Continuations from Generalized Stack Inspection *

Greg Pettyjohn
Northeastern University

John Clements
Northeastern University

Joe Marshall
Northeastern University

Shriram Krishnamurthi
Brown University

Matthias Felleisen
Northeastern University

Abstract

Implementing first-class continuations can pose a challenge if the target machine makes no provisions for accessing and re-installing the run-time stack. In this paper, we present a novel translation that overcomes this problem. In the first half of the paper, we introduce a theoretical model that shows how to eliminate the capture and the use of first-class continuations in the presence of a generalized stack inspection mechanism. The second half of the paper explains how to translate this model into practice in two different contexts. First, we reformulate the servlet interaction language in the PLT Web server, which heavily relies on first-class continuations. Using our technique, servlet programs can be run directly under the control of non-cooperative web servers such as Apache. Second, we show how to use our new technique to copy and reconstitute the stack on MSIL.Net using exception handlers. This establishes that Scheme's first-class continuations can exist on non-cooperative virtual machines.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives; H.3.4 [*Information Storage and Retrieval*]: Systems and Software—World Wide Web (WWW); I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program transformation; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics

General Terms Languages, Theory

Keywords A-normal form, continuation-passing style, stack inspection, Web programming, continuations, defunctionalization, Scheme

1. Motivation: Continuations, VMs, and the Web

When an interactive Web program issues a query, a non-local transfer of control takes place. It is now the user (possibly an intelligent agent) who is in charge. This user may decide to respond to the query once, twice or many times (or not at all), thus re-launching the rest of the program's computation once, twice, or many times. Many people have therefore concluded that a language with first-class continuations (such as Scheme [19]) is a strong match for implementing this kind of interactive program [11, 18, 26, 33].

Over the past several years, we have explored the continuation model of Web programming in two different directions. Initially, we designed, implemented, and evaluated a Web server that implements the run-time primitives for Web interactions via Scheme's first-class continuations [14]. The investigation validated that this approach is suitable for a variety of situations and even led to a commercial product [20]. Sadly, only programs running on a custom Web server can benefit from this approach.

To address this concern, we also experimented with a variant of the continuation-passing transformation (CPS) for automatically restructuring interactive programs for the Web [13, 21]. In principle, this transformation can be used with a wide range of programming languages and should therefore help Web programmers in many situations. The problem, however, is that the transformation affects the entire program. Since modern Web applications are often written in multiple languages, it is nearly impossible to perform a whole-program transformation of them. Worse, CPS requires tail-call optimization or, in its absence, trampolines, a technique due to Jon L. White [personal communication June 2005], which can be much more expensive. It is therefore economically impossible to use this idea with existing languages and run-time libraries. In short, we are left with the challenge of equipping conventional programming languages with the capabilities of grabbing and reinstalling a continuation even if these languages' run-time organizations do not support such actions.

The very same problem comes up in a different context: the implementation of languages with continuations on virtual machines such as Sun's JVM [34] or Microsoft's CLR [22]. Mirroring traditional programming languages, these machines do not provide instructions for installing and saving the run-time stack. Usually, Scheme-on-VM implementors give up on first-class continuations [1, 2], or they allocate the control stack in the heap of the machine [23]. As Bres, et al. [2, section 2] point out, however, with this second strategy, "it might be expected that . . . JITs are far less efficient on codes that manages their own stack" [sic]. Furthermore, allocating the stack on the heap effectively disguises the stack, hiding it from the rich set of programming tools such as steppers, debuggers, and profilers, as well as contemporary security managers, which expect to find run-time information on the stack.

In this paper, we present a solution to this dilemma, first in the context of a theoretical model and then in the context of two pro-

* This research has been supported by NSF awards to Felleisen and Krishnamurthi as well as a Microsoft donation to Felleisen.

```
(define (fact n)
  (if (= n 0)
      (begin
        (display (c-c-m))
        1)
      (w-c-m n (* n (fact (- n 1))))))
(fact 3)
↦
console output: (1 2 3)
computed value: 6
```

Figure 1. A factorial function with continuation marks

```
(define (fact-tr n a)
  (if (= n 0)
      (begin
        (display (c-c-m))
        a)
      (w-c-m n (fact-tr (- n 1) (* n a)))))
(fact-tr 3 1)
↦
console output: (1)
computed value: 6
```

Figure 2. A tail-calling factorial function with continuation marks

totype implementations. The core idea is to translate Scheme programs with call/cc into a language with a generalized stack inspection mechanism. For our theoretical model and for one of our prototypes, we use PLT Scheme's continuation mark mechanism; for the other prototype, we show how exception handlers and exception throws can collaborate to simulate continuation marks. The resulting transformation is less radical than CPS, and it is thus natural to run the resulting program on the natively available stack.

2. Continuation Marks

Most programming languages and programming environments include mechanisms for manipulating the stack in some form or another. Java, for example, implements a form of security via stack inspection [35]. Programming environments such as Visual Studio have privileged access to the stack for various debugging tasks. The most ubiquitous examples of such mechanisms, though, are exception signalers and handlers. They erase portions of the control stack and transfer control to exception handlers in a non-local manner.

MzScheme [9], the implementation language of the DrScheme programming environment [7], provides a novel abstraction of all these mechanisms: continuation marks [4]. Roughly speaking, continuation marks support a powerful form of stack inspection generalizing Java's mechanism with the same name. Using the w-c-m language form (short for "with continuation marks"), a programmer can attach values to the control stack during the execution of a program. Later, the stack-walking primitive c-c-m ("current continuation marks") can retrieve these values from the stack.

To preserve the spirit of Scheme, attaching continuation marks to the stack does not interfere with Scheme's tail-calling requirement. Furthermore, MzScheme's marks are parameterized by keys. By choosing fresh keys, programmers can ensure that adding marks to a computation does not affect the result of the computation, enabling the use of marks for multiple purposes. In particular, MzScheme implements exceptions and a trace facility with continuation marks, while the tools of DrScheme rely on them to implement a stepper, a debugger, and a performance profiler.

The two programs in figures 1 and 2 illustrate how a programmer might use the w-c-m and c-c-m constructs to instrument functions. Both definitions implement factorial; both mark the continuation at the recursive call site; and both report the continuation-mark list before returning. The one in figure 1 is properly recursive, while the one in figure 2 is tail-recursive. For the properly recursive program, the console output shows that the continuation contains three mark frames. For the tail-recursive variant, only one continuation mark remains; the others have been overwritten during evaluation.

$$
\begin{array}{rcl}
e & ::= & a \\
 & | & (\overline{w}\ e) \\
 & | & (\textbf{letrec}\ (\overline{[\sigma\ v]})\ e) \\
 & | & (\mathsf{call/cc}\ w) \\
 & | & (\mathsf{case}\ w\ \overline{l}) \\
l & ::= & (K\ \overline{x}) \Rightarrow e \\
a & ::= & w \mid (K\ \overline{a}) \\
w & ::= & v \mid x \\
v & ::= & (\lambda(\overline{x})\ e) \mid (K\ \overline{v}) \mid \kappa.\mathcal{E} \mid \sigma \\
 & & \\
x & \in & \text{Variables} \\
\sigma & \in & \text{References} \\
 & & \text{where Variables} \cap \text{References} = \emptyset \\
 & & \\
\mathcal{E} & ::= & [\ ] \mid (\overline{v}\ \mathcal{E}) \\
\Sigma & ::= & \emptyset \mid \Sigma[\sigma \mapsto v]
\end{array}
$$

($\overline{X}$ denotes zero or more occurrences of X)

Figure 3. SL Syntax

3. Continuations from Continuation Marks

Equipped with a basic understanding of MzScheme's continuation marks, we can now show how to use this generalized stack inspection mechanism to eliminate call/cc from Scheme programs. We present the idea in two steps. The first step is to translate programs with call/cc into semantically equivalent programs that use continuation marks to store functional representations of continuations. The second step replaces the marks with structures, using a variant of Reynolds' defunctionalization [27].

3.1 The Source Language

The language in figure 3, dubbed SL for *s*ource *l*anguage, is a modified version of A-Normal form (ANF) [8]. It uses λ instead of **let**. Furthermore, we allow applications of arbitrary length. The language is extended with call/cc, **letrec** and algebraic datatypes. The latter are needed in the target language for the operational semantics and are used during defunctionalization, the last step of our translation. For consistency they are also included in the source language.

Instances of algebraic datatypes are constructed with constructors (K, K^m) and destructured with case. We leave the actual set of constructors unspecified, though we assume it contains the standard list constructors cons and nil. For convenience, we use a shorthand for lists, where (*list* $e_0\ e_1\ \ldots$) stands for the aggregate construction (cons e_0 (*list* $e_1\ \ldots$)) and (*list*) stands for the empty list.

The operational semantics is specified via the rewriting system in figure 4. The first rule is the standard β_v-rewriting rule for call-by-value languages [24]. The second handles the destructuring of algebraic datatypes.

Rules (3, 4, 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 variables bound in lambda expressions [6]. Rule 5 specifies how bindings are established by **letrec**. Furthermore, to simplify the syntax for evaluation contexts, store references are treated as values, and dereferencing is performed only when a store reference appears in application position (rule 4) or in the test position of a case expression (rule 5).

$$\mathcal{E} \quad ::= \quad (\text{w-c-m } v\ \mathcal{F}) \mid \mathcal{F} \text{ where } \mathcal{F} ::= [\,] \mid (\overline{v}\ \mathcal{E})$$

$$\begin{array}{rcl}
\Sigma/\mathcal{E}[((\lambda(\overline{x})\ e)\ \overline{v})] & \overset{(1)}{\rightarrow}_{\text{TL}} & \Sigma/\mathcal{E}[e\overline{[x \mapsto v]}] \\
\Sigma/\mathcal{E}[(\text{case } (K\ \overline{v})\ \overline{l})] & \overset{(2)}{\rightarrow}_{\text{TL}} & \Sigma/\mathcal{E}[e\overline{[x \mapsto v]}] \text{ where } (K\ \overline{x}) \Rightarrow e \in \overline{l} \text{ and is unique} \\
\Sigma/\mathcal{E}[(\textbf{letrec } (\overline{[\sigma\ v]})\ e)] & \overset{(3)}{\rightarrow}_{\text{TL}} & \Sigma\overline{[\sigma \mapsto v]}/\mathcal{E}[e] \\
\Sigma/\mathcal{E}[(\sigma\ \overline{v})] & \overset{(4)}{\rightarrow}_{\text{TL}} & \Sigma/\mathcal{E}[e\overline{[x \mapsto v]}] \text{ where } \Sigma(\sigma) = (\lambda(\overline{x})\ e) \\
\Sigma/\mathcal{E}[(\text{case } \sigma\ \overline{l})] & \overset{(5)}{\rightarrow}_{\text{TL}} & \Sigma/\mathcal{E}[(\text{case } \Sigma(\sigma)\ \overline{l})] \\
\Sigma/\mathcal{E}[(\text{abort } e)] & \overset{(6)}{\rightarrow}_{\text{TL}} & \Sigma/e \\
\Sigma/\mathcal{E}[(\text{w-c-m } v_1\ (\text{w-c-m } v_2\ e))] & \overset{(7)}{\rightarrow}_{\text{TL}} & \Sigma/\mathcal{E}[(\text{w-c-m } v_2\ e)] \text{ where } \mathcal{E} \neq \mathcal{E}'[(\text{w-c-m } v_3\ [\,])] \\
\Sigma/\mathcal{E}[(\text{w-c-m } v_1\ v_2)] & \overset{(8)}{\rightarrow}_{\text{TL}} & \Sigma/\mathcal{E}[v_2] \text{ where } \mathcal{E} \neq \mathcal{E}'[(\text{w-c-m } v_3\ [\,])] \\
\Sigma/\mathcal{E}[(\text{c-c-m})] & \overset{(9)}{\rightarrow}_{\text{TL}} & \Sigma/\mathcal{E}[\mathcal{X}(\mathcal{E})]
\end{array}$$

where: $\mathcal{X}([\,]) = (\text{nil})$
$\mathcal{X}((\overline{v}\ \mathcal{E})) = \mathcal{X}(\mathcal{E})$
$\mathcal{X}((\text{w-c-m } v\ \mathcal{F})) = (\text{cons } v\ \mathcal{X}(\mathcal{F}))$

Figure 6. TL Semantics

$$\begin{array}{rcl}
\Sigma/\mathcal{E}[((\lambda(\overline{x})\ e)\ \overline{v})] & \overset{(1)}{\rightarrow}_{\text{SL}} & \Sigma/\mathcal{E}[e\overline{[x \mapsto v]}] \\
\Sigma/\mathcal{E}[(\text{case } (K\ \overline{v})\ \overline{l})] & \overset{(2)}{\rightarrow}_{\text{SL}} & \Sigma/\mathcal{E}[e\overline{[x \mapsto v]}] \\
 & & \text{s.t. } (K\ \overline{x}) \Rightarrow e \in \overline{l} \text{ and is unique} \\
\Sigma/\mathcal{E}[(\textbf{letrec } (\overline{[\sigma\ v]})\ e)] & \overset{(3)}{\rightarrow}_{\text{SL}} & \Sigma\overline{[\sigma \mapsto v]}/\mathcal{E}[e] \\
\Sigma/\mathcal{E}[(\sigma\ \overline{v})] & \overset{(4)}{\rightarrow}_{\text{SL}} & \Sigma/\mathcal{E}[e\overline{[x \mapsto v]}] \\
 & & \text{where } \Sigma(\sigma) = (\lambda(\overline{x})\ e) \\
\Sigma/\mathcal{E}[(\text{case } \sigma\ \overline{l})] & \overset{(5)}{\rightarrow}_{\text{SL}} & \Sigma/\mathcal{E}[(\text{case } \Sigma(\sigma)\ \overline{l})] \\
\Sigma/\mathcal{E}[(\text{call/cc } v)] & \overset{(6)}{\rightarrow}_{\text{SL}} & \Sigma/\mathcal{E}[(v\ \kappa.\mathcal{E})] \\
\Sigma/\mathcal{E}[(\kappa.\mathcal{E}'\ v)] & \overset{(7)}{\rightarrow}_{\text{SL}} & \Sigma/\mathcal{E}'[v]
\end{array}$$

Figure 4. SL Semantics

Dereferencing store locations and beta substitution are combined in order to simplify the treatment of defunctionalization.

First-class continuations are captured using call/cc (rule 6). Capturing a continuation creates a continuation value, $\kappa.\mathcal{E}$ that records the context, $\mathcal{E}$, of the call/cc expression. The argument to call/cc is then applied to this value. When a continuation value is eventually applied to another value (rule 7), the evaluation context containing the application is discarded and replaced with the context contained in the continuation value. The hole in the new context is filled with the argument part of the application.

3.2 The Target Language

The target language in figure 5, dubbed TL for *t*arget *l*anguage, is also a modified A-Normal form much like the source language. Instead of call/cc, TL contains a simple abort construct and two new forms: w-c-m and c-c-m.

The operational semantics is specified via the rewriting system in figure 6. The first five rules are identical to the corresponding rules for the source language. In the source language, a continuation replaces the context when it is invoked. In order to simulate such continuations, we include the abort form in the target language, which handles the task of abandoning the context.

Continuation marks implement a mechanism for manipulating contexts, which we exploit for the elimination of call/cc. Intuitively, (w-c-m v e) installs the value v into the continuation of the expression e, while (c-c-m) recovers a list of all continuation marks embedded in the current continuation. To preserve proper tail-call semantics, if a rewriting step results in more than one w-c-m, surrounding the same expression, the outermost mark is replaced by the inner one. This requirement demands that an evaluation context interleaves w-c-m constructs with other kinds of expressions.

We translate this interleaving requirement into a syntactic constraint with a grammar for evaluation contexts that consists of two non-terminals. The start symbol, $\mathcal{E}$, enforces the interleaving requirement, while $\mathcal{F}$ defines the usual evaluation contexts. Thus, multiple adjacent w-c-m expressions must be treated as a redex. When such a redex is encountered, the redundant marks are removed starting with the outermost (rule 7). Marks surrounding a

$$\begin{array}{rcl}
e & ::= & a \\
 & \mid & (\overline{w}\ e) \\
 & \mid & (\textbf{letrec } (\overline{[\sigma\ v]})\ e) \\
 & \mid & (\text{w-c-m } a\ e) \\
 & \mid & (\text{c-c-m}) \\
 & \mid & (\text{abort } e) \\
 & \mid & (\text{case } w\ \overline{l}) \\
l & ::= & (K\ \overline{x}) \Rightarrow e \\
a & ::= & w \mid (K\ \overline{a}) \\
w & ::= & v \mid x \\
v & ::= & (\lambda(\overline{x})\ e) \mid (K\ \overline{v}) \mid \sigma
\end{array}$$

Figure 5. TL Syntax

Variables and Values:

$$\mathcal{CMT}[\![x]\!] = x \quad \text{(T1)}$$
$$\mathcal{CMT}[\![\sigma]\!] = \sigma \quad \text{(T2)}$$
$$\mathcal{CMT}[\![(\lambda(\overline{x})\ e)]\!] = (\lambda(\overline{x})\ \mathcal{CMT}[\![e]\!]) \quad \text{(T3)}$$
$$\mathcal{CMT}[\![\kappa.\mathcal{E}]\!] = (\lambda(x)\ (\mathsf{abort}\ (\textit{resume}\ \mathcal{X}(\mathcal{CMT}[\![\mathcal{E}]\!])\ x))) \quad \text{(T4)}$$
$$\mathcal{CMT}[\![(K\ \overline{a})]\!] = (K\ \overline{\mathcal{CMT}[\![a]\!]}) \quad \text{(T5)}$$

Redexes:

$$\mathcal{CMT}[\![(\overline{w})]\!] = (\overline{\mathcal{CMT}[\![w]\!]}) \quad \text{(T6)}$$
$$\mathcal{CMT}[\![(\textbf{letrec}\ (\overline{[\sigma\ w]})\ e)]\!] = (\textbf{letrec}\ (\overline{[\sigma\ \mathcal{CMT}[\![w]\!]]})\ \mathcal{CMT}[\![e]\!]) \quad \text{(T7)}$$
$$\mathcal{CMT}[\![(\mathsf{call/cc}\ w)]\!] = (\mathcal{CMT}[\![w]\!]\ ((\lambda(m)\ (\lambda(x)\ (\mathsf{abort}\ (\textit{resume}\ m\ x))))\ (\mathsf{c\text{-}c\text{-}m}))) \quad \text{(T8)}$$
$$\mathcal{CMT}[\![(\mathsf{case}\ w\ \overline{l})]\!] = (\mathsf{case}\ \mathcal{CMT}[\![w]\!]\ \overline{\mathcal{CMT}[\![l]\!]}) \quad \text{(T9)}$$
$$\mathcal{CMT}[\![(K\ \overline{x}) \Rightarrow e]\!] = (K\ \overline{x}) \Rightarrow \mathcal{CMT}[\![e]\!] \quad \text{(T10)}$$

Contexts:

$$\mathcal{CMT}[\![[\ ]]\!] = [\ ] \quad \text{(T11)}$$
$$\mathcal{CMT}[\![(\overline{w}\ \underline{\mathcal{E}})]\!] = ((\lambda(x)(\overline{\mathcal{CMT}[\![w]\!]}\ x))\ (\mathsf{w\text{-}c\text{-}m}\ (\lambda(x)\ (\overline{\mathcal{CMT}[\![w]\!]}\ x))\ \mathcal{CMT}[\![\underline{\mathcal{E}}]\!])) \quad \text{(T12)}$$

Compositions:

$$\mathcal{CMT}[\![\underline{\mathcal{E}}[r]]\!] = \mathcal{CMT}[\![\underline{\mathcal{E}}]\!][\mathcal{CMT}[\![r]\!]] \quad \text{(T13)}$$

Figure 7. Translation from SL to TL

```
(letrec ([resume
          (λ(l v)
            (case l
              (nil)          ⇒ v
              (cons v′ l′) ⇒ (v′ (w-c-m v′ (resume l′ v))))))])
  ...)
```

Figure 8. Definition of Resume

value are discarded after the evaluation of a sub-term completes, i.e., any immediate enclosing w-c-m is removed from the resulting value (rule 8). The side conditions for rules (7,8) guarantee the unique decomposition of a program into an evaluation context and a redex (or a stuck term), which implies that the rewriting relation defines an evaluation function. Finally, c-c-m employs the function $\mathcal{X}()$ to extract the marks from the evaluation context (rule 9). Marks are extracted in order, starting with the oldest.

3.3 Replacing Callcc

Figure 7 specifies a translation from SL to TL dubbed $\mathcal{CMT}$, for *c*ontinuation-*m*ark *t*ransform that eliminates call/cc and continuation invocations. The non-structural translation decomposes a term into a context and redex:

$$\begin{array}{rcll} r & ::= & (\overline{w}) & [redex] \\ & | & (\textbf{letrec}\ (\overline{[\sigma\ w]})\ e) & \\ & | & (\mathsf{call/cc}\ w) & \\ & | & (\mathsf{case}\ w\ \overline{l}) & \\ \underline{\mathcal{E}} & ::= & [\ ] \mid (\overline{v}\ \underline{\mathcal{E}}) & [context] \end{array}$$

The following lemma guarantees that the decomposition is unique and thus the translation is well-defined:

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

The translation rules for variables, values, and redexes are straightforward with the exception of call/cc applications and continuation values. Continuation values are transformed using rule T4. For call/cc, w-c-m is used to gather the requisite information from the surrounding context to construct a function that also relies on *resume* to reconstruct the evaluation context (rule T8). Since continuation values do not appear in the target language, a suitable function must be supplied. The function uses abort to oust the current context, and then calls a top-level function, *resume*, which builds a new evaluation context.

The treatment of both call/cc and continuation values relies on the systematic insertion of continuation marks. The strategy relies on the critical property of ANF terms that makes the continuation of every expression obvious. More precisely, evaluation contexts in the source language are built entirely from function applications; thus the λ-expression in the function position is always the continuation. Hence, the translation can mark each application with the function that is about to be applied. In turn, c-c-m collects all these functions in a list for storage and use at a later time.

The *resume* function (figure 8) faithfully reconstructs an evaluation context from such a list of functions. It traverses the list and recursively applies the functions from the list. An instance of w-c-m is wrapped around each function application so that the resulting evaluation context exactly matches and thus facilitates any future call/cc operations.

3.4 Example

Let us illustrate the translation with a simple example. We start with an expression that captures a continuation from within a non-trivial

evaluation context:

$$\mathcal{CMT}[\![(f\ y\ (\mathsf{call/cc}\ (\lambda\ (k)\ (k\ z))))]\!]$$

Note, we ignore the details of the bindings for *f*, *y*, and *z*.

Applying the translation yields:

```
((λ (x0) (f y x0))
 (w-c-m (λ (x0) (f y x0))
        ((λ (k) (k z))
         ((λ (m)
            (λ (x1) (abort (resume m x1))))
          (c-c-m)))))
```

The original expression can be decomposed into context and redex as:

$$(f\ y\ [])[(\mathsf{call/cc}\ (\lambda\ (k)\ (k\ z)))]$$

Observe that the evaluation context corresponding to the continuation of the call/cc is exactly (*f y* []), which can be rendered as the function (λ (x_0) (*f y* x_0)). This function represents the first part of the continuation and is used as the continuation mark surrounding the call/cc-expression. When the result is evaluated, (c-c-m) returns a list of the continuation marks, which in this case is (*list* (λ (x_0) (*f y* x_0))). This list becomes the first argument to *resume*, i.e., the actual continuation passed to (λ (*k*) (*k z*)) is

```
(λ (x1)
 (abort (resume (list (λ (x0) (f y x0))) x1)))
```

After evaluation of *resume*, we get:

```
((λ (x0) (f y x0))
 (w-c-m (λ (x0) (f y x0))
        z))
```

Thus the evaluation context captured by call/cc has been completely reconstructed. In the implementation, this corresponds to a faithful stack reconstitution.

3.5 Correctness

Let

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

THEOREM 1. $\mathcal{CMT}[\![\mathit{eval}_{SL}(p)]\!] = \mathit{eval}_{TL}(\mathcal{CMT}[\![p]\!])$

To prove Theorem 1, we show that if a source term admits a reduction sequence of length k, then the translation of the source term admits a sequence of length at least k, such that result of the target sequence is the translation of the result of the source sequence. This is proved by induction on the length (k) of the reduction sequence. The base case is trivial. To prove the induction step, we show that if a SL configuration Σ/e takes a single step of evaluation resulting in Σ'/e', then the translation $\mathcal{CMT}[\![\Sigma]\!]/\mathcal{CMT}[\![e]\!]$ admits only a finite number of evaluation steps before producing a term that is the translation of Σ'/e'. This simulation argument is summarized in Lemma 2.

LEMMA 2 (Simulation). *If* $\Sigma/\mathcal{E}[e] \rightarrow_{SL} \Sigma'/\mathcal{E}'[e']$ *then*

$$\mathcal{CMT}[\![\Sigma]\!]/\mathcal{CMT}[\![\mathcal{E}[e]]\!] \rightarrow_{TL}^{+} \mathcal{CMT}[\![\Sigma']\!]/\mathcal{CMT}[\![\mathcal{E}'[e']]\!]$$

The simulation lemma is proved by case analysis on the relation $\rightarrow_{\mathrm{SL}}$. Recall that $\mathcal{CMT}[\![\cdot]\!]$ is defined using unique decomposition. In proving this lemma, a pattern emerges. If the translation of a SL configuration takes a step of evaluation, then the resulting configuration may not be the image of any suitable SL configuration. Lemma 3 guarantees that the target configuration ultimately reaches a desirable state.

$$\begin{array}{rcl}
\mathcal{D}[\![x]\!]_{\mathcal{L}} & = & x \\
\mathcal{D}[\![\sigma]\!]_{\mathcal{L}} & = & \sigma \\
\mathcal{D}[\![(K\ \overline{a})]\!]_{\mathcal{L}} & = & (K\ \overline{\mathcal{D}[\![a]\!]_{\mathcal{L}}}) \\
\mathcal{D}[\![(\lambda^m(\overline{x})\ e)]\!]_{\mathcal{L}} & = & (K^m\ \overline{y}) \\
 & & \text{where } \overline{y} = \mathrm{FV}(\lambda^m(\overline{x})\ e) \\
\mathcal{D}[\![(w_0\ \overline{w})]\!]_{\mathcal{L}} & = & (\mathit{apply}\ \mathcal{D}[\![w_0]\!]_{\mathcal{L}}\ (\mathit{list}\ \overline{\mathcal{D}[\![w]\!]_{\mathcal{L}}})) \\
\mathcal{D}[\![(v\ e)]\!]_{\mathcal{L}} & = & (\mathit{app}\ \mathcal{D}[\![v]\!]_{\mathcal{L}}\ \mathcal{D}[\![e]\!]_{\mathcal{L}}) \\
\mathcal{D}[\![(\mathsf{abort}\ e)]\!]_{\mathcal{L}} & = & (\mathsf{abort}\ \mathcal{D}[\![e]\!]_{\mathcal{L}}) \\
\mathcal{D}[\![(\mathbf{letrec}\ (\overline{[\sigma w]})\ e)]\!]_{\mathcal{L}} & = & (\mathbf{letrec}\ (\overline{[\sigma\mathcal{D}[\![w]\!]_{\mathcal{L}}]})\ \mathcal{L}[\![e]\!]) \\
\mathcal{D}[\![(\mathsf{w\text{-}c\text{-}m}\ a\ e)]\!]_{\mathcal{L}} & = & (\mathsf{w\text{-}c\text{-}m}\ \mathcal{D}[\![a]\!]_{\mathcal{L}}\ \mathcal{D}[\![e]\!]_{\mathcal{L}}) \\
\mathcal{D}[\![(\mathsf{c\text{-}c\text{-}m})]\!]_{\mathcal{L}} & = & (\mathsf{c\text{-}c\text{-}m}) \\
\mathcal{D}[\![(\mathsf{case}\ w\ \overline{l})]\!]_{\mathcal{L}} & = & (\mathsf{case}\ \mathcal{D}[\![w]\!]_{\mathcal{L}}\ \overline{\mathcal{D}[\![l]\!]_{\mathcal{L}}}) \\
\mathcal{D}[\![(K\ \overline{x}) \Rightarrow e]\!]_{\mathcal{L}} & = & (K\ \overline{x}) \Rightarrow \mathcal{D}[\![e]\!]_{\mathcal{L}}
\end{array}$$

Figure 9. Defunctionalization

LEMMA 3 (Compositionality).

$$\mathcal{CMT}[\![\Sigma]\!]/\mathcal{CMT}[\![\mathcal{E}]\!][\mathcal{CMT}[\![e]\!]] \rightarrow_{TL}^{*} \mathcal{CMT}[\![\Sigma]\!]/\mathcal{CMT}[\![\mathcal{E}[e]]\!]$$

Two more technical results are needed. First, *resume* restores the evaluation context represented by its first argument.

LEMMA 4 (Reconstitution).

$$\begin{array}{rl} & \mathcal{CMT}[\![\Sigma]\!]/(\mathit{resume}\ \mathcal{X}(\mathcal{CMT}[\![\mathcal{E}']\!])\ \mathcal{CMT}[\![v]\!]) \\ \rightarrow_{TL}^{+} & \mathcal{CMT}[\![\Sigma]\!]/\mathcal{CMT}[\![\mathcal{E}']\!][\mathcal{CMT}[\![v]\!]] \end{array}$$

Second, substitution commutes with translation.

LEMMA 5 (Substitution).

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

3.6 Defunctionalization

Defunctionalization[1] (figure 9) replaces functions with records. Every such record belongs to exactly one variant of an algebraic datatype, which contains one variant for each λ expression in the program. Thus, we can view λ expressions as constructors for functions. We also need to know what to do when a function record appears in application position. For this, we define a global *apply* function that dispatches on the type of the function record and invokes the appropriate expression.

To defunctionalize a program, we first choose a labeling strategy that assigns a unique label to each λ expression in the program. This amounts to adding a superscript to each occurrence of λ. Based on the labeling, we can define the algebraic datatype. We create one variant for each λ, with fields corresponding to the free-variables of the corresponding λ expression.

The corresponding *apply* function consumes a record and dispatches on the record's constructor using case. The other argument to *apply* is a list of values representing the original function's arguments. There is a clause corresponding to each λ expression, and the right-hand-side is essentially the body of the original λ expression. Once the appropriate definitions have been made, we can replace every λ expression in the program with an application of the

[1] Our treatment of defunctionalization was inspired by the work of Pottier et al. [25], though we depart from their treatment due to additional constructs in our language, such as abort, continuation marks and multi-argument functions

(**letrec** ([*apply* (λ(*f vals*) (case *f* $\overline{c_m}$)]) . . .)
where for each m,

$c_m = (K^m\ \overline{y})$
$\Rightarrow (apply^*\ (\lambda\ (x_0) \ldots (\lambda\ (x_n)\ (\lambda\ ()\ \mathcal{D}[\![e_m]\!]_{\mathcal{L}})) \ldots)\ vals)$

(**letrec** ([$apply^*$ (λ(*f vals*)
(case *vals*
(nil) ⇒ (*f*)
(cons $val'\ vals'$) ⇒ ($apply^*$ (*f* val') $vals'$))])
. . .)

(**letrec** ([*app* (λ (*f v*) (*apply f* (*list v*)))])
. . .)

Figure 10. Definition of Apply, Apply*, and App

constructor of the corresponding variant. Finally, we rewrite each application with *apply*.

The definition of *apply* is complicated because we support multi-argument functions. If every function had only a single argument, substitution for that argument would happen automatically via the second argument to *apply*. For multi-argument functions, this breaks down, because we are confined to a single arity specification of *apply*. We therefore separate application into two stages. The first stage dispatches on the function record as described. The second stage handles substitution of values for variables in the body of the original function.

The multi-argument version of *apply* (figure 10) lumps all of the function arguments into a single list. In the clause corresponding to a particular function, we use a curried version of the original function. This curried function is passed as the first argument to the auxiliary, *apply**. The list of arguments is passed as the second argument to *apply**. If the list argument to *apply** is empty, the function argument should be a thunk, which is thus applied to no arguments. Otherwise, *apply** applies the function to the first value in the list, producing another function, and shorter argument list.

An application of a function to a single arbitrary expression is a special case. These are the applications that make up the evaluation context. For all other applications, the arguments are syntactic variables or values and can thus be collected in a list. For the special case, we define another apply-like function *app*.

4. Pragmatics

Translating the theory into practice poses different challenges in different contexts. Thus far, we have gathered experience in two different contexts: a language for servlets in the spirit of the PLT Web server [14] and an implementation of Scheme on Microsoft's .Net IL [22]. In this section, we briefly discuss how the translation works in these contexts and a few obstacles that we encountered.

4.1 Scheme and the Web Server

The PLT Web server [14] acts as an operating system for its servlets. Most importantly, the server implements I/O primitives and, when a servlet is loaded, links those primitives into the servlet. The primary I/O primitive is[2]

send/suspend ;; ((*URL* → *Response*) → *Request*)

The function consumes a function that maps a *URL* to a *Response*; its result is the next *Request* from the client (if any). When the servlet calls *send/suspend* with a function *f*, the PLT server grabs

[2] With *send/suspend*, one can implement a variety of multi-dispatch interaction functions [17].

```
(module add "persistent-web-interaction.ss"
  (require (lib "url.ss" "net"))

  ;; add2: → Number
  ;; obtain two numbers from client and compute the sum
  ;; (+ (get-number "first") (get-number "second"))
  (define (add2)
    (let ((one (get-number "first"))
          (two (get-number "second")))
      (+ one two)))

  ;; get-number: String → Number
  ;; ask the user for a number
  (define (get-number msg)
    (define (generate-html k-url)
      `(hmtl (head (title ,(format "Get ~a number" msg)))
             (body
               (form ([action ,(url->string k-url)]
                      [method "post"]
                      [enctype "application/x-www-form-urlencoded"])
                     ,(format "Enter the ~a number to add: " msg)
                     (input ([type "text"] [name "number"] [value ""]))
                     (input ([type "submit"])))))))
    (let ([req (send/suspend/url generate-html)])
      (string->number
        (extract-binding/single 'number (request-bindings req)))))

  ;; run, servlet, run
  (let ([initial-request (start-servlet)])
    `(html (head (title "Final Page"))
           (body
             (h1 "Final Page")
             (p ,(format "The answer is ~a" (add2)))))))
```

Figure 11. A PLT servlet for adding two numbers

the current continuation *k*, creates a unique *URL*, uses it as a key for indexing *k* in a hash-table, and then applies *f* to the *URL*. The resulting *Response* is shipped to the client and the servlet is suspended. If the client visits the generated *URL*, the server resumes the continuation from the hash-table and invokes it on the client's data.

The major advantage of this approach is that programmers do not have to understand the CGI protocol to get interactions with clients correct. Instead the programmer may act as if she were implementing an ordinary interactive program. For an example, consider the *add2* [26] function in figure 11. It reads two numbers from some input medium and adds them. The program is organized like a naive console-style program, and yet it works properly even in the face of Web interactions such as back buttons and clone functions.

An ordinary CGI script or Java servlet could not use this standard, 1960-ish organization. Because of the nested use of *get-number* (and its input action), the program would have to be contorted[3] to match the Web interaction protocol. More precisely, the script or servlet would consist of three distinct programs (roughly callbacks), the first two for the inputs and the last one for the output.

Naturally, the continuation-based approach comes at a price. Every interaction allocates space for a continuation on the server. Since the dynamic extent of this interaction is indefinite, it is impossible to garbage-collect this space in an ordinary manner. Keeping with the garbage collection analogy, some method must

[3] The "contorted" structure roughly corresponds to GUI callbacks. This analogy is misleading, however, because the control flow actions of ordinary GUI programs never include capabilities such as going back, cloning the GUI and exploring actions on the model in parallel, or bookmarking an interaction step. In contrast, the developer of a Web program must be aware of these kinds of actions, because they are an ordinary part of any Web browser. For details see our prior work [12].

be employed to determine the liveness of a URL. One possibility is to group interactions temporally into some kind of "session" construct and clean up at the close of the session. This approach precludes the bookmarking and emailing of URLs. Alternatively, continuations can be given a timeout so that after a period of time, the URL expires and can be cleaned up. Once again, bookmarking and emailing cause problems—in some cases, timeouts would have to be set to very large values. The timeout model degrades to the case where continuations are simply not cleaned up at all. In general, the approach doesn't scale well because it places client-oriented space on the server.

Based on our framework of call/cc elimination, we can overcome this obstacle. Specifically, we have prototyped a new implementation of our servlet interaction language, dubbed "persistent-web-interaction.ss". This prototype enables us to use a standard Web server such as Apache [32] to execute the servlets.[4]

The servlet language is provided as a PLT module language [10]. To create a program fragment in this language, a programmer creates a module, specifying the name of the module and the language: see the first line in figure 11. Our language comprises a subset of PLT Scheme, plus the *send/suspend* primitive. Furthermore, it is possible to import standard libraries and other modules written in plain PLT Scheme into a servlet. In figure 11, (**require** (*lib* "url.ss" "net")) imports the standard PLT libraries for manipulating *URL*s.

A module language such as "persistent-web-interaction.ss" consists of a set of macros and library functions, also known as a "run-time library." The macros translate the code in the subject module. The generated code typically refers to the library functions from the language module.

PLT Scheme's macro system suffices to implement the translation from section 3 in an almost literal manner because PLT Scheme provides continuation marks. The run-time environment for "persistent-web-interaction.ss" consists of two functions, one used by the server and one used when writing servlets:

1. *send/suspend*, which builds on call/cc. This function collects the continuation marks and serializes them. Of course, it must also terminate the servlet so that the server can send the response to the client. In short, this new implementation of *send/suspend* is completely consistent with the standard CGI protocol, to ensure compliance with traditional servers.

   Since it is not obvious where an interactive Web program should store the serialized continuation, our prototype actually provides two *send/suspend* primitives: *send/suspend/url*, which creates a URL from the serialized continuation, and *send/suspend/hidden*, which stores it in a hidden field on the generated page. The URL-based version accommodates the bookmarking facility of browsers; the Web-based version overcomes systems limitations, which sometimes restrict the amount of information that can be stored in URLs. We intend to experiment with both primitives until we have definitive experimental data.
2. *dispatch*, used by the server, which uses a URL to reconstitute the embedded continuation, thus re-launching a servlet's computation from the last interruption point. The function is like *resume* from section 3, but also deals with the decoding of the URL and some other book-keeping details.

The rest of the section presents three specific problems and obstacles, how we have solved them for now, and what a general solution may look like.

[4] To simplify the prototyping effort, we actually implemented our own standard (continuation-free) Web server; in principle, however, our servlets could run on any standard server.

1. The first problem is due to the defunctionalization phase of the translation. Recall that the defunctionalization phase attaches a label to each λ expression in the program. These labels become the structure tags for the function values that are published as part of the continuation. The theoretical treatment of defunctionalization ignores the details of how the labels are generated. Furthermore, it assumes that the labeling is fixed throughout the program's execution.

   A Web-server may suspend and resume the same servlet several times during a single interaction with a particular user. Thus if a continuation is published during the execution of a particular translated version of a servlet then the same translated version must be used at the time when the continuation is invoked. Otherwise, the labeling chosen during the latter's defunctionalization would likely be inconsistent with the labels that were generated during the earlier defunctionalization and that are now embedded in the URL.

   An obvious but naive solution is to simply compile the servlet, i.e., translate the servlet once, storing the result in a file, and then arrange for the server to load the compiled version for all requests to the servlet's URL. Consider the case, however, where the servlet is modified and then recompiled. It is critical that the set of labels chosen during defunctionalization of the new version be disjoint from the previous set of labels. Otherwise, outstanding continuations could be misinterpreted under the new version. The opposite problem arises when the program's compiled code is kept in memory, rather than the file system. If an identical version of a servlet is compiled then the same labeling should be used.

   Notice that none of these problems arise if when given identical input, elaboration yields identical output, i.e., if elaboration is a function. This requires generating identical labels each time the servlet is defunctionalized. To achieve a consistent labeling, the translator computes a message digest based on the syntactic structure of the program. The translator stores the digests in a database and associates a unique small key with each digest. The key is then used as the prefix for each label in the program. If the servlet is changed in some non-trivial way (comments and whitespace are trivial), the computed message-digest changes and a new key is generated. Values published with the old key become obsolete and the server can fail gracefully. Note that the digests are associated with *program source versions*, not with servlet invocations, so the space costs are negligible.

One of the goals of this research was to allow interoperability between translated code and existing untranslated libraries. Unlike CPS, our translation does not change the calling signature of translated functions, so in theory, it should be possible for translated and untranslated modules to call each other's functions. The remaining two problems illustrate the problems encountered when a servlet interoperates with untranslated code. Most of these problems arise when trying to use higher-order functions, so we will use the example in figure 12 for illustrative explanations.

The program in figure 12 consists of two modules: one in our new Web programming language and one in ordinary PLT Scheme. The purpose of the program is to ask a series of multiple choice questions. When the servlet is loaded, it uses *map* to pose each question and to accumulate the answers in a list. Afterwards, it tallies the results and presents them to the student. The questions are represented as instances of the *mc-question* structure; the quiz itself is a list of instances of this structure. The function *get-answer* retrieves the answer for a single question from the client.

2. The second problem concerns the interoperability between ordinary Scheme functions and function application in our new Web programming language. The most intricate example is the

```
(module quiz "persistent-web-interaction.ss"
  (require "quiz-lib.ss"
           (lib "url.ss" "net")
           (lib "servlet-helpers.ss" "web-server"))

  ;; get-answer: MC-Question → Number
  ;; get an answer for a multiple choice question
  (define (get-answer mc-q)
    (let* ([req (send/suspend/hidden (make-cue-page mc-q))]
           [bdgs (request-bindings req)])
      (if (exists-binding? 'answs bdgs)
          (string->number (extract-binding 'answs bdgs))
          -1)))

  ;; tally: (Listof MC-Question) (Listof Number) → Number
  ;; count the number of correct answers
  (define (tally mc-qs answs) ...)

  ;; run, servlet, run:
  (let ([initial-request (start-servlet)])
    `(html (head (title "Final Page"))
           (body
            (h1 "Quiz Results")
            (p ,(format
                 "You got ~a correct out of ~a questions."
                 (tally quiz (map get-answer quiz))
                 (length quiz)))
            (p "Thank you for taking the quiz")))))
```

```
(module quiz-lib mzscheme
  (require (lib "serialize.ss")
           (lib "url.ss" "net"))

  (provide ;; type: MC-Question
           ;; = (make-mc-question String (Listof String) Number)
           (struct mc-question (cue answers correct-answer))
           make-cue-page
           quiz)

  (define-struct mc-question (cue answers correct-answer))

  ;; make-cue-page: MC-Question → URL HiddenField → HtmlPage
  ;; generate the page for the question
  (define (make-cue-page mc-q)
    (λ (ses-url k-hidden)
      `(hmtl (head (title "Question"))
             (body
              ... (form ([action ,ses-url]) ... ,k-hidden ...)))))

  ;; the quiz: (Listof MC-Question)
  (define quiz
    (list
     (make-mc-question "Where do babies come from?"
                       (list "The cabbage patch"
                             "The stork"
                             "A watermelon seed"
                             "Wal-Mart") 1)
     ...)))
```

Figure 12. A multi-module servlet

underlined use of *map* on *get-answer*. Since *get-answer* is defined in the servlet module, it is subject to elaboration. In the resulting code, *get-answer* is a structure that *send/suspend* can serialize into a URL if it is found on the stack. In contrast, *map* is a standard library function and is therefore not translated.

Fortunately, PLT Scheme provides structs that act as functions. If a structure definition specifies its instances as procedural, it must provide an additional slot in which it stores the function to be used in function applications. Using such structures, our translation can actually represent the defunctionalized functions in a way that represent continuations as functions and serializable structs simultaneously.

The call to *map* poses another problem due to calling its arguments in a higher-order context. Recall *map*'s conventional definition:

```
;; map: (α → β) (Listof α) → (Listof β)
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l)) (map f (rest l)))]))
```

The definition reminds us that the "callback" to *f* takes place in a non-trivial evaluation context. Since native *map* is not subject to translation, a call to *send/suspend* during the dynamic extent of the callback would miss the context fragment (cons [] (*map f* (*rest l*))). The result would be a continuation with parts missing, resulting in undefined behavior.

Fortunately, we can employ stack-inspection to detect the special circumstances that would otherwise lead to undefined behavior and instead signal a runtime error with an informative error message. To detect the special case, we must use a property of PLT Scheme's continuation marks that is not a part of the theoretical model from section 3. In particular, PLT Scheme supports the definition of multiple disjoint sets of continuation marks by allowing programs to associate marks with a key that uniquely identifies the set to which the marks belong. Using this mechanism, we create a set of marks for the sole purpose of annotating possibly unsafe calls to higher-order functions; when a continuation is to be captured and serialized, *send/suspend* inspects this set for unsafe marks. If any such marks are encountered, the function signals an error.

For clarification, we illustrate the details of using such "safety" marks via our running example. For use as a key, associated with boolean values, we create a unique value and bind it to the identifier, *safe?*. The translator marks the body of every translated function using true and when it encounters an application of a possibly untranslated function it uses false. After safety annotations are added, the underlined call to *map* in figure 12 becomes:

```
(w-c-m safe? false (map get-answer quiz))
```

The translated version of *get-answer* is

```
(define (get-answer mc-q)
  (w-c-m safe? true
    (let* ...)))
```

And finally, the following fragment of code results from reducing the now annotated call to *map*:

```
(w-c-m safe? false
 (cons (w-c-m safe? true
         (let* ([req (send/suspend/hidden ...)])
           ...))
       (map get-answer (rest quiz))))
```

The inner w-c-m is not in tail position with respect to the outer w-c-m, so both marks appear in the list associated with the *safe?* key. The presence of the false value in this list results in an error when *send/suspend/hidden* is invoked.

Now our servlet always signals an error at the first interaction with the user. To overcome this error, the servlet writer is forced to move the definition of *map* into the servlet module so that it becomes subject to translation. In the translated version, the outer mark is canceled by the mark around the body of the

```
int fact (int x) {
  if (x < 2)
     return 1;
  else
     return
      x * fact (x - 1);
}
```

```
int fact (int x) {
  if (x < 2)
     return 1;
  else {
     int temp0
       = fact (x - 1);
     return x * temp0;
  }
}
```

```
int fact (int x) {
  if (x < 2)
     return 1;
  else {
     int temp0;
     try {
         temp0 = fact (x - 1);
         }
     catch (SaveContinuation sce) {
         sce.Extend (new fact_frame0 (x));
         throw;
         }
     return x * temp0;
  }
}
```

Figure 13. Continuations and MSIL

programmer-defined *map* due to the tail-call optimization for continuation marks.

Despite these complications, our translation is superior to CPS. In particular, our translated code can always interoperate with untranslated code whereas CPS breaks down in the presence of higher-order functions. Furthermore, we have a general technique that discovers the mismatch and signals an error. Ideally, there would be no such error cases, so in this regard we claim only a partial solution.

Pragmatically, in the context of Web interactions, the cases involving unsafe continuation capture are precisely those cases that require special treatment with regard to managing program state across interactions. Obtaining finer control over servlet state requires moving higher order code into the domain of the translation. This can be problematic, as the next example illustrates.

3. The third problem becomes visible when we eliminate the use of *map* by supplying our own definition (here called *get-answers*) and subjecting it to translation:

```
;; get-answers:
;;       (Listof MC-Question) → (Listof Number)
;; get the answers to all the questions in mc-qs
(define (get-answers mc-qs)
   (cond
      [(empty? mc-qs) empty]
      [else (cons (get-answer (first mc-qs))
                  (get-answers (rest mc-qs)))]))
```

Thus, in place of (*map get-answer mc-qs*) we can write (*get-answers quiz*).

Since *get-answer* uses *send/suspend*, it captures the continuation of (*get-answer* (*first mc-qs*)), which is closed over *mc-qs*, the list of questions. This information is serialized into the URL transmitted to the user, which can lead to a significant growth in the size of this URL. Furthermore, an implementation may even leak sensitive information in this URL.[5]

Our partial solution would be complete but for the question of what to do when a Web-interaction is encountered from within the a call to a higher-order function. In a system that provides native continuations, we can extend the partial solution to a full solution by falling back to native continuations when "unsafe" marks are discovered on the stack. This proposal automatically places program state on the server in the case when it is not explicitly handled by the Web programmer. The use of native continuations is still subject to the limitations discussed previously and thus the Web programmer must still be aware of the implications of using higher order libraries.

To overcome the limitations of native continuations, the Web programmer will have to explicitly code certain higher order functions so that they will be subject to translation. In this case, continuations are closed over program data which can either be stored on the server or encoded in the outgoing response. We are currently considering a memoization mechanism that keeps immutable data, such as the list of questions in our example, on the server and then encodes an opaque reference in the URL. When the servlet is re-launched, the mechanism (re)computes the necessary value based on the reference.[6]

4.2 Scheme and MSIL

The use of virtual machine languages as intermediate representations has become the norm in recent compiler developments. Microsoft's Common Language Runtime [22] and Sun's JVM [34] are prominent examples. Compiling a language to either of these VMs almost immediately equips the language with rich run-time libraries and with access to programming environment tools, including debuggers and profilers. Due to various reasons, however, these machines do not grant programs full access to their stacks.

Compiling Scheme, Smalltalk, Ruby or any other language that uses first-class continuations to such an architecture thus poses a dilemma. At least at first glance, the compiler writer must either forego the implementation of continuations or manage a stack-away-from-the-VM stack. The first choice limits the programmers of these languages, and the second gives up on many of the advantages that these machines supposedly offer.

Our discovery that continuation marks can implement first-class continuations resolves this dilemma. Even though the widely used VMs don't implement continuation marking mechanisms in the spirit of PLT Scheme, they do implement means for installing exception handlers, and those are suitable for mimicking continuation marks. It is in this regard that we consider continuation marks as a generalization of other stack inspection mechanisms.

In this section, we illustrate how generalized stack inspection is adapted to the restricted virtual machine environment of the Common Language Runtime [22]. Roughly speaking, marking a continuation (w-c-m) corresponds to the installation of an exception handler; the inspection of the continuation marks (c-c-m) can be accomplished by raising an exception, thus transferring control to code that writes a representation of the mark to the heap as the

[5] The continuation also includes the user's answers to the questions that have already been completed; in this instance this is less problematic because the information is being transmitted to its very author, but in general this requires a cryptographic solution.

[6] The idea of recomputing such values is also present in the WASH/CGI framework [33].

stack unwinds. To avoid the complexities of the machine language, we present our discussion in terms of C#. The use of C# makes it easier to convey the ideas behind the implementation without hiding any of the engineering problems that we encounter.

The chosen example is the `fact` program, shown on the left of figure 13. As described, the first step is to convert the program to ANF. This requires the introduction of local variables to hold the intermediate results of compound expressions. The normalized program has the property that all compound expressions are composed only of primitive subexpressions and appear either on the right hand side of a new variable binding or as the expression in a return statement. For `fact`, this conversion is near-trivial and its result is the function in the center of figure 13.

Once the program is in ANF, we wrap each function call with an exception handler: see the right-most code fragment in figure 13. This use of the `try-catch` construction is equivalent to w-c-m in the theoretical model. In the model, the continuation marks introduced by w-c-m are closures and are eagerly created when the function is entered. In the C#-implementation, the analogous closures are only created as the stack is unwound by the special exception. In effect, the representation is created lazily.

Next consider two scenarios. First, if the program must capture a continuation during the dynamic extent of the `try` block, it throws an exception. Specifically, it throws an exception—`SaveContinuation`—that only the newly inserted handlers know about and catch. Second, the `try` block returns normally. In this case, the continuation mark isn't needed and no extra work is performed. That is, a program that does not use first-class continuations pays only for the establishment of exception handlers around (non-tail) function calls. Fortunately, the implementors of the virtual machine assume that exception handlers are established with some frequency and have therefore made this a reasonably inexpensive operation.

Following our model, a C#-implementation of call/cc must implement a search of all continuation marks. A targeted `throw` of an exception takes control to the handlers around function calls. The handler then constructs the closure—represented as objects—that represents the respective mark in the continuation:

```
...
try {
  temp0 = fact (x - 1);
}
catch (SaveContinuation sce) {
  sce.Extend (new fact_frame0 (x));
  throw;
}
...
```

Once the current frame has been added to the list of continuation marks, the program re-raises the exception so that it eventually stops the program.

Naturally, we can't let the throw of a `SaveContinuation` exception stop the program. Instead, our implementation surrounds the entire program with a handler that, according to the semantics of call/cc immediately *resume*s the program with the current continuation and hands the continuation to the argument of call/cc.

The implementation of *resume* poses an additional complication. The raising of a `SaveContinuation` exception (and its re-raising) collects the continuation marks in most-recent to the least-recent order. Because we reconstruct the context from the least-recent first, the natural ordering is conveniently correct. If a second continuation is captured within this restored context, however, we must ensure that the marks common to both continuations are not duplicated. Duplicating these marks would duplicate closures, both causing problems with synchronization and changing the order of space usage. Obviously the former might affect correctness, and the latter would significantly hurt the performance of our strategy.

```
void Resume (Context frame,
             ContextList moreRecentFrames) {
   object returnValue;

   if (moreRecentFrames == null)
     returnValue = null;
   else
     returnValue =
      Resume (moreRecentFrames.first,
               moreRecentFrames.rest);

   try {
     return frame.Invoke (returnValue);
   }
    catch (SaveContinuation sce) {
      sce.AppendSharedContext (
           frame.OlderContext);
      throw;
   }
}
```

Figure 14. The implementation of *resume* for MS IL

For these reasons, we implement *resume* such that it avoids duplicating the shared parts of the continuation marks that represent the evaluation context: see figure 14. More precisely, each frame in the context recursively reloads the more recent frames. Note that the recursive call returns to a `try-catch` block like the one in `fact`. The recursion terminates when the most recent frame is reloaded. Pending chains of calls to *resume* are not protected by an exception handler so that they aren't duplicated by the capture. The exception handler is only established when a frame is about to continue execution (the remainder of `Resume`). The exception handler is different from the one we wrap around the original code. It uses the already computed representation of the evaluation context. When the `SaveContinuation` exception is thrown again, each newly created evaluation context saves its state, but the topmost restored frame arranges for these to be linked to the previously saved context.

5. Related Work

Three pieces of past research bear a strong resemblance to ours: Cartwright and Felleisen's work on extensible denotational semantics [3], Sekiguchi, Sakamoto and Yonezawa's work on checkpointing and transparent migration [29], and Tao's work on migrating Java threads [31].

Cartwright and Felleisen adapt Felleisen's work [5, 6] on an imperative extension of the lambda calculus to a denotational setting. In the traditional Scott-Strachey framework for denotational semantics [30], a language extension deeply affects the structure of the mapping from syntax to semantics. For example, if a language designer wishes to add continuations to a functional language, a revision of the semantic mapping must introduce a parameter that abstracts over the continuation of an expression (statement, definition). Worse, the change to the denotational mapping radically changes the denotation of an expression in the original language.

In the Cartwright-Felleisen framework, the denotation of an expression may return either a value or an effect. Returning an effect is analogous to throwing an exception. Each denotation is equipped with a handler-like function that augments the effect in an appropriate manner and passes it to the context. Ultimately, an effect reaches a handler function, dubbed `admin`, at the root of the denotational tree. Given this arrangement, a language designer can easily extend a language by injecting new effects and adding

corresponding clauses in the `admin` function. A universal theorem governs such language extensions. Specifically, for each language extension, there is a projection that can eliminate the relevant parts of a denotation for any expression in the core language and re-create its original denotation.

The call/cc-elimination transformation of this paper is to the Cartwright-Felleisen framework what the CPS transformation is to the Scott-Strachey framework of denotational semantics. In other words, it is a syntactic analog of the semantic mapping in Cartwright-Felleisen. From this perspective, the soundness theorem in this paper finally confirms that the call/cc expressed in this framework is equivalent to the original call/cc, something that Cartwright and Felleisen failed to confirm.

Sekiguchi et al. describe a method for implementing partial continuations in Java and C++, with check-pointing and thread migration as possible applications. The method uses exception handling to construct continuation values as the stack unwinds. In lieu of call/cc, Sekiguchi et al. use Gunter et al.'s partial and delimited continuations [15]:

`cupto` p `as` x `in` e	capturing the functional continuation
`set` p `in` e	delimiting the effect of `cupto`

with the following evaluation rule for `cupto`:

$$\texttt{set}\ p\ \texttt{in}\ E[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ e] \rightarrow (\lambda x.e)(\lambda y.E[y])$$

Notice that evaluation of `cupto` abandons the surrounding context, which more closely models the semantics of stack unwinding than other control-flow operations such as call/cc.

Our translation is defined on program source code, while Sekiguchi et al. define their translation on byte-codes. Their translation also performs a fragmentation step analogous to A-Normalization. During resumption an extra parameter carries the control state and extra logic is added to each function to distinguish between normal function calls and resumption. This amounts to an in-lining of *resume*.

Sekiguchi et al., like us, are concerned with faithfully reconstructing the stack. More critical to their choice of languages, Sekiguchi et al. are concerned about "preserving the call-graph" of the original program. The authors dismiss a potential solution based on CPS transformation, because CPS transformed code creates an unbounded sequence of tail calls.

Sekiguchi et al. do not go into much depth with regards to how to combine translated and untranslated code. For example, they do not discuss the problems associated with changing the calling signature of each method. They do recognize the problem where the call stack contains stack frames of non-transformed methods. We introduce safety marks as a partial solution to this case, while they have no solution and identify it as a limitation of their scheme. Finally, Sekiguchi et al. offer no theoretical treatment of their technique.

Like Sekiguchi et al., Tao describes a method for thread persistence and migration, leveraging Java's exception mechanism. Tao defines the translation for source code and byte-codes.

Tao gives a minimal theoretical treatment of the method based on a denotational semantics for a small while-loop language. There is no theorem stating the equivalence of the original and translated forms of the program, i.e., there is no correctness result. Tao does attempt to prove that evaluating a program in the presence of any number of shutdown operations is equivalent to evaluating the program without interruptions. The proof is skeletal and relies on an unspecified extension to the denotational semantics.

Tao does not employ any kind of fragmentation before annotating the source code. For example, statement sequences are not broken. Instead, her work maintains an index and adds extra logic to the code so that the resumption point can be relocated when a continuation is reinstated. A further complication is that Tao essentially makes two versions of the program: a "shutdown" version and a "restart" version. Extra logic needs to be sprinkled throughout the code to distinguish between normal function calls and resumption. As with Sekiguchi et al., this is analogous to in-lining of *resume*.

For the source code version of the transformation, Tao makes the restriction that continuations cannot be captured during the evaluation of an expression. This restriction is needed because it is supposedly difficult to "save temporary variables at the source file level" [31, section 3.2.7]. This can be seen as a consequence of not fragmenting the original program. Because there is no explicit closure identified with the continuation of each expression, it is difficult to reason about the free variables of that closure.

Finally, Schinz and Odersky [28] use a small portion of our transformation to implement tail-call optimizations for the JVM. Specifically, they install a handler (a.k.a., trampoline) at each transition from a properly nested call to a chain of tail-calls; if the depth of tail-calls gets too deep, they erase them with a jump to the closest handler, which then reconstitutes the *one* stack frame for the properly nested call. Schinz and Odersky did not recognize that their transformation could be generalized to deal with first-class continuations.

6. Perspective and Conclusion

We have presented a novel formal transformation for explicating the continuation of a computation. Our transformation is defined using continuation marks, which we regard as a generalized form of stack inspection. We have also demonstrated that the transformation can be implemented using exceptions—a stack inspection mechanism—which can be regarded as standard on modern virtual machines (and in a growing set of languages). By exploiting stack inspection, we show how to capture and reconstruct the stack without the need for CPS, which depends on tail calls (that many virtual machines don't provide) and "hides" the stack (which is necessary not only for many debugging and program understanding tools, but also for contemporary security mechanisms).

Our transformation has two immediate applications. First, it is valuable for continuation-based Web applications that need to run in a stateless manner for greater scalability. Second, it helps implement languages with continuations, such as Scheme, on traditional virtual machines without emasculating the language or making its code incompatible with the VM's assumptions (since virtual machines are not designed to assume the code they run will be in CPS).

We conjecture that there are deeper applications, which we have not yet explored in detail:

- The transformation could improve the usability of continuation-based Web applications. Currently, the PLT Web server generates a nonce for each captured continuation and embeds this in the URL. These nonces become invalid when the server reboots, making it useless to bookmark or distribute these URLs. In contrast, like the CPS solution, our transformation can generate more durable URLs by referring to (named) code fragments and placing the free variable values in an argument list, creating a form of "Web command line".
- The transformation makes it possible to employ a variety of techniques for implementing first-class continuations. For instance, the scheme by Dybvig, et al. [16] enables constant-time continuation capture and application, by making stack copy and restoration lazy. Our representation of the stack makes it easy to similarly reconstitute only a constant number of stack segments, leaving a pointer to a function closed over the rest of the stack that reconstitutes more of it on demand. Implementing a similar scheme atop CPS would require customization of the

function charged with marshalling and unmarshalling closures to recognize the special case of the continuation argument.

- The transformation has implications for virtual machine designers also. The Parrot virtual machine, which is intended to provide a common platform for scripting languages such as Perl, Python and Ruby, has considered using CPS just to support continuations; its developers have been evaluating the trade-off between the benefits of continuations and the (user) cost of programming the entire system in CPS.[7] Our transformation offers them a potential way out of this quandary.
- Current implementations of Scheme targeted at virtual machines do not interact with or exploit the underlying security mechanisms. Indeed, it would be difficult for those that use CPS to do so, due to the (unfortunate) explicit reliance of these mechanisms on the stack. Our technique restores the primacy of the stack, making it possible to provide these security features as language extensions. Furthermore, even Web applications running on these virtual machines may be able to use these security extensions, because our transformation reconstitutes the stack.

Our immediate goal is to extend this transformation to handle the full Scheme language, and to apply it to both our compiler and Web server. In particular, we hope to use this to improve the performance of our conference management application.

References

[1] Anderson, K., T. Hickey and P. Norvig. JScheme. `http://www.norvig.com/jscheme.html`.

[2] Bres, Y., B. Serpette and M. Serrano. Bigloo .NET: compiling Scheme to .NET CLR. *Journal of Object Technology*, 3, October 2004.

[3] Cartwright, R. and M. Felleisen. Extensible denotational language specifi cations. In *Theoretical Aspects of Computer Software*, pages 244–272, 1994.

[4] Clements, J., M. Flatt and M. Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, pages 22–37, 2001.

[5] Felleisen, M. and D. Friedman. A syntactic theory of sequential state. In *Theoretical Computer Science*, pages 243–287, 1989.

[6] Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. In *Theoretical Computer Science*, pages 235–271, 1992.

[7] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[8] Flanagan, C., A. Sabry, B. F. Duba and M. Felleisen. The essence of compiling with continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–247. 1993.

[9] Flatt, M. PLT MzScheme: Language manual. `http://www.plt-scheme.org/software/`, 1996–2005.

[10] Flatt, M. Composable and compilable macros: You want it *when?* In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[11] Graham, P. Beating the averages. `http://www.paulgraham.com/avg.html`, April 2001.

[12] Graunke, P., R. Findler, S. Krishnamurthi and M. Felleisen. Modeling web interactions. In *European Symposium on Programming*, pages 238–252, April 2003.

[13] Graunke, P. T., R. B. Findler, S. Krishnamurthi and M. Felleisen. Automatically restructuring programs for the Web. In *IEEE International Symposium on Automated Software Engineering*, pages 211–222, November 2001.

[14] Graunke, P. T., S. Krishnamurthi, S. van der Hoeven and M. Felleisen. Programming the Web with high-level programming languages. In *European Symposium on Programming*, pages 122–136, April 2001.

[15] Gunter, C. A., D. Rémy and J. G. Riecke. A generalization of exceptions and control in ML. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, June 1995.

[16] Hieb, R., R. K. Dybvig and C. Bruggeman. Representing control in the presence of fi rst-class continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990.

[17] Hopkins, P. W. Enabling complex UI in Web applications with send/suspend/dispatch. In *Scheme Workshop*, 2003.

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

[19] Kelsey, R., W. Clinger and J. Rees (Eds.). Revised^5 report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[20] Krishnamurthi, S. The CONTINUE server. In *Symposium on the Practical Aspects of Declarative Languages*, pages 2–16, January 2003.

[21] Matthews, J., R. B. Findler, P. Graunke, S. Krishnamurthi and M. Felleisen. Automatically restructuring programs for the Web. *Automated Software Engineering*, 11(4):337–364, 2004.

[22] Microsoft Corporation. The .NET common language runtime. `http://msdn.microsoft.com/net/`.

[23] Miller, S. SISC. `http://sisc.sourceforge.net`.

[24] Plotkin, G. D. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[25] Pottier, F. and N. Gauthier. Polymorphic typed defunctionalization and concretization. To appear in *Higher-Order and Symbolic Computation*, May 2005.

[26] Queinnec, C. The influence of browsers on evaluators or, continuations to program web servers. In *ACM SIGPLAN International Conference on Functional Programming*, pages 23–33, 2000.

[27] Reynolds, J. C. Defi nitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740. ACM Press, 1972.

[28] Schinz, M. and M. Odersky. Tail call elimination on the java virtual machine. In *Proc. ACM SIGPLAN BABEL'01 Workshop on Multi-Language Infrastructure and Interoperability.*, volume 59 of *Electronic Notes in Theoretical Computer Science*, pages 155–168. Elsevier, 2001. `http://www.elsevier.nl/locate/entcs/volume59.html`.

[29] Sekiguchi, T., T. Sakamoto and A. Yonezawa. *Portable implementation of continuation operators in imperative languages by exception handling*, volume Advances in Exception Handling Techniques, pages 217–233. Springer-Verlag, 2001.

[30] Stoy, J. E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[31] Tao, W. *A portable mechanism for thread persistence and migration*. PhD thesis, University of Utah, 2001.

[32] The Apache Software Foundation. Apache HTTP Server Project. `http://httpd.apache.org`.

[33] Thiemann, P. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *Practical Applications of Declarative Languages*, 2002.

[34] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specifi cation (Second Edition)*. Sun Microsystems, 1999.

[35] Wallach, D., E. Felten and A. Appel. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4), October 2000.

[7] Dan Sugalski, message to the Lightweight Languages mailing list, May 7, 2003.