until the problem is defeated.

We find that in practice the *LRU* threshold policy is essential for running scalable and reasonable Web applications. This practical finding is backed up by experimental results discussed in Section 6.

5.3 Soft State

Managers allow stateless Web applications to interact seamlessly with server state when it is inconvenient or impossible to serialize that state to the Web clients. Once this mechanism is in place, it is tempting to use it to access other kinds of state, such as data that is too large to transmit or too sensitive to put in the hands of the untrusted. For example, it is common for Web applications to make use of "soft state": state that is stored on the server, but can be recomputed when necessary so it is allowed to be forgotten at any time. A common use is a cache of a user's information from a database.

Our implementation of soft state provides two calls: (*soft-state expr*), which creates a piece of soft state that has the same value as *expr*, but *expr* may be evaluated many times to compute it; and (*soft-state-ref ss*), which extracts the value of the soft state bound to *ss* and may evaluate the soft state expression to compute it.

The implementation uses managers to store the value of the soft-state such that the expiration handler recomputes the value if the manager reclaims it. The soft-state serializes to the manager's identifier and a serializable thunk that recomputes the value, rather than the value itself, ensuring that the soft-state can be safely serialized to the client and recomputed as needed. This implementation is entirely local and requires no additional cooperation from the rest of the Web application.

6. Evaluation

The formal model of Section 4 helps to establish the soundness of the two-state implementation technique, but it cannot tell us anything about the savings or costs of its use. Since the use of the two-state solution cannot be divorced from the use of server-side state managers, we evaluate their performance as well.

6.1 Managers

We measured the performance of server-side state managers on two metrics.

First, we measured what percentage of all continuations ever stored on the server were available as the application ran. If this percentage is high, then users will not follow links or bookmarks and find the page unavailable. If this percentage is low, then they will, unless the application has established other ways to restore a user's session. It is important to realize, however, that users may not actually care about one of these expired values. For example, after users have totally finalized their online purchase, they have no need to access the continuation that sets the shipping address. Thus this percentage establishes an imperfect bound on user happiness; a low percentage does not, in practice, mean the application is unusable.

Second, we measured the total memory used by the server. If this is high or is proportional to server use, then the server is unlikely to be scalable because memory will invariably be exhausted during periods of high activity.

All measurements of managers are based on logs of the use of the CONTINUE conference manager by an elite conference during the periods of its highest activity: just before the paper and review submission deadlines. These periods of activity were characterized as a six hour window with 30 active users at any time performing a total of about 3,000 interactions distributed throughout the entire period.

In these experiments, the default *LRU* manager was configured with a threshold that limits the number of continuations to approximately 2000 and the *timeout* manager was used with a one hour, two hour, and four hour timeout.

These experiments do not constitute stress tests; they are just enough activity to identify and compare the trends of each manager.

Continuation Availability. Figure 7 (top) presents the percentage of available continuations that were available to users at a number of points in the run. The X-axis shows the progress of time, while the Y-axis show the percentage of all continuations ever stored that were available for invocation by users. Naturally, every manager starts off with 100% of the continuations available, but the *timeout* managers slowly lose availability as more continuation timeouts expire. In contrast, the *LRU* manager delivers 100% availability approximately as long as the 2-hour *timeout* manager, but loses availability more slowly before leveling out.

Memory Usage. Figure 7 (bottom) presents the amount of memory used by the managers at each sampled point in the run. The graphs show the usage of the *LRU* manager and one of the *timeout* managers. The first row shows the 1-hour and 2-hour *timeout* managers. The second row shows the 4-hour and 6-hour *timeout* managers. (The oscillation in these graphs are a side-effect of the sampling thread competing with the manager thread, because Racket does not have truly concurrent threads. Since the higher timeout threads do less work, they compete less and therefore the sampler takes more samples.)

The *LRU* manager uses consistently less memory, despite giving better availability than the 1- and 2-hour *timeout* managers always and the 4-hour manager after the timeout. The 6-hour *timeout* manager (not shown on top because it always has 100% during the 6-hour experiment) shows the memory cost of never expiring server-side continuations.

These experiments show that while the *timeout* manager can guarantee 100% availability before the timeout, the *LRU* provides strong memory bounds without sacrificing availability too severely. For example, the 4-hour *timeout* manager has 100% availability until 4 hours, whereas the *LRU* manager only has 50%; but after 2 more hours, both managers have the same availability.

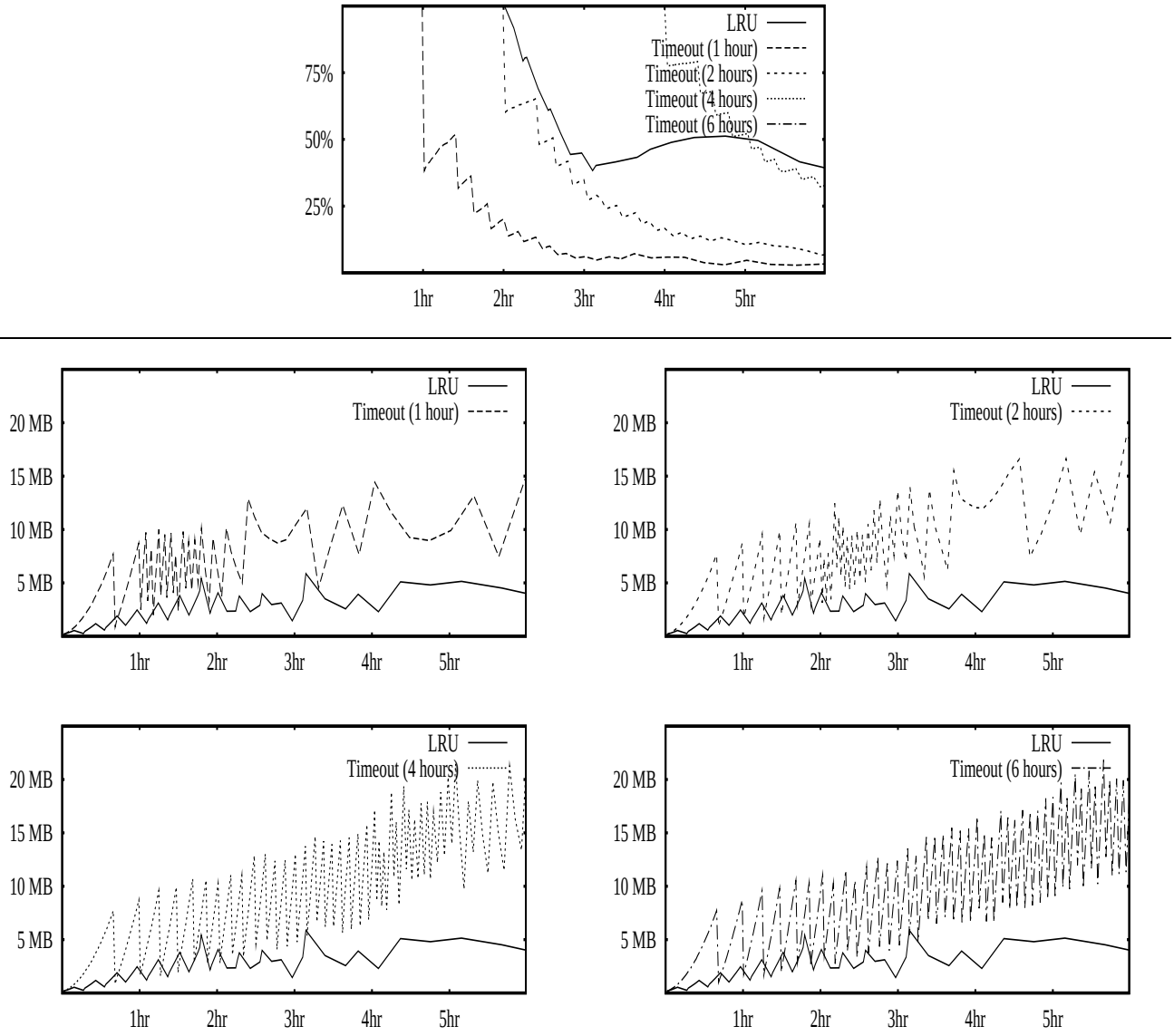

Figure 7. Memory Usage of Different Managers

6.2 Two-State Solution

We measured four different comparisons between native continuations, two-state continuations, and serializable continuations. First, we repeated the manager tests after specializing them for each of these techniques, then we performed two separate timing tests.

Continuation Availability. We ran a native and two-state version of the same Web application through the workload above. The average continuation size of the native version was approximately 2KB, whereas the server-part of the two-state continuations were on average only 512B. The *LRU* manager was used for both versions and was configured with a memory threshold of 4MB.

Figure 8a presents the availability of continuations during the test. It verifies our expectation that the two-state solution provides more access to server state, by reducing the size of each continuation, but after the threshold is reached, the two have similar performance.

It does not make sense to compare these to serializable continuations, because all serializable continuations are always available.

Memory Usage. Figure 8b shows the memory usage for the same experiment. It verifies that the native continuations

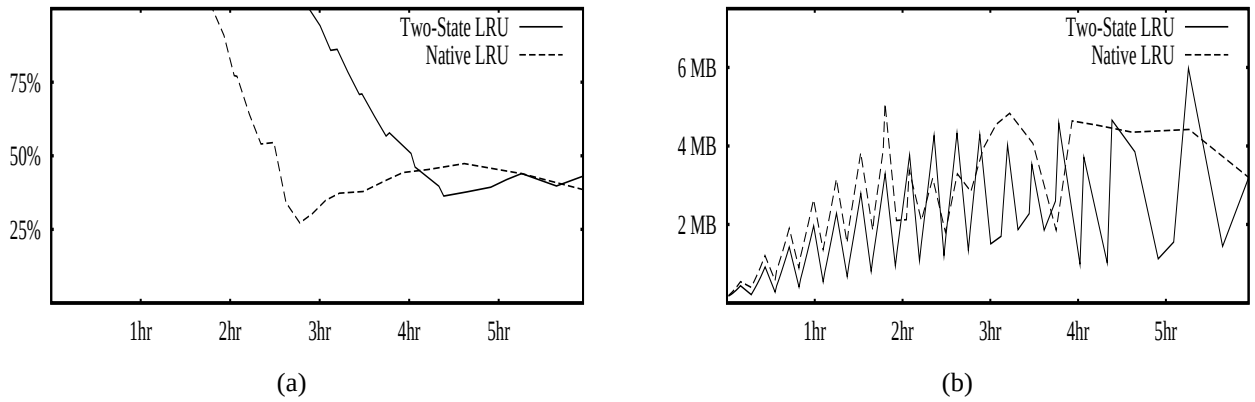

Figure 8. Native and Two-State Continuation Availability and Memory Usage

take more space on the server, but that the *LRU* manager controls memory usage regardless of what the server state is.

It may seem surprising that the two-state solution sometimes uses more memory than the native solution, such as between two and three hours. This has to do with when reclamation occurs. Since the native solution exhausts the memory limit faster, it causes a reclaiming just before the two-hour mark. At that point the two-state solution uses more memory because it hasn't expired anything. Once it does (around three hours), it use less memory than native continuations, until the process repeats just before four hours.

It does not make sense to compare these to serializable continuations, because serializable continuations do not consume server memory resources.

Compilation Time. The two-state solution relies on a complex transformation that intuitively increases compilation time relative to using native continuations. We measured this by compiling three versions of the our example application from Section 2 one hundred times and averaging the compilation times.

Native	661	ms
Two-State	1007	ms
Serial	1048	ms

Native compilation is almost fifty percent faster than either transformation-based compilations, while the two-state compilation is only slightly faster—presumably because there is less code to compile since third-party higher-order library functions are not examined.

In our experience, the increased compilation time for using non-native continuations is quickly noticed by programmers but easy to ignore during development.

Execution Time. The two-state solution uses a combination of serializable closures and native delimited continuations to simulate full continuations that are implemented natively. It is natural to assume that the Racket compiler and runtime perform better with native continuations than simulated continuations, because they have been optimized over many years, while these simulated continuations have not. This assumption is true for continuation *invocation*, but false for continuation *capture*.

We created a micro-benchmark that captures 200 continuations and averaged 1,000 runs per implementation.

Native	1.5975	ms
Two-State	0.1528	ms
Serial	0.6744	ms

It is ten times cheaper to capture our simulated continuations. The intuitive explanation is that the native continuation capture has to copy large parts of the stack and check if previous stack tails have already been captured so they may share and thus reduce the overall space consumed by continuations. A tail-sharing continuation representation reduces space consumption in most Web applications because many intermediate computations return to the same points. For example, each stage of a purchase returns to the same "Shipping Confirmation" page generator.

The simulated continuation capture is faster because it only inspects the stack and copies a few small pointers that were set up when the continuation component was added in a continuation mark. There would be no space savings if we were to implement tail sharing, because these continuations are not stored on the server and the entire continuation must be serialized to the client.

A different implementation of native continuation capture may not have this performance difference (Hieb et al. 1990; Clinger et al. 1999). The difference that exists, however, can be significant because most Web applications capture many more continuations than they invoke, because most pages have many links and the majority are not chosen.

We created a micro-benchmark that captures *and invokes* 200 continuations and averaged 1,000 runs per continuation implementation.

Native	1.7609	ms
Two-State	49.6306	ms
Serial	13.8949	ms

It is almost *thirty* times more expensive to invoke our simulated continuations than native continuations, especially when we also must install a continuation prompt and capture and invoke a native delimited continuation.

Despite these micro-benchmarks, none of these differences are actually observable by end users of Web applications, given the already long network delays users are accustomed to. A 50ms delay is dwarfed by the typical network latency over the Internet.

Ease of Use. A frustrating aspect of the two-state solution is that foreign code must be explicitly identified and call must be wrapped in serial→native and the higher-order arguments must be wrapped in native→serial. This limits the ease of porting a code-base from the purely native continuation technique to the two-state solution.

Fortunately, it is safe to insert paired calls to serial→native and native→serial anywhere. Therefore, we have experimented with automatically inserting them around every call to a function from another module—the only functions that *could* be foreign. We measured two situations where this could impose costs.

First, we tested when the possibly foreign function doesn't actually have higher-order arguments, so there is no reason to use serial→native. We timed and averaged 20,000 calls with this profile.

Without Wrapping	0.0843	ms
Wrapping	0.0860	ms

There is only about a 2% performance penalty and the absolute values are tiny. This essentially is a measurement of the cost of serial→native.

Second, we tested when the possibly foreign function has higher-order arguments, but they don't capture continuations. We timed and averaged 20,000 calls with this profile.

Without Wrapping	0.0841	ms
Wrapping	0.0905	ms

There is about an 8% performance penalty for extraneous uses of both serial→native and native→serial.

These experiments suggest that it is not prohibitive to runtime performance to automatically use the two-state solution. However, we are philosophically opposed to obscuring when server state is used from programmers and prefer that this decision be made explicitly with an idea to the overall scalability of the Web application.

7. Conclusion

We presented an implementation technique that allows scalable and stateless Web programs written in direct style to make use of third-party higher-order library functions with higher-order arguments that capture continuations and cause user interaction. We have presented a formal model of a portion of this implementation technique. We have discussed extensions necessary for real deployment and additional facilities easily provided by this infrastructure. We have evaluated this work and found that it increases scalability compared to purely native continuations and reduces the burden of reimplementation compared to purely serial continuations. Overall, the two-state solution works.

This work relies on the state-of-the-art in stack inspection and manipulation—continuation marks *and* delimited continuations—therefore it is only directly applicable to programming languages in the Racket family.

Acknowledgments We thank Matthew Flatt for his superlative work on Racket, where our implementation lives. We thank Robby Findler for his fabulous work on Redex, where our model lives. We thank Casey Klein for his work on the randomized testing facility of Redex, which aided our model development effort.

References

Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, September 1995.

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

William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher Order and Symbolic Computation*, 12(1):7–45, 1999.

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.

R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, 2007.

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

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 and PLT. Reference: Racket. Technical Report PLT-TR2010-1, PLT Inc., June 7, 2010. `http://racket-lang.org/tr1/`.

Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *International*

Conference on Functional Programming, 2007. URL `http://www.cs.utah.edu/plt/delim-cont/`.

Martin Gasbichler and Michael Sperber. Final shift for call/cc:: direct implementation of shift and reset. *SIGPLAN Notices*, 37 (9):271–282, 2002.

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

Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990.

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

Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, pages 190–203, 1985.

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. Automatically restful web applications or, marking modular serializable continuations. In *International Conference on Functional Programming*, 2009.

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.