Implementing Continuation Marks in JavaScript

John Clements
California Polytechnic State University
clements@brinckerhoff.org

Ayswarya Sundaram
California Polytechnic State University
asundara@calpoly.edu

David Herman
Northeastern University
dherman@ccs.neu.edu

Abstract

MzScheme's continuation marks provide a flexible mechanism for implementing a number of useful language features and tools. We demonstrate the simplicity and utility of continuation marks by adapting them for JavaScript as frame-based stack marks using the Rhino implementation, showing a simple model of their behavior, and using them to build a toy debugger.

Along the way, we discover a few interesting things. First, it requires some thinking—but not much code—to add continuation marks to JavaScript. Second, coupling tail-calling with the "return" of statement-based languages leads to some interesting problems in formulating a semantics. Third, building a debugger based on continuation marks highlights (by its absence) the elegance of Scheme's simple syntax and hygienic macro system.

1. Introduction

Many languages and language tools require information about the shape and content of a program's dynamic context. Scheme contains several of these: `dynamic-wind`, `with-output-to-file`, and exceptions are part of the standard (Sperber et al. 2007), and many implementations extend this set with derived features such as `fluid-let` and `parameterize`. Language tools such as debuggers and profilers likewise depend on information about the dynamic context.

Earlier work (Flatt et al. 2007; Clements 2005) suggests that continuation marks provide a portable and high-level basis for the storage and retrieval of such dynamic information in a way that respects tail-calling.

Continuation marks allow programmers to create dynamic bindings whose behavior exposes the tail-calling nature of the underlying language. This enables the development of debugging and profiling tools that can observe tail-calling and also the implementation of existing features (e.g., `dynamic-wind`) in a way that has asymptotically better memory behavior in some cases.

No other languages currently implement continuation marks. In order to determine whether continuation marks are easy to implement and applicable to a broad range of languages, we chose to adapt them to the Rhino implementation of JavaScript, a language that supports closures and continuations and is designed to be properly tail-calling, but that is otherwise fairly different; it is statement-based, it makes frequent idiomatic use of return, and its basic unit of dynamic organization is the call frame. These complicate the addition of continuation marks to such languages.

This paper introduces *stack marks*, a variation of MzScheme's continuation marks designed for languages like JavaScript, to solve these problems. Like MzScheme's continuation marks, stack marks allow programmers, language implementers, and IDE designers alike to build tools that observe the computational behavior of programs without interfering with proper tail calling.

Stack marks enable a large range of language features that could otherwise be difficult to implement without interfering with tail calls. Their addition to languages like JavaScript thus paves the way for the inclusion of proper tail calls to the language specification.

We present a more detailed intuition for stack marks in section 2. We build a small model for the computational core of JavaScript in section 3. We describe the addition of stack marks to Mozilla's Rhino implementation of JavaScript in section 4, and describe a debugger built on stack marks in section 5.

Following this, we explore additional related features and tools, including stack mark combination, which turns out not to change the expressiveness of the language (section 6), dynamic binding (section 7), a slightly unusual model for continuations (section 8), a way to implement exceptions (section 9), and stack inspection (section 10).

2. Stack Marks

Stack marks are designed to allow programs to observe their own contexts.

One way to achieve this is to simply add a primitive function that returns a value that represents information about the dynamic context. This is a problematic approach, for two reasons. Firstly, it requires a language definition that explicitly describes the features of the context that this primitive function returns—a `StackFrame` object, for instance. This makes changing the language difficult. Secondly, it exposes aspects of the evaluator that may prevent many optimizations.

The alternative chosen by MzScheme is to allow programs to explicitly associate values with dynamic context and later retrieve them in such a way that they can deduce certain aspects of their own evaluation.

For JavaScript, unlike MzScheme, the natural granularity of such a mark is the procedure activation, or stack frame. The dynamic evaluation of a JavaScript program is defined with respect to stack frames, and programmers are comfortable thinking of their dynamic context as a chain of stack frames.

We therefore introduce the idea of "stack marks," a mechanism for attaching a mark to a stack frame, and retrieving all marks that are associated with still-living stack frames. Since only one set of marks is associated with a stack frame, the mark-storing primitive is a non-local operation, and cannot be modeled as it is in our earlier work on continuation marks, as simply another kind of expression whose presence in the context can be detected by a primitive.

2008 Workshop on Scheme and Functional Programming

$$
\begin{aligned}
M &= n \mid x \mid \texttt{[}M, \ldots\texttt{]} \mid M; M; \ldots \mid \texttt{if0 (}M\texttt{) \{ }M\texttt{ \} else \{ }M\texttt{ \}} \\
&\quad \mid \texttt{function(}x, \ldots\texttt{) \{ }M\texttt{ \}} \mid M\texttt{(}M, \ldots\texttt{)} \mid op\texttt{(}M, \ldots\texttt{)} \\
&\quad \mid \underline{\texttt{tail}}\ M\texttt{(}M, \ldots\texttt{)} \mid \texttt{return}\ M \mid \underline{\texttt{sf}}\ T\ \texttt{\{}\ M\ \texttt{\}} \\
&\quad \mid \texttt{setmark(}M, M\texttt{)} \mid \texttt{getmarks(}M\texttt{)} \\
V &= n \mid \texttt{[}V, \ldots\texttt{]} \mid \texttt{function(}x, \ldots\texttt{) \{ }M\texttt{ \}} \\
F &= [V, \ldots, F, M, \ldots] \mid F; M; \ldots \mid \texttt{if0 (}F\texttt{) \{ }M\texttt{ \} else \{ }M\texttt{ \}} \\
&\quad \mid F\texttt{(}M, \ldots\texttt{)} \mid V\texttt{(}V, \ldots, F, M, \ldots\texttt{)} \mid op\texttt{(}V, \ldots, F, M, \ldots\texttt{)} \\
&\quad \mid \underline{\texttt{tail}}\ F\texttt{(}M, \ldots\texttt{)} \mid \underline{\texttt{tail}}\ V\texttt{(}V, \ldots, F, M, \ldots\texttt{)} \\
&\quad \mid \texttt{return}\ F \mid \texttt{setmark(}F, M\texttt{)} \mid \texttt{setmark(}V, F\texttt{)} \mid \texttt{getmarks(}F\texttt{)} \\
&\quad \mid [\,] \\
E &= \texttt{[}V, \ldots, E, M, \ldots\texttt{]} \mid \ldots \mid \underline{\texttt{sf}}\ T\ \texttt{\{}\ E\ \texttt{\}}
\end{aligned}
$$

where

$$
\begin{aligned}
T \in \text{Mark Tables} &= (V/V, \ldots) \\
x &\in \text{Variables} \\
op &\in \text{PrimOps} \\
\delta &\in op \times \langle V, \ldots \rangle \rightarrow_p V \\
\text{Program States} &= \underline{\texttt{sf}}\ T\ \texttt{\{}\ M\ \texttt{\}} \mid V \mid \text{error}
\end{aligned}
$$

Figure 1. Values, Expressions and Evaluation Contexts for a JavaScript Subset that Supports Stack Marks

Properly tail-calling languages require further refinement to this model. Tail-calling languages guarantee that runtime systems do not needlessly waste memory when one function calls another in tail position—that is, in a position whose result will immediately be the result of the calling function. (Clinger 1998; Sperber et al. 2007). This enables programmers to equate loops and recursive procedures, simplifying the language. In a language that is not properly tail-calling, obtaining good memory behavior often requires an awkward rewriting to ensure that all traversals take the form of loops.

In many tail-recursive language implementations, function calls made in tail position reuse the stack frame of the caller. What should happen to the marks associated with these frames? When using stack marks to build debuggers and security inspection tools, it is vital not to discard this information. The solution of languages like Java is to disable proper tail-calling in the presence of such language features.

A better solution is to decouple the lifetime of the mark from the lifetime of the stack frame by placing it, conceptually, at the boundary between two stack frames. Stack marks may be overwritten by marks placed by tail calls, but they are not discarded by a tail call that does not place marks.

3. Semantics of Stack Marks

Figures 1 and 2 define the syntax and evaluation rules of our language.

Our model adopts JavaScript syntax, while being as simple as possible. Its constants include only numbers, arrays, and functions. It includes sequences (of expressions), and `if0`.

To simplify the model, we choose not to distinguish statements and expressions. However, by including a sequencing form and an explicit `return`, we believe that our language is a plausible superset of a simple JavaScript model, in the sense that we could design a set of static restrictions over the expression language to ensure, for instance, that `return` occurs only in "statement" positions, etc.

Like most statement-based languages, JavaScript makes heavy use of `return`. In order to model `return` in our semantics, we must augment our contexts with information about procedure call boundaries, so that the `return` can discard the right fragment of the context. In our model, this boundary is denoted by the `sf` expression.[1]

Furthermore, modeling `return` requires a bit of thought. If the `return` discards its local context and the `sf` boundary before evaluating its argument, then a nested `return` could return from a function that it is not lexically a part of. Waiting to discard the local context until after the return value is evaluated would destroy proper tail-calling behavior. Finally, removing the context but leaving the `sf` would cause nested calls to pile up `sf`s, leading to a slower-growing but still non-tail-calling implementation.

To solve this problem, we introduce a tail-calling return form, `tail`, with a syntactic restriction and a special reduction form. The syntactic restriction guarantees that the argument can only be a function call; the special reduction rule re-uses the local `sf` boundary in the new call.

As an aside, the additional semantic hardware required to model `return` makes it clearer to us why traditional languages employ the "allocate-on-entry" philosophy that looks so pointless in Clinger's explicit frame-allocation model (Clinger 1998).

We do not expect that programmers should be required to deduce which procedure calls should be labeled as `tail` expressions. All procedure calls as arguments to `return` should be wrapped as `tail` expressions, and figure 3 shows that this is simple. The judgment $C \vdash M \hookrightarrow M'$ inserts annotations in term M, producing a new term M', in a context $C \in \{\text{TAIL}, \text{NON-TAIL}\}$ indicating whether M occurs in tail position. Note that `return tail` M is equivalent to `tail` M, but that automatically removing the `return` would complicate the transformation.

Each stack frame (`sf`) has a set of marks associated with it, possibly empty. The language includes `setmark` and `getmarks` forms for placing and retrieving marks, respectively. These sets are described using a finite function syntax; the implication is that later bindings (extensions) obscure earlier ones. In an implementation of this model, the memory used by obscured bindings may safely be reclaimed.

To define the legal locations for reductions, we use evaluation contexts in the style of Felleisen and Hieb (Felleisen and Hieb 1992). The set of evaluation contexts is defined by the sets E and F. The elements of E are obtained by replacing F's with E's in the definition of F and by adding the `sf` context. The only difference between the two is that F cannot include `sf` boundaries; this restriction is used to decompose evaluation contexts into nested sequences of activation records, and is required, for instance, in the reduction of `return`.

The language definition is parameterized by a set of primitive operations and a partial function δ that maps an operation and a sequence of values to a value. We take these operations to include array lookup.

A program state consists either of an expression with an outermost frame, a value, or the final state "error." Note that this set is closed under reduction; that is, the result of reducing an expression in an outermost frame must be an error, a value, or another expression wrapped in an outermost frame. The requirement that an expression have an outermost frame simplifies the presentation of stack marks and (later) of continuations.

[1] We use the underlining here and elsewhere to indicate synthetic forms; programmers would not normally write these forms, but it presents no problems for the model if they do.

$$
\begin{aligned}
E[V; M; \ldots] &\mapsto E[M; \ldots] \\
E[\texttt{if0 (}V\texttt{) \{}M\texttt{\} else \{}N\texttt{\}}] &\mapsto E[M] \text{ if } V = 0 \\
&\quad\ E[N] \text{ otherwise} \\
E[op\texttt{(}V_1, \ldots\texttt{)}] &\mapsto E[\delta(op, \langle V_1, \ldots\rangle)] \\
&\quad \text{if } \delta(op, \langle V_1, \ldots\rangle) \text{ is defined} \\
E[V_0\texttt{(}V_1, \ldots, V_n\texttt{)}] &\mapsto E[\underline{\texttt{sf}}\ \texttt{() \{}M[V_1/x_1, \ldots, V_n/x_n]\texttt{\}} \\
&\quad \text{if } V_0 = \texttt{function(}x_1, \ldots, x_n\texttt{)\{}M\texttt{\}} \\
&\quad \text{error otherwise} \\
E[\underline{\texttt{sf}}\ T\ \texttt{\{}F[\underline{\texttt{tail}}\ V_0\texttt{(}V_1, \ldots, V_n\texttt{)}]\texttt{\}}] &\mapsto E[\underline{\texttt{sf}}\ T\ \texttt{\{}M[V_1/x_1, \ldots, V_n/x_n]\texttt{\}} \\
&\quad \text{if } V_0 = \texttt{function(}x_1, \ldots, x_n\texttt{)\{}M\texttt{\}} \\
&\quad \text{error otherwise} \\
E[\underline{\texttt{sf}}\ T\ \texttt{\{}F[\texttt{return}\ V]\texttt{\}}] &\mapsto E[V] \\
E[\underline{\texttt{sf}}\ T\ \texttt{\{}V\texttt{\}}] &\mapsto \text{error} \\
E[\underline{\texttt{sf}}\ T\ \texttt{\{}F[\texttt{setmark(}V_1, V_2\texttt{)}]\texttt{\}}] &\mapsto E[\underline{\texttt{sf}}\ T[V_2/V_1]\ \texttt{\{}F[V_2]\texttt{\}}] \\
E[\texttt{getmarks (}V\texttt{)}] &\mapsto E[V_2] \\
&\quad \text{where } E = \underline{\texttt{sf}}\ T_0\ \texttt{\{}F_1[\underline{\texttt{sf}}\ T_1\ \texttt{\{}\ \ldots\ F_n[\underline{\texttt{sf}}\ T_n\ \texttt{\{}F_{n+1}[\,]\texttt{\}}]\ldots\texttt{\}}]\texttt{\}} \\
&\quad \text{and } V_2 = [\texttt{[}T_{a_0}(V), T_{a_1}(V), \ldots\texttt{]} \mid V \in \mathrm{dom}(T_{a_i}), 0 \le a_0 < a_1 < \ldots \le n]
\end{aligned}
$$

Figure 2. Reduction Rules for a JavaScript Subset that Supports Stack Marks

Sequences, applications of primitives, and function calls are reduced normally.

The `setmark` expression associates a key/value pair with the nearest frame boundary and returns the value. The `getmarks` expression extracts an array containing all values associated with the given key in all of the currently active mark tables. This rule uses an "array comprehension" syntax to choose the sub-sequence of the tables (indexed here by the sequence $a_0, a_1, \ldots$) in which the key is bound.

Note that frames without bindings are not represented in the result of `getmarks`. This is deliberate, and differentiates stack marks from a simple stack-disclosure operation. Since the only frames observed by the `getmarks` operation are those containing marks placed by the program, it is not possible, for instance, for a program to observe additional stack frames inserted or removed by an optimizing compiler or a rewriting tool. By limiting the observations possible through `getmarks`, we increase the number of equivalence-preserving transformations.

3.1 Tail Position, Tail Calling

Since marks allow programs to observe tail-call relationships, tail annotation cannot be seen as a meaning-preserving optimization; rather, it is a part of the language evaluator. An evaluator that fails to identify tail calls will produce different results than one that does.

For example, the program

```
sf () {
  setmark(0,1);
  return (function() {setmark(0,2);
                      return getmarks(0)})()}
```

reduces to the value `[1,2]`—that is, there are two mark frames with bindings for the key 0. However, the program

```
sf () {
  setmark(0,1);
  tail (function() {setmark(0,2);
                    return getmarks(0)})()}
```

(which is the result of inserting `tail` annotations in the first program) reduces to the value `[2]`. This is because the tail call forces the reuse of the existing stack frame—or, put differently, forestalls the needless insertion of an extra stack frame.

3.2 Proper Tail Recursion

If we wish to precisely define a notion of proper tail recursion for the JavaScript language, we must follow Clinger (1998) and define a space use function that maps a configuration of an abstract machine onto a number representing space use, and then define as properly tail recursive only those implementations that fall into the same asymptotic space efficiency class as that model.

We conjecture that a straightforward transformation of our reduction semantics into the corresponding abstract register machine would serve as a reasonable definition for proper tail recursion in JavaScript.

4. Adding Stack Marks to Rhino

Mozilla's Rhino is a Java-based implementation of JavaScript. Its source trunk consists of approximately 60K lines of Java. It is designed to be properly tail-calling, and to support first-class continuations.[2]

In order to support stack marks for Rhino, we added a hash table containing marks to each call frame. As the earlier semantics shows, though, we do not want to lose the marks placed during a call's evaluation when it makes a tail call. To preserve marks for their full dynamic duration, the marks placed by a procedure's invocation are stored in the table associated with the parent's call frame.

Placing the marks in the parent call frame, however, creates another problem: when a procedure returns, shortening the chain of call frames, we must ensure that the marks stored during the final frame's existence are explicitly removed. This cleanup cost is in-

[2] These continuations—like our stack marks—are defined at the granularity of frames, as discussed in section 8

$$\frac{\text{NON-TAIL} \vdash M_0 \hookrightarrow M_0' \quad \cdots}{C \vdash \texttt{[}M_0, \ldots\texttt{]} \hookrightarrow \texttt{[}M_0', \ldots\texttt{]}} \qquad \frac{\text{NON-TAIL} \vdash M_0 \hookrightarrow M_0' \quad \cdots}{C \vdash M_0; \ldots \hookrightarrow M_0'; \ldots}$$

$$\frac{\text{NON-TAIL} \vdash M \hookrightarrow M'}{C \vdash \texttt{function(}x, \ldots\texttt{) \{ }M\texttt{ \}} \hookrightarrow \texttt{function(}x, \ldots\texttt{) \{ }M'\texttt{ \}}} \qquad \frac{\text{TAIL} \vdash M \hookrightarrow M'}{C \vdash \texttt{return }M \hookrightarrow \texttt{return }M'}$$

$$\frac{\text{NON-TAIL} \vdash M_0 \hookrightarrow M_0' \quad \cdots}{\text{TAIL} \vdash M_0\texttt{(}M_1, \ldots\texttt{)} \hookrightarrow \underline{\texttt{tail}}\ M_0'\texttt{(}M_1', \ldots\texttt{)}} \qquad \frac{\text{NON-TAIL} \vdash M_0 \hookrightarrow M_0' \quad \cdots}{\text{NON-TAIL} \vdash M_0\texttt{(}M_1, \ldots\texttt{)} \hookrightarrow M_0'\texttt{(}M_1', \ldots\texttt{)}}$$

$$\frac{\text{NON-TAIL} \vdash M_0 \hookrightarrow M_0' \quad \text{NON-TAIL} \vdash M_1 \hookrightarrow M_1'}{C \vdash \texttt{setmark(}M_0, M_1\texttt{)} \hookrightarrow \texttt{setmark(}M_0', M_1'\texttt{)}} \qquad \frac{\text{NON-TAIL} \vdash M \hookrightarrow M'}{C \vdash \texttt{getmarks(}M\texttt{)} \hookrightarrow \texttt{getmarks(}M'\texttt{)}}$$

$$\frac{\text{NON-TAIL} \vdash M_0 \hookrightarrow M_0' \quad \cdots}{C \vdash op\texttt{(}M_0, \ldots\texttt{)} \hookrightarrow op\texttt{(}M_0', \ldots\texttt{)}} \qquad \frac{\begin{array}{c}\text{NON-TAIL} \vdash M_0 \hookrightarrow M_0' \\ C \vdash M_1 \hookrightarrow M_1' \quad C \vdash M_2 \hookrightarrow M_2'\end{array}}{C \vdash \texttt{if0 (}M_0\texttt{) \{ }M_1\texttt{ \} else \{ }M_2\texttt{ \}} \hookrightarrow \texttt{if0 (}M_0'\texttt{) \{ }M_1'\texttt{ \} else \{ }M_2'\texttt{ \}}}$$

Figure 3. Inferring the placement of $\underline{\texttt{tail}}$ annotations

curred whenever a non-tail return is evaluated. We have therefore shifted the administrative burden from once-per-tail-call to once-per-return, and equivalently once-per-non-tail-call. Since both operations are constant-time, neither affects the asymptotic running time of a program. We believe that our choice makes for a somewhat simpler implementation.

Adding support for `setmark` and `getmarks` to Mozilla's Rhino implementation of JavaScript was remarkably easy. We added a field to each call frame and implemented routines to store and retrieve these marks. This required fewer than 100 lines of code. We added support for the `setmark` and `getmarks` primitive operators to the parser, again using fewer than 100 lines of code.

5. A Simple Debugger

In order to check the utility of stack marks in JavaScript, we built a simple experimental debugger that uses stack marks—much like the MzScheme stepper—as its only interface to the runtime system.

Stack marks provide a way to associate key-value pairs with the activation frames and thereby observe the dynamic program state. We built our debugger as an annotator that transforms the source code into target source code with `setmark` constructs. These `setmark` expressions capture the values that the user needs to observe during run time. Our debugger places a `setmark` expression at the beginning of every function body, and before each function call. The mark at the start of the function body captures the values of the arguments and local variables, while the marks inserted before function calls capture information about the source code position where the calls are made.

Our debugger models the breakpoints of a traditional debugger using calls to a `break` function, notable for the fact that it has no privileged access to the runtime system, as it uses the `getmarks` primitive to retrieve and display its information.

Example 1 of figure 4 is a simple JavaScript program, while example 2 is the same program with (substantially simplified) debugging annotations inserted. We see that the debugger has added a `setmark` to the start of the function body to capture the values of the arguments and local variables. We also notice a `setmark` expression before every function call to capture the line number in the source code where the call is made.

Example 1 :

```
function f(m,n){
  if0 (n){
    breakpoint() ;
    Return m;
  }
  else{
    return tail f( n * m , n - 1) ;
  }
}

f(1,1)
```

Example 2 :

```
function f(m,n){
  setmark("key",["f",m,n])
  if0 (n){
    setmark("key",["f",m,n,"breakpoint",3]) ;
    breakpoint() ;
    return m;
  }
  else{
    setmark("key",["f",m,n,"f",6]) ;
    return tail f( n * m , n - 1) ;
  }
}

f(1,1)
```

Figure 4. Inserting Debugging Annotations in a Simple JavaScript Program

5.1 Implementing the Debugger

Implementing the debugger turned out to be substantially harder than adding stack marks to the language. In particular, the debugger requires a syntactic transformation on the source code that is a great deal more difficult in Rhino JavaScript than it would have been in a language with a hygienic macro system.

Lacking such a system, we designed our transformation as a mapping from abstract syntax trees to abstract syntax trees. This was laborious, because the abstract syntax trees generated by Rhino observed internal invariants that were hard to deduce and preserve. For instance, each abstract syntax tree contains a reference to a source string; faking these references in inserted code cost us substantial time and effort.

Another alternative would have been to implement the transformation as a textual source-to-source transformation—essentially, to find or generate our own parser (and its inverse). This would have been more portable, and the end result would have been more easily inspected. Given the syntactic complexity of JavaScript however, this probably would have required even more work.

This experience served to highlight the significance of Scheme's hygienic system. Either of these two approaches would have been substantially simpler in Scheme.

Figure 5 illustrates the debugger output for two simple JavaScript programs. In the first example we see a function that makes recursive calls. When a function call is made a new call frame is allocated and a new set of marks are associated with that frame. In the second example we see a function that makes recursive calls at the tail position. Calls made at tail position reuse the stack frame of the caller. Hence in this case their marks overwrite the marks place by their caller.

5.2 What About Stack Traces?

One objection to such a debugger is that—since it observes the elision of stack frames—it produces less information than a traditional debugger. That is, the call frames reused by tail calls are no longer available for inspection when a breakpoint occurs.

In fact, it turns out that the user can (almost) have his cake and eat it too; by using the mark combination feature described in section 6, the debugger may choose to preserve information about call frames that were reused. This is possible because the lifetime of a mark is decoupled from the lifetime of the call frame in which it was placed.

Naturally, there is memory associated with this mark. If the debugger chooses to retain all such information, then it would alter the asymptotic memory use of the program, effectively turning a tail-calling computation back into a non-tail-calling one. Note that this would not preclude reuse of stack frames; it's simply an observation that this style of debugger must by definition store a quantity of information proportional to the number of nested dynamic calls.

An interesting alternative is implemented in Standard ML of New Jersey(Appel and MacQueen 1991), which allows programmers to specify a bounded "window" of recent tail calls. That is, for each stack frame, the debugger maintains information about the last N tail calls that reused this frame, for a value of N specified by the user. This guarantees that the debugger still behaves in a tail-calling way (assuming a constant bound on the size of the debugging information associated with a single stack frame), and still gives the user most of the information he needs.

A basic observation of functional programming is that loops and recursive procedures are interchangeable, and this suggests applying the prior technique to loops as well, allowing the debugger to capture the values of the loop variables over the prior N iterations without altering the asymptotic memory use of the program.

Example 1 :

```
function f(n){
  if(n == 0){
    breakpoint();
  }
  else{
    return n * f(n - 1);
  }
}

f(1)
```

Output

```
Frame 1
Function: f
Arguments:
n  =  0
Call at line no. 3

Frame 2
Function: f
Arguments:
n =  1
Call at line no. 6
```

Example 2 :

```
function f(m,n){
  if(n == 0){
    breakpoint();
  }
  else{
    return f( n * m , n - 1);
  }
}

f(1,1)
```

Output

```
Frame 1
Function: f
Arguments:
n =  1
n =  0
Call at line no. 3
```

Figure 5. A Simple Debugger

5.3 Benchmarks

To get a rough measure of the cost of using this debugger, we use a pair of micro-benchmarks and also a set of benchmarks from the online "Computer Language Benchmarks Game" (formerly the Great Language Shootout) (Various) to evaluate the performance of Rhino with debugging annotations. We compare the performance of the benchmarks on Rhino with no annotations to that of Rhino with the debug annotations enabled.

The two micro-benchmarks that we used are factorial functions with and without tail calls. For a somewhat more realistic test, we selected five benchmarks from the JavaScript SpiderMonkey Suite. The recursive benchmark makes three function calls all of which in turn make a large number of recursive calls. The partial sum benchmark makes about hundred calls to a function that does a series of arithmetic computations and displays the output to the user. The N-sieve repeatedly makes two function calls both of which perform loop iterations. The N-body makes a series of

	Rhino without annotations	Rhino with debugging annotations
Factorial micro-benchmark without tail calls (input 10^6)	16.38s	58.81s
Factorial micro-benchmark with tail calls (input $5 * 10^6$)	12.68s	69.24s
Recursive benchmark (input 3)	9.68s	26.18s
Partial-sums benchmark (input 10^4)	3.86s	4.72s
The N-body benchmark (input 10^5)	24.07s	26.70s
The N-sieve benchmark (input 10^4)	22.33s	25.82s
Binary-trees benchmark (input 15)	83.33s	289.06s

Table 1. Benchmark Evaluation Results

function calls. The binary tree benchmark builds a binary tree of height obtained from the programmer. The evaluation times are shown in table 1.

Our simple implementation leads to a significant increase in the size of the source code and the call stack. The increase is linearly proportional to the number of local variables, function parameters and function calls inside the function body. Our implementation provides ample opportunity for optimization; for instance, we capture closures that allow us to observe mutation, even for bindings that are never mutated; a simple analysis would allow us to avoid the creation of these unnecessary closures. Earlier experiments suggest an order of 2x speedup on some of these benchmarks.

6. Allowing Mark Combination

In this paper's model, marks overwrite other marks with the same context and key. That is, the statement sequence

```
setmark(<k>,<v1>);
setmark(<k>,<v2>);
```

where k, v1, and v2 are values can safely be replaced by the sequence

```
setmark(<k>,<v2>);
```

because the second mark will overwrite the first one.

However, there are instances in which a programmer may wish for a more expressive construct, that allows marks to be combined, somehow. Take, for instance, the examples of exceptions, stack inspection, contract checks, and gradual typing (Herman et al. 2007); in each of these instances, the information stored in new stack marks is to be combined with the existing information, rather than simply replacing it.

A tempting but faulty approach to this would be to formulate a new mark by combining the new mark with the most recent one, obtained using `getmarks`. That is, if `combine` is a function that takes the old mark value and produces the new one, we might imagine writing

```
var oldMarks = getmarks(<key>);
setmark(<key>,combine(oldMarks
                      [oldMarks.length - 1]));
```

to combine the old mark and the new one. This would not work, because the `getmarks` primitive elides stack frames without a mark for the given key. This means that the final element of the array returned by a `getmarks` might in fact not be on the nearest frame boundary, and that the new mark might not replace the existing one.

Surprisingly, it turns out that the proximity of the nearest mark *can* be deduced, without adding new primitives, *if* the program is willing to commit to providing a new mark regardless of the existence of the old one. Here's an example that adds one to the value associated with key `k` if `k` has a mark binding in the nearest frame, and binds it to one otherwise.

```
var oldMarks = getmarks("k");
setmark("k",78);
var newMarks = getmarks("k");
if (oldMarks.length != newMarks.length)
  setmark("k",1);
else
  setmark("k",oldMarks[oldMarks.length - 1] + 1);
```

Note that this device is contingent upon the program's willingness to place a mark regardless of whether the nearest mark is discovered to be on the current frame or not.

This device is verbose, and slow, requiring the construction of two arrays. However, once we've demonstrated that this operation is expressible in the current language, we can add the corresponding primitive to the language, without fear that it needlessly increases the expressiveness of the language.

Figure 6 illustrates this extension to the language, using a primitive `setmark_c` that accepts a key, as before, but in place of a value has instead a procedure that will be applied in order to compute the new value. In order to distinguish "no-value" from any particular value, this procedure's argument is wrapped in an array, which is of length zero if no mark is present in this frame.

$$M = \ldots \mid \texttt{setmark\_c}(M, M)$$
$$E = \ldots \mid \texttt{setmark\_c}(E, M) \mid \texttt{setmark\_c}(V, E)$$
$$F = \ldots \mid \texttt{setmark\_c}(F, M) \mid \texttt{setmark\_c}(V, F)$$

and

$$E[\underline{\texttt{sf}}\ T\ \{\ F[\texttt{setmark\_c}(V_1, V_2)]\ \}] \mapsto$$
$$E[\underline{\texttt{sf}}\ T\ \{\ F[\texttt{setmark}(V_1, V_2([V_3]))]\ \}]$$
$$\text{if } T(V_1) = V_3$$
$$E[\underline{\texttt{sf}}\ T\ \{\ F[\texttt{setmark}(V_1, V_2(\texttt{[]}))]\ \}]$$
$$\text{otherwise}$$

Figure 6. Adding a mark-combination primitive

7. Dynamic Bindings

Many languages provide the ability to create dynamic bindings: a current output port, a log file, or a current privilege level. For instance, this is the behavior of Moreau's "dynamic let" (Moreau 1998). One obvious way to implement such bindings in a traditional imperative language is simply to bracket the expression in which the binding is to be created with a pair of mutations to a global variable or table:

```
var oldPrivileges = privileges;
privileges = noPrivileges;
```

```
<some code>
privileges = oldPrivileges;
```

However, stack marks provide a natural way to implement such bindings, with two natural advantages over the typical "global mutations bracketing an expression" solution. First, they preserve tail calling, because no additional work is required to undo the binding. Secondly, they respect context manipulations. That is, a computation that abandons a portion of the context will also abandon the corresponding bindings. Similarly, a computation that reinstates a context will also reinstate the bindings associated with that context.

The natural way to implement dynamic bindings using stack marks would be to associate a unique key with each binding, to use `setmark(<key>,<new-value>)` to bind the value, and to refer to the last element of the array generated by `getmarks(<key>)` to obtain the most recent dynamic binding.

This solution fails to model dynamic let in the same way that a JavaScript variable binding fails to model a "let". That is, its extent must be determined by the end of the procedure call.

If the programmer wishes to limit the dynamic extent of the binding, restoring the original binding before the procedure call has finished, the program must restore the earlier bindings by explicitly overwriting the existing one. Unfortunately, this means that that leaving the scope of a dynamic binding may require the placement of a "dummy," or false mark. This is not a serious problem, because the progrmmer will be aware of these dummies, and because the number of such dummies is bounded by the number of stack frames currently active. However, it does provide an additional small piece of evidence for the clumsiness of frame-based models.

8. First-Class Continuations

Many dynamic languages provide some facility for manipulating dynamic context: continuations, delimited continuations, exceptions, and the like. Rhino supports "call/cc"-style continuations (Sperber et al. 2007), with a twist; since JavaScript's notion of context is inextricably bound to stack frames, the continuation-capture operation captures only the portion of the context outside the nearest frame boundary.

To model this, we can introduce a new continuation-capturing primitive, and a new class of continuation values, shown in figure 7. The reduction rules in figure 8 for continuation capture show the truncation of the context at the innermost `sf` frame boundary.

Note that programmers who wish to obtain the more standard sort of continuation may do so by wrapping the `NewCont` in an immediately-called thunk: `function(){NewCont}()`.

Since stack marks are a part of the continuation, capture and invocation of continuation naturally respects stack marks. That is, a computation that abandons a portion of the context will also abandon the corresponding stack marks. Similarly, a computation that reinstates a context will also reinstate the bindings associated with that context.

Extending the tail-inserting $\vdash_S$ relation to include the new language forms is straightforward though syntactically cumbersome because of the need to apply it to the evaluation contexts inside of continuation values.

$$M = \ldots \mid \texttt{NewCont} \mid \underline{\langle\!\langle E \rangle\!\rangle}$$
$$V = \ldots \mid \underline{\langle\!\langle E \rangle\!\rangle}$$

Figure 7. Extending the language to include continuations

$$E[\underline{\texttt{sf}}\ T\ \texttt{\{}\ F[\texttt{NewCont}]\ \texttt{\}}] \mapsto E[\underline{\texttt{sf}}\ T\ \texttt{\{}\ F[\underline{\langle\!\langle E[\underline{\texttt{sf}}\ T\ \texttt{\{}\ [\,]\ \texttt{\}}] \rangle\!\rangle}]\ \texttt{\}}]$$
$$E[\underline{\langle\!\langle E' \rangle\!\rangle}(V)] \mapsto E'[\texttt{return}\ V]$$

Figure 8. Reduction rules for continuation operations

9. Exceptions with Tail Calls

In a language with dynamic bindings and continuations, we may implement exceptions without direct support for them, and without losing the ability to reuse stack frames. In particular, a `try` block consists of the dynamic binding of an exception-handler key to a new exception-handling procedure that contains a reference to the existing exception handler. A `throw` consists of an invocation of the most recent exception handler with the value being thrown.

Rather than precisely define the transformation, we illustrate it with an example. To simplify the reading of the example, we include several features that are not a part of the model that we have provided. We use strings in place of numbers to make our constants more legible, we assume the existence of local variables. Our set of primitives is taken to include array reference and manipulation operations.

Finally, this example refers to several functions whose implementations are straightforward. In particular:

- pushMaker : a curried function that takes a and `[]` to `[`a`]`, and a and `[[`b`, ...]]` to `[`b`, ..., `a`]`
- popMaker : a function that takes `[]` to `[]` and `[[`b`, ..., `a`]]` to `[`b`, ...]`.
- lastExnHandler : a function that takes `[...,` `[`a`, ..., `b`]`, `[]`, `...]` to b (that is, it finds the last element of the flattened array).

With these assumptions, we can transform the program:

```
g();
try {
 h();
} catch (err) {
 i();
}
j();
```

... into:

```
g();
var t = function() {return [0,NewCont];}();
if0 (t[0]) {
  setmark_c ("exnkey",pushMaker(t[1]));
  h();
  setmark_c ("exnkey",popMaker);
} else {
  var err = t[1];
  setmark_c ("exnkey",popMaker);
  i();
}
j();
```

The try block is modeled by a continuation capture, followed by an `if0` test that has the flavor of a `fork` system call. If the first element of the returned array is 0, then this is the return from the initial continuation capture, and the body of the try block is evaluated. If the first element of the returned array is 1, then this is the invocation of an exception, and we evaluate the catch block with `err` bound to the thrown value. In either case, we continue by evaluating the remainder of the body.

The implementation of `throw` is simpler:

```
throw "someExn";
```

. . . turns into:

```
lastExnHandler(getmarks("exnhandler"))
   ([1,"someExn"]);
```

Note that this implementation of exceptions permits the placement of `tail` calls in the body of the try block. In such a scenario, the remainder of the body is safely captured in the stored continuation, which appears in a mark, and it is safe to reuse the existing stack frame for a tail call.

Furthermore, this implementation technique does not invalidate the earlier specification for the placement of `tail` expression wrappers. That is, we may take a program that uses this technique for exceptions and apply the earlier transformation and obtain a program that operates as we expect exceptions to.

10. Stack Inspection

In this paper, we have by no means done justice to all the possible language tools and features that may be built upon stack marks. Stack inspection is one of these features.

Stack inspection (Wallach et al. 1997, 2000; Karjoth 2000; Gong 1999; Fournet and Gordon 2002) is a security protocol that attempts to prevent malicious code from evaluating certain expressions. The core idea is that when the evaluator reaches a "risky" expression, it examines the stack frames (hence the name) to make sure that there is an unbroken chain of "trusted" frames from the current frame all the way to a trusted frame that has explicitly authorized this kind of risky expression.

Stack inspection would appear to be incompatible with tail calling, since the stack frames no longer correspond in a one-to-one way with the dynamically nested calls, and it may be the case that an untrusted frame has been reused, thus making a check that should fail into one that passes.

However, it turns out that this is not the case (Clements and Felleisen 2004). By placing the information about trust into stack marks, and by using a mark-combining protocol, we can guarantee that this information is not lost. In fact, we can prove that if the set of risky categories is bounded, then an implementation that tracks this information using stack marks consumes asymptotically no more space than one for which this information is asserted to take no space at all.

This work applies without change to JavaScript.

11. Related Work

This paper builds directly upon earlier work on MzScheme's continuation marks (Clements et al. 2001; Flatt 1995–2008).

This paper is also indebted to other earlier work on the basic notion of building a debugger based on annotation (Tolmach and Appel 1995; Kellomäki 1993).

This paper attempts to formulate a notion of tail calls for statement-based languages. In many ways, this draws directly on Landin's SECD machine (Landin 1964), and is also related to Ramsdell's tail-recursive SECD machine (Ramsdell 1999). Schinz and Odersky (Schinz and Odersky 2001) discuss the problems of compiling tail-calling languages into a language whose evaluator (the JVM) does not provide support for tail calls.

12. Conclusion

Adding stack marks to Rhino was refreshingly straightforward. Though this prevents us from claiming victory over a difficult technical problem, it strongly validates our central claim, which is that stack marks are an easy addition to most stack-based languages.

Our work thus far has been on the simplest possible implementation of stack marks, and on the simplest possible definition of a debugger that uses them. A natural next step would be to seek to improve the implementation of stack marks, to show how to add them to Rhino's compiling evaluator as well as its "level-zero" evaluator, and to improve the utility and efficiency of the associated debugger.

References

Andrew W. Appel and David B. MacQueen. Standard ML of new jersey. In *plilp*, volume 528. Springer Berlin / Heidelberg, 1991.

John Clements. *Portable and high-level access to the stack with Continuation Marks*. PhD thesis, Northeastern University, 2005.

John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1–24, November 2004.

John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2001.

William D. Clinger. Proper tail recursion and space efficiency. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, 1998.

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

Matthew Flatt. PLT MzScheme: Language manual. Online at `<http://www.plt-scheme.org>`, 1995–2008.

Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *icfp*, 2007.

Cedric Fournet and Andrew D. Gordon. Stack inspection: theory and variants. In *ACM SIGPLAN Conference on Principles of Programming Languages*, pages 307–318, 2002.

Li Gong. *Inside Java 2 Platform Security*. Sun Microsystems, 1999.

Dave Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming*, 2007.

Günter Karjoth. An operational semantics of Java 2 access control. In *The Computer Security Foundations Workshop*, pages 224–232, 2000.

Pertti Kellomäki. Psd—a portable scheme debugger. *Lisp Pointers*, VI(1), 1993.

P. J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6(4):308–320, 1964.

Luc Moreau. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, 1998.

John D. Ramsdell. The tail-recursive SECD machine. *Journal of Automated Reasoning*, 23(1):43–62, July 1999.

Michel Schinz and Martin Odersky. Tail call elimination on the Java virtual machine. In *SIGPLAN BABEL Workshop on Multi-Language Infrastructure and Interoperability*, pages 155–168, 2001.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, William Clinger, and Jonathan Rees. Revised^6 report on the algorithmic language scheme. online at `http://www.r6rs.org/`, 2007.

Andrew P. Tolmach and Andrew W. Appel. A debugger for standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

Various. The computer language benchmarks game, spidermonkey suite. On the web at `http://shootout.alioth.debian.org/gp4/`.

Dan Wallach, Dirk Balfanz, Drew Dean, and Ed Felten. Extensible security architectures for Java. In *The 16th Symposium on Operating Systems Principles*, pages 116–128, october 1997.

Dan Wallach, Edward Felten, and Andrew 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):341–378, October 2000.