Compiler and Runtime Support for Continuation Marks

Matthew Flatt
University of Utah
USA
mflatt@cs.utah.edu

R. Kent Dybvig
Cisco Systems, Inc.
USA
dyb@cisco.com

Abstract

Continuation marks enable dynamic binding and context inspection in a language with proper handling of tail calls and first-class, multi-prompt, delimited continuations. The simplest and most direct use of continuation marks is to implement dynamically scoped variables, such as the current output stream or the current exception handler. Other uses include stack inspection for debugging or security checks, serialization of an in-progress computation, and run-time elision of redundant checks. By exposing continuation marks to users of a programming language, more kinds of language extensions can be implemented as libraries without further changes to the compiler. At the same time, the compiler and runtime system must provide an efficient implementation of continuation marks to ensure that library-implemented language extensions are as effective as changing the compiler. Our implementation of continuation marks for Chez Scheme (in support of Racket) makes dynamic binding and lookup constant-time and fast, preserves the performance of Chez Scheme's first-class continuations, and imposes negligible overhead on program fragments that do not use first-class continuations or marks.

CCS Concepts: • **Software and its engineering → Compilers; Runtime environments; Control structures**.

Keywords: Dynamic binding, context inspection

ACM Reference Format:
Matthew Flatt and R. Kent Dybvig. 2020. Compiler and Runtime Support for Continuation Marks. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3385412.3385981

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3385981

1 Binding and Control

Suppose that a program needs to call a function with output redirected to a file instead of the current default output stream. One way a language can support such redirection is by having a global variable like `stdout` hold the default output destination for functions like `printf`, and then a program can temporarily change the value of `stdout` and restore it when the function call returns:

```
FILE *orig_stdout = stdout;
stdout = f_output;
func();
stdout = orig_stdout;
```

This approach has several potential problems. If the host language supports threads, then `stdout` must be a thread-local variable so that redirecting output in one thread does not interfere with other threads. If the host language supports exceptions, then a form analogous to `try-finally` is needed to ensure that `stdout` is reset on an exception escape. If the host language supports proper handling of tail calls, then `func` no longer can be called in tail position (in contrast to passing the output stream explicitly to `func` and having it threaded throughout `func`'s computation), which might limit the use of this kind of redirection. If the host language supports first-class continuations, then in case a continuation is captured during the call to `func`, `stdout` should be set and restored using a mechanism like Scheme's `dynamic-wind` (which is a kind of generalization of `try-finally`), but that adds a winding cost for jumping into or out of a continuation.

All of these issues are related to using global state to track an intent about a program's dynamic extent. A language implementation must already include some explicit representation of dynamic extent as a *continuation*, whether or not that continuation is exposed as a first-class value. The continuation may be implemented as a call stack, for example. The language implementer must have taken care to select a representation of the continuation that is efficient and expressive enough for the language's control constructs. So, instead of having users of a language track dynamic extent through external state, the language should include built-in constructs to reflect on continuations, provided that supporting these additional constructs does not unduly burden the implementation. Built-in operations enable a more expressive and efficient implementation that takes advantage of internal representations of continuations.

In Racket, the output-redirection example is written

```
(parameterize ([current-output-port f-output])
  (func))
```

where current-output-port is defined to hold a dynamically scoped binding, and that binding is used by functions like printf. In this example, the output stream is set to f-output during the call to func, which is called in tail position with respect to the parameterize form and might capture its continuation. If it does capture the continuation, the dynamic binding of current-output-port to f-output sticks with the continuation without increasing the cost of capturing or restoring the continuation. The parameterize form is a library-implemented language extension (i.e., a macro), but it relies on a lower-level form that is built into the core language and compiler: *continuation marks* [7].

Like closures, tail calls, and first-class continuations, *continuation marks* enable useful language extensions without further changes to the compiler's core language. Continuation marks in Racket have been used to implement dynamic binding [17], debugging [8, 22], profiling [1], generators [16 §4.14.3, 26], serializable continuations in a web server [23], security contracts [24], and space-efficient contracts [14].

To support a Racket implementation on Chez Scheme [15], we add *continuation attachments* to the Chez Scheme compiler and runtime system. This even-simpler core construct supports a layer that implements Racket's rich language of continuation marks and delimited continuations [17], which in turn supports Racket's ecosystem of library-implemented languages and language extensions. The performance of continuation marks in Racket on Chez Scheme compares favorably with the traditional implementation of Racket, which sacrifices some performance on every non-tail call to provide better performance for continuation marks.

Our implementation of continuation attachments for Chez Scheme is nearly pay-as-you go, imposing a small cost on programs that use first-class continuations and dynamic-wind and no cost on programs that do not use them (or continuation marks). The implementation is modest, touching about 35 (of 18k) lines in Chez Scheme's C-implemented kernel and 500 (of 94k) lines in the Scheme-implemented compiler and run-time system. Compiler-supported continuation attachments perform 3 to 20 times as fast as an implementation without compiler support; for Racket on Chez Scheme, that improvement makes continuation marks perform generally better than in the original Racket implementation.

Relatively few languages currently offer first-class continuations, much less the richness of Racket's control constructs. However, interest is growing around delimited continuations, particularly in the form of algebraic effect handlers [25], and the implementation strategies for effect handlers are the same as for delimited continuations with tagged prompts. Although all control and binding constructs (including continuation marks) can be encoded with effect handlers or delimited continuations, recent work on effect handlers includes direct semantic support for specialized constructs, partly on the grounds that they can be more efficient [9, 10] including for dynamic binding [5]. We are following a similar path, but with a baseline implementation of continuations that tends to be much faster already compared to other existing implementations.

Contributions in this paper:

- We offer the first report on implementing direct compiler and runtime support for continuation marks.
- We demonstrate that the implementation of continuation marks is compatible with a high-performance implementation of first-class, delimited continuations.
- We demonstrate that compiler and runtime support for continuation marks can improve the performance of applications.

2 Using Continuations Marks

The core Racket constructs for continuation marks include operations to set a mark on the current continuation and to get marks of any continuation.

2.1 Setting Marks

The expression form

```
(with-continuation-mark key val body)
```

maps *key* to *val* in the current continuation and evaluates *body* in tail position. Since *body* is in tail position, its value is the result of the with-continuation-mark expression.

For example,

```
(with-continuation-mark 'team-color "red"
  (player-desc))
```

produces the result of calling player-desc while the key 'team-color is mapped to the value "red" during the call.

If *key* is already mapped to a value in the current continuation frame, then with-continuation-mark replaces the old mapping in that frame. If *key* is instead set in a more nested continuation frame, then the nested mapping for *key* is left in place while a new mapping is added to the current frame.

For example, in

```
(with-continuation-mark 'team-color "red"
  (place-in-game
   (player-desc)
   (with-continuation-mark 'team-color "blue"
     (all-teams-desc))))
```

then the call to player-desc sees 'team-color mapped to "red", while the call to (all-teams-desc) sees 'team-color immediately mapped to "blue" and also mapped to "red" in a deeper continuation frame; overall, it sees `(list "blue" "red")` as a list of mappings for `'team-color`. Whether a list of mappings or only the newest mapping is relevant depends on how a key is used in a program.

2.2 Extracting Marks

The function call

```
(continuation-marks continuation)
```

extracts the continuation marks of *`continuation`* as an opaque *mark set* in amortized constant time. A separate mark-set representation is useful (e.g., in an exception record) for keeping just a continuation's marks without its code, but it is not fundamentally different from accessing marks directly from a continuation.

A convenience function call

```
(current-continuation-marks)
```

captures the current continuation with `call/cc` and passes it to `continuation-marks` to get its marks.

The most general way to inspect continuation marks is

```
(continuation-mark-set->iterator set (list key ...))
```

which produces an iterator for stepping through frames that have values for at least one of the *`key`*s in the given list. The iterator reports values in a way that indicates when multiple *`key`*s have values within the same continuation frame, and it provides access to mark values in time proportional to size of the continuation prefix that must be explored to find the values.

The convenience function call

```
(continuation-mark-set->list set key)
```

returns a list of values mapped by *`key`* for continuation frames (in order from newest to oldest) in *`set`*. The time required to get all keys can be proportional to the size of the continuation.

The function call

```
(continuation-mark-set-first set key default)
```

extracts only the newest value for *`key`* in *`set`*, returning *`default`* if there is no mapping for *`key`*. Although a similar function could be implemented by accessing the first value in a sequence produced by `continuation-mark-set->iterator`, the `continuation-mark-set-first` function works in amortized constant time no matter how old the newest continuation frame that contains a value for *`key`*. As a convenience, `#f` is allowed in place of *`set`* as a shorthand for `(current-continuation-marks)`.

For example, if `player-desc` as used above calls `current-team-color` defined as

```
(define (current-team-color)
  (continuation-mark-set-first #f 'team-color "?"))
```

then `(current-team-color)` will return `"red"`. If the `all-teams-desc` function calls

```
(define (all-team-colors)
  (continuation-mark-set->list (current-continuation-marks)
                               'team-color))
```

then the result is `(list "blue" "red")`.

The function call

```
(call-with-immediate-continuation-mark key proc default)
```

is similar to `continuation-mark-set-first`, but instead of getting the first continuation mark in general, it gets the first mark only if the mark is on the current continuation frame. Also, instead of returning the mark value, the value is delivered to the given *`proc`*, which is called in tail position. (A primitive that directly returns a mark value would not be useful, because calling the function in a non-tail position would create a new continuation frame.)

2.3 Implementing Exceptions

As a practical application of continuation marks, consider the problem of implementing exceptions. A `catch` form and `throw` function should cooperate so that

```
(catch (λ (v) (list v))
  (+ 1 (throw 'none)))
```

produces `(list 'none)`, because the `throw` in the body of the `catch` form escapes from the `+` expression and calls the handler procedure associated with `catch`.

Here's an implementation of `catch` and `throw` using a private `handler-key`:

```
(define handler-key (gensym))

(define (throw exn)
  (define escape+handle
    (continuation-mark-set-first #f handler-key #f))
  (if escape+handle
      (escape+handle exn)
      (abort "unhandled exception")))

(define-syntax-rule (catch handler-proc body)
  ((call/cc
    (λ (k)
      (λ ()
        (with-continuation-mark
          handler-key (λ (exn)
                        (k (λ () (handler-proc exn))))
          body))))))
```

The extra parentheses around `(call/cc ...)` allow a thunk to be returned to the captured continuation, so a procedure associated to `handler-key` can escape before it calls the given *`handler-proc`*. To match that protocol, the function returned by the argument to `call/cc` wraps the use of `with-continuation-mark` and *`body`* in a thunk; that thunk goes to the same current continuation and ensures that *`body`* is in tail position with respect to the `catch` form.

Instead of always escaping, `throw` could give the handler the option of recovering from the exception by returning a value. But if a handler is called without first escaping, then what if the handler reraises the exception or raises another exception to be handled by an outer handler? We can get the full stack of handlers using `continuation-mark-set->list` and loop over the stack:

```
(define (throw exn)
  (define escape+handles
    (continuation-mark-set->list
     (current-continuation-marks)
     handler-key))
  (throw-with-handler-stack exn escape+handles))

(define (throw-with-handler-stack exn escape+handles)
  ....)
```

While calling each handler, `throw-with-handler-stack` can wrap the call with a new handler that escapes to continue looping through the handler stack.

Although we could map `handler-key` to a list of handlers to manage the stack ourselves, an advantage of using `continuation-mark-set->list` is that delimited and composable continuations will capture and splice subchains of exception handlers in a natural way. Also, using `continuation-mark-set->iterator` would be better to access the sequence of handlers, since that avoids a potentially large up-front cost (proportional to the size of the continuation) to get the first handler.

Having `catch` evaluate its body in tail position conflicts somewhat with the idea of a stack of handlers, because a body that that starts with another `catch` will replace the current handler instead of chaining to it. We can have our cake and eat it too by mapping `handler-key` to a list of handlers, where the list is specific to the current continuation frame, as opposed to keeping the full chain of handlers in a list:

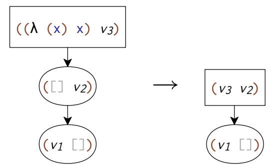
```
(define-syntax-rule (catch handler-proc body)
  (call-with-immediate-continuation-mark
   handler-key
   (λ (existing-handlers)
      ....
      (with-continuation-mark
        handler-key (cons (λ (exn) (handler-proc k exn))
                          existing-handlers)
        ....))
   null))
```

With this change, `throw` must inspect a list of handler lists instead of just a list of handlers, but that's just a matter of flattening the outer list or using a nested loop.

The body of a `catch` form is an unusual kind of tail position where the continuation proper does not grow, but the stack of exception handlers does grow. Whether this is a good design is up to the creator of a sublanguage for exception handling. The point of continuation marks is to put that decision in the hands of a library implementer instead of have one answer built into the compiler.

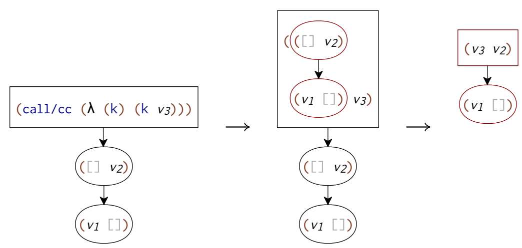
3 A Model of Continuations and Marks

In a reduction-semantics model of a programming language, the continuation of a currently evaluating expression e_1 is the surrounding expression that is waiting for e_1's value. For example, to evaluate $(v_1\ (((\lambda\ (x)\ x)\ v_3)\ v_2))$, the inner call $((\lambda\ (x)\ x)\ v_3)$ is reduced to the value v_3, which is delivered to the enclosing expression $(v_1\ ([]\ v_2))$ by putting v_3 in place of $[]$. The value of the inner expression $(v_3\ v_2)$, in turn, will be delivered to the enclosing $(v_1\ [])$. If we break up the context $(v_1\ ([]\ v_2))$ into its two stages $(v_1\ [])$ and $([]\ v_2)$, and if we draw them as a vertical sequence, we end up with a picture that looks like an upward-growing control stack. Completing an evaluation pops a stack frame to return a value down the stack:



The chain of ovals in this picture is a continuation. The rectangle at the top holds the current expression to evaluate. Each individual oval or rectangle corresponds to a continuation frame.

The `call/cc` function captures the current continuation and passes it as an argument to a given function. In the following picture, the first step of evaluation captures the continuation and passes it as the argument `k` to the function $(\lambda\ (k)\ (k\ v_3))$, so the captured continuation replaces `k` in the function body $(k\ v_3)$:

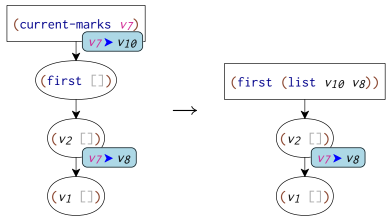


When a continuation is applied like a function, as it is here to the argument v_3, then the next step discards the current continuation and delivers the application argument to the applied continuation. So, in the second step above, the current continuation is replaced by the captured one (although they happen to be the same), and then the value v_3 is delivered to the continuation frame $([]\ v_2)$, so the new expression to evaluate is $(v_3\ v_2)$.

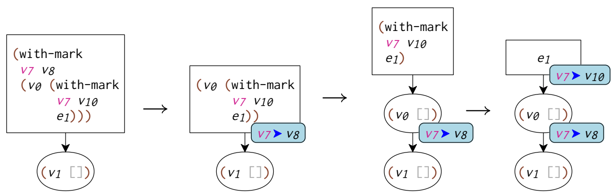
The picture above illustrates capture and application of a non-delimited, non-composable continuation, but the same ideas and pictures apply straightforwardly to delimited, composable continuations [17].

A continuation mark is a key–value mapping that is attached to a continuation frame. We draw marks as badges attached to the lower right of a frame. A `current-marks` function takes a key and returns a list of all values for the key in badges of the current continuation frames:[1]

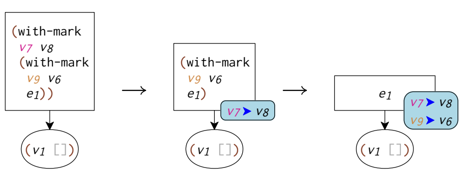
[1]The model's `current-marks` function is `continuation-mark-set->list` composed with with `current-continuation-marks`.



A `with-mark` form (short for `with-continuation-mark`) adds a continuation mark given a key, value, and body expression. It evaluates the body in tail position while adding a key–value mapping to the current frame. In the following example, a `with-mark` form has another `with-mark` form in non-tail position, so two continuation frames end up with marks:



In the following example, the body of the outer `with-mark` form has another `with-mark` form in tail position, so both marks end up on the same frame:



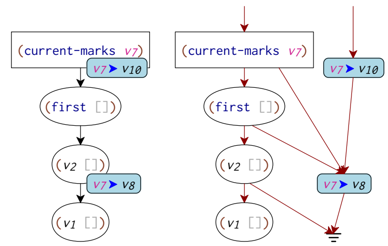
Both marks end up on the final frame because they use different keys, *v7* and *v9*. If they had used the same key, then the second mark would replace the first part, leaving just one mark on the frame.

If a captured continuation includes marks, then the captured marks are carried with the continuation, just as the pictures would suggest. Also as the pictures suggest, capturing a continuation does not need to retain the marks on the current evaluation frame (i.e., the rectangular frame), since there would be no way to inspect those marks when the continuation is later applied to a value; application of the continuation immediately turns the topmost oval of the captured continuation into an evaluation rectangle.

4 Heap-Based Continuations and Marks

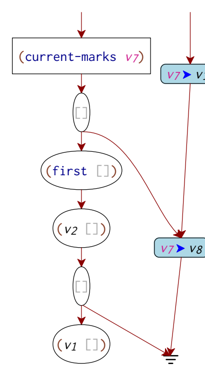
The pictures of the previous section are intended for understanding the semantics of continuation marks, but they also suggest an implementation: allocate continuation frames as a linked list in the heap, and similarly allocate and chain sets of marks. Instead of having a frame directly reference its own marks, however, we should pair any reference to a frame with a reference to the frame's marks. That way, a continuation can be captured and/or applied without copying, since a frame is not mutated when its marks are updated by further evaluation. For the current frame (the rectangular one), use a separate global register for marks alongside the one for the heap-allocated frame.

Applying these ideas to the continuation shown on the left below, we arrive at a pointer structure on the right:



A naive implementation of this strategy would evaluate a primitive arithmetic expression like `(+ (+ 1 x) 3)` by creating a continuation frame for the intermediate `(+ [] 3)` expression, which is unlikely to perform well. A practical implementation will instead create frames only around subexpressions that are function calls.[2] For the moment, assume that only continuation frames for function calls have marks. Marks for other (conceptual) continuation frames will be easy to handle, because we can locally push and pop mark records for those.

Another problem with the simple strategy is that it adds a reference to every continuation frame, imposing a potentially significant cost on parts of a program without continuation marks. We can avoid this cost by introducing extra continuation frames at only points where marks have changed:



These extra frames just pass through whatever value they receive, but they also adjust the global register for the current marks to a previously saved pointer. Other continuation frames can return while leaving the mark register as-is. The extra frame must be created by `with-mark` whenever the current continuation does not already start with a mark-restoring frame, so it must be able to recognize and create this distinct kind of frame.

Capturing a continuation must still pair the current continuation pointer with the current marks pointer, which adds

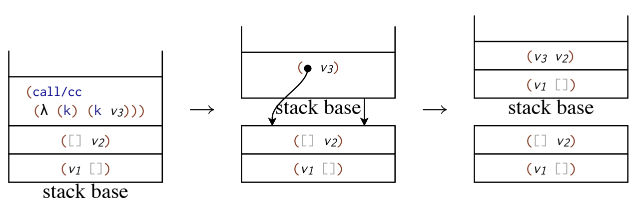
[2] With the practical adaptation to create continuation frames only around function calls, we could view a continuation frame as created by a function call instead of by an expression surrounding a function call—as long as tail calls within a function reuse the current call's frame or discard it before creating a new one.

some overhead to program regions that use first-class continuations but do not use continuation marks. In practice, those regions are much rarer than ones that create new continuation frames. Also, capturing a continuation tends to require extra allocation, perhaps to package the continuation pointer as a value that can be used like a function, so there would be just one extra pointer move in each of a continuation capture and application.

5 Stack-Based Continuations

A potential concern with our strategy so far is that it heap-allocates continuation frames. Although an analysis by Appel and Shao [2] suggests that heap-allocated frames can be nearly as efficient as stack-allocated frames, the analysis makes two assumptions about the implementation of stack-allocated frames that do not hold for Chez Scheme.[3] Removing the costs associated with these assumptions from their analysis results in a much lower cost for stack-allocated frames than for heap-allocated frames. We thus consider the implementation of continuation marks for a stack-based representation of continuations that retains this advantage.

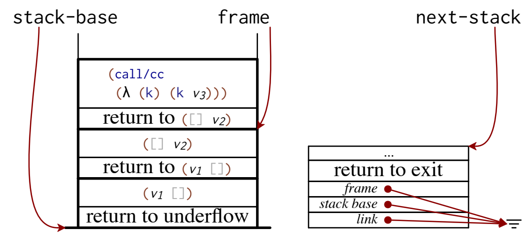
In a language with first-class continuations, a stack-based representation of continuations requires copying either on capture or application of a continuation. Chez Scheme's hybrid stack–heap representation of continuations [18] puts copying on the continuation-application side. When `call/cc` captures a continuation, the current stack ceases to be treated as stack space and instead is treated as part of the heap, while a new stack is started for the frame of the function that `call/cc` calls, as shown in the first step below:



In the second step above, applying the continuation copies the frames that are (now) in the heap to the new stack. To avoid unbounded copying, application may split large continuations so that only part of the continuation is copied, and the rest is deferred until later [18].

If the expression in the example above had been just v_3 instead of (k v_3), the result would be the same: the second step would copy the captured continuation (referenced by the right-hand arrow) onto the new stack to return v_3 to that continuation. The result is the same because the two arrows in the middle figure both refer to the same continuation.

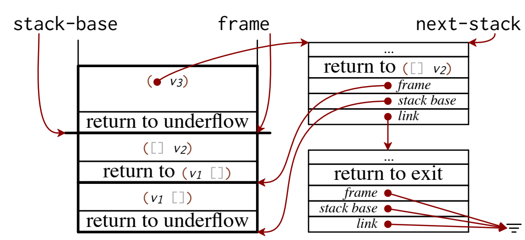
Still, the arrows in that figure must be different kinds of arrows, because one of them is wrapped as a procedure that jumps to the continuation, while the other must be used by a normal function-call return. To see the difference, we can zoom into the first two figures and refine the picture, starting with the left figure:



In this expanded view, we show that there's a global `stack-base` register that points to the stack base as well as a `frame` register for the current frame. Each stack frame, outlined with a bold rectangle, starts with a return address as shown at the bottom of the frame. For example, the frame for the `call/cc` call returns to the function that has ([] v_2), and so on. The first frame in a stack always has a return address to a special underflow handler, which deals with returning from this stack.

The underflow handler consults a global `next-stack` register, which points to a record that has an actual return address, plus pointers to a stack base to restore and a chain to the next underflow record. In the example, the return address is to process exit, which needs no stack.

After `call/cc` captures the continuation and passes it to the function whose body is (k v_3), the stack and registers look like this:

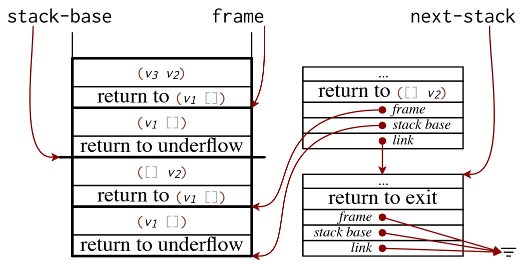


- The "new" stack is just the remainder of the old stack after the captured part. The `stack-base` register has changed to point to the new stack base.
- The stack frame that used to have a return to ([] v_2) now has a return to the underflow handler.
- The old return address to ([] v_2) is in a new underflow record, and the `next-stack` register points to the new record. The new underflow record also saves the old `stack-base`, `frame`, and `next-stack` values.

An underflow record starts with extra data that makes it have the same shape as a procedure. In other words, a continuation procedure is represented directly as an underflow record. Calling the underflow record as a procedure jumps

[3] These assumptions are (1) A stack-overflow check is required for each non-tail call. Chez Scheme typically performs at most one stack-overflow check no matter how many non-tail calls a caller makes. (2) Closures tend to be large. This assumption held in SML/NJ, the heap-based implementation upon which Appel and Shao based their analysis, because each closure for a function included the values of each primitive and global variable referenced by the function. Chez Scheme embeds pointers to (or inlines) primitives and global-variable locations directly in the code stream.

to the underflow handler, which copies the stack as recorded in the underflow record to the current `stack-base`, adjusts `frame` to point to the newly copied frame, restores the `next-stack` register to the one saved in the underflow record, and jumps to the return address saved in the underflow record:



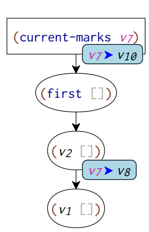
At this point, the upper underflow record and lower stack are garbage and can be reclaimed.

If the body of the function passed to `call/cc` were just *v3* instead of `(k` *v3*`)`, then a return of *v3* would use the address in the newest continuation frame. Since that address is the underflow handler, the same changes take place, resulting in the same ending picture.

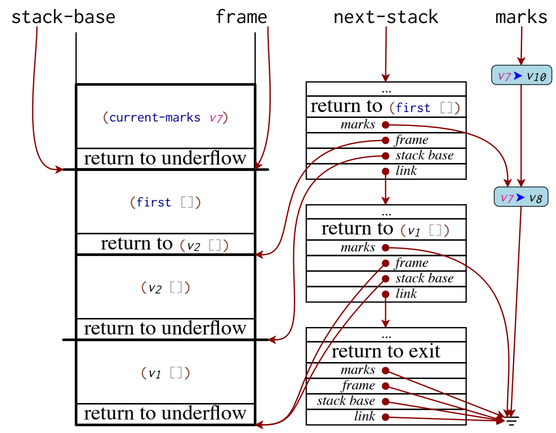
6 Stack-Based Marks

Section 5 provides a recap of Chez Scheme's stack-based representation of continuations, and section 4 sketches how to manage continuation marks alongside a heap-allocated continuation. In this section, we bring those together to represent marks with a stack-based continuation.

Our main idea is to *reify* the continuation whenever we need to attach a mark to a continuation frame, doing so in the same way as `call/cc`. We also make an underflow record point to a mark chain for the rest of the continuation in the same way that a continuation frame in the heap-based model points to a mark chain. As a result, the model picture from section 4,



corresponds to the following stack, underflow, and marks picture:



In the small, mark-intensive example, the underflow chain appears more elaborate than the stack-based continuation frames that they track. When continuation marks turn out to be dense, the implementation ends up similar to heap-allocated continuation frames. More commonly, continuation marks are sparse relative to continuation frames.

The cost of continuation-mark management is minimized for code that does not use continuation marks. As in the case of heap-allocated frames, the only difference is one pointer slot and a pointer move when creating an underflow record or when handling an underflow, which corresponds to explicit continuation capture or implicit overflow handling for deep recursion.[4]

Furthermore, a continuation frame must be reified to manage marks only when the frame corresponds to a function-call frame—that is, when `with-continuation-mark` is used in tail position of a function body or another another function call. As we discuss in section 7, continuation marks in other positions can be implemented by just pushing and popping the `marks` linked list.

The cost of reifying a continuation for a mark can be further mitigated because the reified continuations are *one-shot* [6]. That is, they are not truly first-class continuations but instead used only to return. Chez Scheme already supports one-shot continuations, but they do not pay off here. Our implementation of continuation marks takes advantage of a new *opportunistic one-shot* variant.

The underflow handler can detect when it is returning to a one-shot continuation where the end of the resumed stack frame matches the former stack base. In that case, it can revert the stack split that was performed when the continuation was reified. Fusing the current stack back with the continuation's stack, which is just a matter of updating the stack-length register, means that no copying is necessary; computation can continue with the saved stack base and restored stack length as the new stack. This handling is *opportunistic* in the sense that a garbage collection may

[4] A winder record for `dynamic-wind` must also have an extra field for marks in the `dynamic-wind` call's continuation. Those marks are restored while running one of the winder thunks.

move the captured stack and current stack so that they're no longer contiguous and available to fuse. The garbage collector promotes each opportunistic one-shot continuation to a full continuation, so the underflow handler will not attempt to fuse stacks when they have been separated.

When `call/cc` captures a continuation, it must also promote any one-shot continuations in the tail of the continuation to full continuations—whether the one-shot continuation was opportunistic or not. This promotion is already implemented for Chez Scheme's existing one-shot support, and opportunistic one-shot continuations get promoted by the same mechanism.

7 Compiler Support

The run-time representation of continuations and marks is only half of the story. The other half is compiler support to take care of conceptual continuation frames that do not correspond to function calls, to avoid some closure creations, and to avoid some indirect function calls.

Without compiler support, the form

```
(with-continuation-mark key val body)
```

could expand to a call of a primitive `call/cm` function

```
(call/cm key val (λ () body))
```

where `call/cm` manipulates the global `stack-base`, `underflows`, and `marks` registers. An advantage of this approach is uniformity: conceptual continuation frames that may have attached marks will always correspond to actual function calls with continuation frames that can be reified. A drawback of this expansion starts with the allocation of a closure for the thunk to wrap *body*, even though the closure is always immediately applied. If *body* is simple enough, the cost of just allocating the closure could dominate the cost of the expression, not to mention the cost of reifying the continuation to pop the continuation mark when *body* returns.

The alternative is to build `with-continuation-mark` into the core language or have the compiler recognize calls to `call/cm` with an immediate `lambda` argument. We take the latter approach for Chez Scheme, except that the primitive functions that are recognized by the compiler handle plain *attachment* values, instead of key–value dictionaries. In other words, arbitrary values take the place of the key–value badges that we have so far shown in pictures. The creation and update of key–value dictionaries, meanwhile, are implemented in the expansion of `with-continuation-mark`.

7.1 Continuation Attachment Primitives

Our modified version of Chez Scheme recognizes four new primitives to handle continuation attachments:[5]

```
(call-setting-continuation-attachment val (λ () body))
```

Installs *val* as the attachment of the current continuation frame, replacing any value that is currently attached to the frame, then evaluates *body* in tail position.

```
(call-getting-continuation-attachment dflt (λ (id) body))
```

Evaluates *body* in tail position, binding *id* to the value attached to the current continuation frame, or *dflt* if no attachment is present on the immediate frame.

```
(call-consuming-continuation-attachment dflt (λ (id) body))
```

Like `call-getting-continuation-attachment` but also removes the value (if any) attached to the current continuation frame.

```
(current-continuation-attachments)
```

Simply returns the content of the `marks` register, since the linked list for "marks" is implemented as a Scheme list.

Using these functions, the expansion of

```
(with-continuation-mark key val body)
```

is

```
(let ([k key] [v val])
  (call-consuming-continuation-attachment
   empty-dictionary
   (λ (d)
    (call-setting-continuation-attachment
     (dictionary-update d k v)
     (λ () body)))))
```

7.2 Attachment Low-Level Optimizations

A compiler pass that recognizes the new primitives runs after high-level optimizations such as inlining, constant propagation, and type specialization. Uses of primitive operations, such as arithmetic, have not yet been inlined, but the attachment-optimization pass can recognize uses of primitives and take advantage of the way that they will be inlined later. Consequently, in an expression like

```
(+ 1 (call-setting-continuation-attachment v
                                            (λ ()
                                              (+ 2 (f)))))
```

the compiler can infer that no attachment already exists on the continuation of `(+ 2 (f))`, and it also knows that `+` does not tail-call any function that might inspect or manipulate continuation attachments,

so the expression can be simplified to

```
(+ 1 (begin (set! marks (cons v marks))
            (let ([r (+ 2 (f))])
              (set! marks (cdr marks))
              r)))
```

In contrast, a function body

```
(call-setting-continuation-attachment v
                                      (λ () (f)))
```

is compiled as

```
(begin (reify-continuation!)
       (set! marks (cons v (underflow-marks underflows)))
       (f))
```

[5] The compiler specifically recognizes uses of the primitives with an immediate `lambda` form. Other uses are treated as regular function references.

because the function may have been called in a continuation that already has an attachment, and because underflow must be used to pop the new attachment when `f` returns. The internal `reify-continuation!` intrinsic checks whether the current frame's return address is the underflow handler, and if not, it allocates a new underflow record and updates `underflows`. The internal `underflow-marks` function accesses the `marks` field of an underflow record.

More generally, the compiler categorizes each use of a `call-...-continuation-attachment` as one of three cases:

In tail position within the enclosing function. Expressions that set an attachment in this position must reify the current continuation so that the attachment is removed on return from the function (via underflow), and then a new value can be pushed onto the `marks` list. Expressions that retrieve an attachment must similarly check for a reified continuation; the current frame has an attachment only if the continuation is reified and the current marks list in the `marks` register differs from the marks list in the current `next-stack` underflow record, in which case the attachment is the first element of the current marks list.

Not in tail position, but with a tail call in the argument *body*. Expressions that set an attachment in this position change the way that each tail call within *body* is implemented. Just after the new frame for the call is set up, a new continuation is reified, and `(rest marks)` is installed in the underflow record. Setting the underflow record's marks field to be `(rest marks)` instead of `marks` communicates to the called function that an attachment is present, and it causes the attachment to be popped when the function returns.

Not in tail position, no tail call in *body*. Expressions in this category can be changed to direct operations on `marks`, because the conceptual continuation frame does not correspond to a function-call frame. Furthermore, the compiler will be able to tell statically whether an attachment is present to be replaced or retrieved.

Besides the transformations implied by the categories, the compiler can detect when attachment calls are nested for the same continuation within the function. For example, a "set" operation following a "consume" operation can safely push to `marks` without a reification check. The compiler specifically detects the "consume"–"set" sequence that `with-continuation-mark` uses to avoid redundant work and make the "set" step as efficient as possible.

7.3 Continuation Mark High-Level Optimizations

High-level optimizations can reduce the cost of continuation marks or even remove them from programs where they turn out to be irrelevant. For example, in

```
(let ([x 5])
  (with-continuation-marks 'key 'val x))
```

evaluating the reference to the variable `x` cannot inspect continuation marks, so there's no reason to actually push a mark for `'key`. Racket compiles this expression to the constant `5`, as expected.

Currently, these high-level optimizations are implemented in the *schemify* pass of Racket on Chez Scheme [15], so the Chez Scheme compiler itself has only minimal support. Adding new optimizations to Chez Scheme's `cp0` pass could benefit Scheme programs (as opposed to Racket programs that are run through the schemify pass). Additions to `cp0` could also benefit Racket programs if `cp0` finds simplifications that schemify missed and that expose continuation-attachment operations, but we have not yet explored that possibility.

7.4 Constraints on Other Optimizations

Continuation marks and their applications require a small refinement to the semantics of Scheme beyond the addition of new operations. In the same way that proper handling of tail calls obliges a compiler and runtime system to *avoid* extending the continuation in some cases, the semantics of continuation marks oblige a compiler and runtime to *extend* the continuation in some cases. For example, the expression

```
(let ([x (work)])
  x)
```

is not equivalent to just `(work)`, because the right-hand size of a `let` form is not in tail position with respect to the `let` form.

Prior to our addition of continuation attachments to Chez Scheme, its `cp0` pass would simplify the above expression to just `(work)`, possibly making a program use less memory or even turning a space-consuming recursion into a constant-space loop. Our modified version of Chez Scheme disables that simplification if it could possibly be observed through continuation-attachment operations. The simplification is still possible in many cases, such as in

```
(+ 1 (let ([x (work)])
       x))
```

where there is no way using attachments to distinguish between a single continuation frame created for the second argument to `+` and that frame plus another one for the right-hand side of `let`. Performance measurements (reported in section 8.2) suggest that the more restricted simplification rarely affects Scheme programs or their performance.

7.5 Representing and Accessing Marks

At the Chez Scheme level, the only operation to retrieve continuation marks is `current-continuation-attachments`, which returns a list of all attachments. To make key-based mark lookup efficient, Racket CS implements a form of path compression by having each attachment in the list potentially hold a key–value dictionary and a cache. When a request for a specific key is satisfied by searching the first N items of the attachment list, the result is stored at position $N/2$ in the attachment list. When a second request finds the result at $N/2$, then it caches the answer again at $N/4$, and so on, until the depth for caching becomes too small to be worthwhile. This caching strategy ensures that `continuation-mark-set-first` works in amortized constant time. A specific attachment uses a representation that makes common cases inexpensive and evolves to support more complex cases: no marks, one mark, multiple marks (using a persistent hash table), and caching (using an additional hash table for the cache).

8 Performance Evaluation

Our performance evaluation for compiler-supported continuation marks has five parts: (1) establishing that continuations in unmodified Chez Scheme perform well; (2) demonstrating that our changes to Chez Scheme have only a small effect on programs that do not use continuation marks; (3) demonstrating that compiler and runtime support provides a significant improvement for continuation-mark operations; (4) demonstrating that compiler support in Chez Scheme helps Racket CS and makes its implementation of continuation marks competitive with the original Racket implementation; and (5) demonstrating that some specific optimizations have a measurable effect on performance.

Benchmark sources are provided as supplementary material, and the sources indicate the exact versions of software used (all latest as of writing). Measurements are performed on five runs with average run times and standard deviations reported. The measurement platform was a 2018 MacBook Pro 2.7 GHz Intel Core i7 running macOS 10.14.6.

8.1 Performance of Continuations

The following table shows results for the traditional `ctak` Scheme benchmark on several implementations:

	average	*stdev*
Pycket	74 ms	±8 ms
Chez Scheme	156 ms	±3 ms
Racket CS	439 ms	±14 ms
CHICKEN	747 ms	±4 ms
Gambit	1646 ms	±9 ms
Racket	19112 ms	±461 ms

These results illustrate that Chez Scheme's continuations perform well compared to other established Scheme implementations. Pycket [4] employs heap-allocated stack frames, which is ideal (only) for continuation-intensive benchmarks like `ctak`, so we count performance on the order of Pycket's performance as good. "Racket CS" is Racket on Chez Scheme, as opposed to the original variant of Racket. Racket CS wraps Chez Scheme's `call/cc` to support delimited continuations, threads, asynchronous break exceptions, and more; creation of a wrapper and the indirection for calling a wrapper accounts for the performance difference relative to Chez Scheme. The non-CS variant of Racket's has very slow continuations, which is among the reasons that Racket CS will replace it.

	average	*stdev*
Pycket native	155 ms	±10 ms
Chez Scheme [K]	202 ms	±5 ms
GHC [DPJS]	273 ms	±6 ms
GHC [K]	325 ms	±12 ms
Multicore OCaml native	410 ms	±2 ms
Chez Scheme [DPJS]	467 ms	±3 ms
Racket CS [K]	569 ms	±6 ms
Racket CS native	600 ms	±14 ms
Racket CS [DPJS]	1113 ms	±11 ms
CHICKEN [K]	1270 ms	±53 ms
Pycket [K]	1547 ms	±71 ms
Gambit [K]	1577 ms	±15 ms
Pycket [DPJS]	5377 ms	±53 ms
Racket [DPJS]	14932 ms	±557 ms
Racket [K]	16374 ms	±234 ms
Koka on Node.js native	17687 ms	±202 ms
Racket native	18526 ms	±436 ms

Fig. 1. Run times for the `triple` delimited-continuation benchmark. "Native" means the language's built-in constructs for delimited control, "[DPJS]" uses the implementation provided by Dybvig et al. [13] for Scheme using `call/cc` and Haskell using monads, and "[K]" uses related implementations by Kisylyov [20].

To provide a rough comparison of Chez Scheme's performance to other language implementations, we show results for the `triple` delimited-continuation benchmark in figure 1. This benchmark finds the 3,234 combinations of three integers between 0 and 200 that add up to 200; it searches the space of possibilities using delimited continuations and two kinds of prompts for two different kinds of choices. All of the implementations explore the space in the same deterministic order. With such different implementations, the results are useful only as an order-of-magnitude suggestion of relative performance, and at that level, the results confirm that Chez Scheme continuations perform well.

8.2 Performance of Modified Chez Scheme

To check how much continuation-attachment support affects Chez Scheme's performance, we re-run the `triple` benchmarks with unmodified and modified Chez Scheme. The "attach" variant in the table below includes continuation-attachment support and constraints on `cp0` to preserve non-tail positions. For completeness, we also check a variant modified in additional ways to support Racket CS [15], which is the "all mods" variant in the table. The table shows run times with the `triple` search space increased to 400:

benchmark	*unmod time*	*attach time*	*all mods time*	*max relstdev*
`collatz-q`	1730 ms	×0.99	×0.99	1%
`cpstak`	1066 ms	×1.03	×1.02	3%
`dderiv`	1317 ms	×1.01	×1.03	4%
`destruct`	936 ms	×0.94	×1.01	3%
`earley`	805 ms	×0.97	×1.04	1%
`fft`	1692 ms	×1.02	×1.04	2%
`lattice`	935 ms	×1.01	×0.92	0%
`maze`	796 ms	×1.01	×1.03	1%
`maze2`	2118 ms	×1.01	×1.03	1%
`nboyer`	597 ms	×0.97	×0.96	2%
`nqueens`	1352 ms	×1.00	×0.96	0%
`nucleic2`	3790 ms	×1.05	×1.06	5%
`peval`	1102 ms	×1.02	×0.96	1%
`sboyer`	941 ms	×1.00	×0.86	0%
`scheme-c`	694 ms	×1.01	×1.06	1%
`sort1`	2011 ms	×1.03	×1.01	2%
Not shown: 22 more with attach within 1 stddev				

Fig. 2. Run times for variants of Chez Scheme on a suite of traditional Scheme benchmarks.

	average	*stdev*
unmodified [K]	1389 ms	±26 ms
attach [K]	1448 ms	±20 ms
all modifications [K]	1509 ms	±58 ms
unmodified [DPJS]	3283 ms	±46 ms
attach [DPJS]	3322 ms	±13 ms
all modifications [DPJS]	3374 ms	±46 ms

The difference in the first two lines of the table shows the cost, for a program that does almost nothing but capture and call continuations, of adding an `attachments` field to a continuation. The additional constraints imposed on `cp0` have no effect on these benchmarks; the output of `cp0` is the same for the "unmodified" and "attach" variants. The output of `cp0` for "CS" is different due to a type-reconstruction pass that replaces some checked operations with unchecked ones, and the small additional slowdown appears to be due to secondary effects of a larger compiler footprint.

To check the effect of modifying Chez Scheme for programs other than continuation-intensive examples, we run the Chez Scheme variants on traditional Scheme benchmarks. The results are shown in figure 2, but only for benchmarks where the difference was greater than a standard deviation; that is, we omit most of the 38 benchmarks because the difference is clearly not significant. Even so, the difference between "unmodified" and "attach" is in the noise. (Type reconstruction in "CS" sometimes pays off.)

8.3 Performance of Continuation Attachments

To measure the benefit of compiler and runtime support for continuation attachments, we compare to an implementation of continuation attachments illustrated in figure 3. This implementation uses `eq?` on continuation values to detect when an attachment should replace an existing attachment. It may also insert extra continuation frames: the `call-setting-continuation-attachment` argument is not always called in tail position, because a pop of the attachments stack must be added before returning from the thunk. However, the thunk is called in tail position if an attachment already exists for the current frame, which means that the number of frames for a program is at most doubled, and a program cannot detect the extra frame using continuation attachments.

```
(define ks '(#f)) ; stack of frames with attachments
(define atts '()) ; stack of attachments

(define (call-setting-continuation-attachment v thunk)
  (call/cc
   (lambda (k)
     (cond [(eq? k (car ks))
            (set! atts (cons v (cdr atts)))
            (thunk)]
           [else
            (let ([r (call/cc
                      (lambda (nested-k)
                        (set! ks (cons nested-k ks))
                        (set! atts (cons v atts))
                        (thunk)))])
              (set! ks (cdr ks))
              (set! atts (cdr atts))
              r)]))))
```

Fig. 3. Imitation of built-in attachment support

benchmark	*builtin time*	*imitate time*	*speedup range*
`base-loop`	918 ms	×1.0	×1.0– ×1.0
`base-callcc-loop`	3603 ms	×1.1	×1.0– ×1.2
`base-deep`	20 ms	×0.9	×0.8– ×1.1
`base-callcc-deep`	648 ms	×1.0	×0.8– ×1.2
`set-loop`	2353 ms	×4.6	×4.4– ×4.8
`get-loop`	1582 ms	×4.5	×4.5– ×4.6
`get-has-loop`	2068 ms	×3.8	×3.7– ×3.8
`get-set-loop`	2819 ms	×5.7	×5.3– ×6.1
`consume-set-loop`	2798 ms	×7.0	×6.1– ×8.2
`set-nontail-notail`	175 ms	×22.3	×21.0–×23.7
`set-tail-notail`	916 ms	×4.2	×3.8– ×4.7
`set-nontail-tail`	888 ms	×4.3	×3.9– ×4.7
`loop-arg-call`	7023 ms	×6.1	×6.0– ×6.1
`loop-arg-prim`	3422 ms	×12.5	×12.3–×12.7

Fig. 4. Performance of built-in support for continuation marks and speedups of average runs compared to the imitation strategy of figure 3. The "speedup range" column is derived from standard deviations: the low end is the ratio of built-in plus standard deviation to imitation minus standard deviation, and the high end is the ratio of minus to plus.

benchmark	*Racket CS time*	*Racket time*	*speedup range*
`base-loop`	929 ms	×1.4	×1.3– ×1.5
`base-deep`	738 ms	×5.8	×4.4– ×7.4
`base-arg-call-loop`	2326 ms	×2.3	×2.3– ×2.4
`set-loop`	6349 ms	×0.6	×0.6– ×0.7
`set-nontail-prim`	509 ms	×5.7	×4.9– ×6.7
`set-tail-notail`	1503 ms	×1.3	×1.2– ×1.5
`set-nontail-tail`	1461 ms	×1.3	×1.2– ×1.5
`set-arg-call-loop`	8658 ms	×0.9	×0.9– ×1.0
`set-arg-prim-loop`	5360 ms	×1.0	×0.9– ×1.1
`first-none-loop`	1710 ms	×1.1	×1.1– ×1.1
`first-some-loop`	1009 ms	×0.6	×0.6– ×0.6
`first-deep-loop`	5067 ms	×1.1	×1.0– ×1.1
`immed-none-loop`	5515 ms	×1.1	×1.1– ×1.2
`immed-some-loop`	5723 ms	×1.2	×1.2– ×1.2

Fig. 5. Performance on continuation-mark benchmarks for Racket CS versus the old implementation of Racket. Speedup ranges are based on standard deviations as in figure 4.

Figure 4 summarizes the performance improvements from built-in compiler and runtime support on benchmarks. The initial rows with names starting "base" do not use continuation attachments and provide a baseline for other loops of 1000M iterations and nested calls 1M deep repeated 10 times, with and without continuation capture. The next "loop" rows involve a get, set, consume, or combination of those operations wrapping the recursive call. The next non-"loop" rows perform deep recursion where each frame gets an attachment in varying tail and non-tail positions. The final "loop" rows use a set operation around the argument of the recursive call, sometimes a primitive and sometimes a call to a non-inlined function.

The results show substantial improvements across the board for compiler and runtime support. The benefits derive from avoiding an extra continuation frame, avoiding closure allocations, and avoiding reification of the continuation around primitive operations.

8.4 Performance of Continuation Marks

Finally, we compare the overall performance of continuation marks in Racket CS to the performance of the old Racket implementation. Although the performance of first-class *continuations* in old Racket is poor (see figure 1), the performance of setting and getting continuation marks is relatively good (as can be inferred from figure 5).

Figure 5 shows results of running benchmarks on Racket CS and speedups compared to the old version of Racket. Loop runs use 1000M iterations while non-loop runs use 1M recursions repeated 10 times. The `base-loop` result shows that Racket CS and Racket start with similar performance for plain loops, but `base-deep` and `base-arg-call-loop` and show the improved baseline performance of non-tail calls in Racket CS (derived from Chez Scheme's better performance). The old Racket implementation outperforms Racket CS in some specific cases, because it uses a stack representation for the mark stack instead of a heap-allocated linked list, but that choice creates complexity and costs when capturing continuations.

End-to-end performance for most Racket programs is affected by continuation-mark performance, but the difference between built-in and imitated continuation attachments is often only 1%. Racket's contract library is significantly affected, so applications that rely heavily on contract checking also depend more heavily on continuation-attachment performance. The following table shows the performance of calling an imported, non-inlined identity function 20M times with and without checking a `(-> integer? integer?)` contract:

contract mode	*buitin time*	*imitate time*	*max relstdev*
`unchecked`	42 ms	×1.00	2%
`checked`	428 ms	×3.42	1%

Improvements to contact checking and dynamic binding affect some Racket applications measurably. The following table shows the end-to-end performance of useful Racket programs on realistic inputs, where an significant dependence on contract checking or dynamic binding (usually for configuration) show the benefit of faster continuation marks:

application	*builtin time*	*imitate time*	*max relstdev*
`ActivityLog import`	7189 ms	×1.11	2%
`Xsmith cish`	5128 ms	×1.09	3%
`Megaparsack JSON`	2287 ms	×1.24	1%
`Markdown Reference`	4777 ms	×1.16	2%
`OL1V3R gauss.175.smt2`	1816 ms	×1.10	2%

8.5 Effect of Optimizations

Figure 6 shows the results of the continuation mark, contract, and application benchmarks using variants of Racket CS with different optimizations disabled:

- The *no 1cc* variant disables optimistic one-shot continuations and instead always uses multi-shot continuations. Optimistic one-shot continuations affect the `set-arg-call-loop` microbenchmark with a speedup of about ×1.5. That benchmark reflects a common behavior of contracts, and optimistic one-shots also speed up the contract-checking benchmark by about ×1.4.
- The *no opt* variant disables the compiler's recognition and specialization of continuation-attachment operations. Compiler optimizations affect many microbenchmarks with improvements of ×1.2 to ×3.5. The unaffected benchmarks are mostly the ones about continuation mark access (such as `first-none-loop`), as they should be. Optimizations speed up the contract-checking benchmark by about ×2.

benchmark	*Racket CS time*	*no 1cc*	*no opt*	*no prim*	*max stdev*
`base-deep`	738 ms	×1.04	×0.97	×1.00	4%
`set-loop`	6349 ms	×1.02	×1.97	×0.89	4%
`set-nontail-prim`	509 ms	×1.02	×3.51	×1.10	6%
`set-tail-notail`	1503 ms	×0.94	×1.09	×0.98	7%
`set-nontail-tail`	1461 ms	×0.92	×1.06	×1.00	9%
`set-arg-call-loop`	8658 ms	×1.48	×1.30	×1.00	3%
`set-arg-prim-loop`	5360 ms	×1.04	×2.03	×1.60	9%
`first-none-loop`	1710 ms	×1.05	×1.02	×0.98	0%
`first-some-loop`	1009 ms	×1.05	×1.01	×1.04	1%
`first-deep-loop`	5067 ms	×1.04	×1.00	×0.96	2%
`immed-none-loop`	5515 ms	×1.10	×1.45	×0.95	6%
`immed-some-loop`	5723 ms	×1.10	×1.22	×0.98	2%

contract mode	*Racket CS time*	*no 1cc*	*no opt*	*no prim*	*max stdev*
`unchecked`	42 ms	×0.98	×1.05	×1.02	4%
`checked`	428 ms	×1.38	×1.98	×1.41	0%

application	*Racket CS time*	*no 1cc*	*no opt*	*no prim*	*max stdev*
`ActivityLog`	7189 ms	×1.07	×1.04	×1.03	2%
`Xsmith`	5128 ms	×1.04	×1.01	×0.99	2%
`Megaparsack`	2287 ms	×1.05	×1.07	×1.04	2%
`Markdown`	4777 ms	×1.05	×1.03	×1.01	3%
`OL1V3R`	1816 ms	×1.04	×1.04	×1.00	3%

Fig. 6. Effect of optimizations on benchmark suites.

- The *no prim* variant disables only the part of the compiler's optimization to recognize primitives that will never affect the current continuation's attachments. Just the compiler optimizations for non-tail primitive applications affects microbenchmarks to a lesser degree, but still sometimes ×1.6. This optimization speeds up the contract-checking benchmark by ×1.4.

The effect of individual optimizations on the example applications is small, but still sometimes large enough to be measurable at around ×1.05.

9 Related Work

Continuation marks have been a part of Racket from its early days [8], but delimited control was added later [17]. Kiselyov et al. [19] provided a semantics of dynamic binding with delimited control earlier. They offer some implementation strategies, but they do not discuss potential compiler support or investigate performance in depth.

The implementation of dynamic scope was of particular interest for early dialects of Lisp. The attachments list in our implementation is reminiscent of the "deep" strategy for dynamic binding, but without explicitly threading an environment through all function calls. An explicit threading approach was also part of an early attempt to support a dynamic-binding mechanism that does not grow the control stack [11], but that solution simply shifted stack growth from the main control stack to a separate dynamic-binding stack, so that tail recursion involving dynamic binding still caused unbounded memory growth. In contrast, such memory growth in our mechanism is limited by the number of dynamically bound variables; i.e., it is proportional to the number of unique keys attached a frame.

The "shallow" strategy for dynamic scope [3] is sometimes implemented by setting and restoring the value of a global or thread-local variable and using constructs like `dynamic-wind` to cooperate with escapes and continuation capture—including in Chez Scheme [12 §1.3]. That strategy puts the body of each dynamic binding in non-tail position and imposes a cost on continuation jumps. On the other hand, references to dynamic variables implemented via shallow binding are less expensive; the best choice of dynamic binding strategy depends on the relative expected frequencies of binding operations, control operations, and references.

Implicit parameters [21] solve a problem similar to dynamic scope. By leveraging the type system, they automate the addition of optional arguments through layers of calls, making those parameters easier to provide and propagate. The Reader monad serves a similar purpose. Continuation marks communicate through layers more pervasively and without threading an argument or dictionary through a computation. At the same time, continuations make sense only with eager evaluation, where evaluation has a well-defined notion of dynamic extent that is reflected in the continuation.

10 Conclusion

The Racket ecosystem of languages depends on an efficient implementation of the core constructs that enable language construction. Continuation marks have proven to be an important component of Racket's language-construction toolbox, where they serve as a universal building block that permits library-level implementation of extent-sensitive features: dynamic binding, exceptions, profiling, and more. We have shown how continuation marks can be implemented as part of an efficient, stack-based implementation of continuations. Compiler and runtime support for continuation marks provide a significant reduction in end-to-end run times (10-25%) for practical Racket applications.

References

[1] Leif Andersen, Vincent St-Amour, Jan Vitek, and Matthias Felleisen. Feature-Specific Profiling. *Transactions on Programming Languages and Systems* 41(1), 2019.

[2] Andrew W. Appel and Zhong Shao. Empirical and Analytic Study of Stack Versus Heap Cost for Languages with Closures. *Journal of Functional Programming* 6(1), 1996.

[3] Henry G. Baker Jr. Shallow Binding in Lisp 1.5. *Proceedings of the ACM* 27(7), 1978.

[4] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: a Tracing JIT for a Functional Language. In *Proc. International Conference on Functional Programming*, 2015.

[5] Jonathan Brachthauser and Daan Leijen. Programming with Implicit Values, Functions, and Control (or, Implicit Functions: Dynamic Binding with Lexical Scoping). Microsoft, MSR-TR-2019-7, 2019.

[6] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing Control in the Presence of One-Shot Continuations. In *Proc. Programming Language Design and Implementation*, 1996.

[7] John Clements and Matthias Felleisen. A Tail-Recursive Machine with Stack Inspection. *Transactions on Programming Languages and Systems* 26(6), pp. 1029–1052, 2004.

[8] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an Algebraic Stepper. In *Proc. European Symposium on Programming*, 2001.

[9] Daniel Hillerström and Sam Lindley. Liberating Effects with Rows and Handlers. In *Proc. Workshop on Type-Driven Development*, 2016.

[10] Daniel Hillerström and Sam Lindley. Shallow Effect Handlers. In *Proc. Asian Symposium on Programming Languages and Systems*, 2018.

[11] Bruce F. Duba, Matthias Felleisen, and Daniel P. Friedman. Dynamic Identifiers can be Neat. Indiana University, 220, 1987.

[12] K. Kent Dybvig. Chez Scheme Version 9 User's Guide. 2019. https://cisco.github.io/ChezScheme/csug9.5/

[13] R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. A Monadic Framework for Delimited Continuations. *Journal of Functional Programming* 17(6), pp. 687–730, 2007.

[14] Dan Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. Collapsible Contracts: Fixing a Pathology of Gradual Typing. In *Proc. Object-Oriented Programming, Systems, Languages and Applications*, 2018.

[15] Matthew Flatt, Caner Derici, R. Kent Dybvig, Andrew W. Keep, Gustavo E. Massaccesi, Sarah Spall, Sam Tobin-Hochstadt, and Jon Zeppieri. Rebuilding Racket on Chez Scheme (Experience Report). In *Proc. International Conference on Functional Programming*, 2019.

[16] Matthew Flatt and PLT. The Racket Reference. 2010. https://docs.racket-lang.org/reference/

[17] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding Delimited and Composable Control to a Production Programming Enviornment. In *Proc. International Conference on Functional Programming*, 2007.

[18] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing Control in the Presence of First-Class Continuations. In *Proc. Programming Language Design and Implementation*, 1990.

[19] Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited Dynamic Binding. In *Proc. International Conference on Functional Programming*, 2006.

[20] Oleg Kisylyov. Continuations and Delimited Control. 2019. http://okmij.org/ftp/continuations/

[21] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit Parameters: Dynamic Scoping with Static Types. In *Proc. Principles of Programming Languages*, 2000.

[22] Xiangqi Li and Matthew Flatt. Debugging with Domain-Specific Events via Macros. In *Proc. Software Language Engineering*, 2017.

[23] Jay McCarthy. Automatically RESTful Web Applications Or, Marking Modular Serializable Continuations. In *Proc. International Conference on Functional Programming*, 2009.

[24] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. Extensible Access Control with Authorization Contracts. In *Proc. Object-Oriented Programming, Systems, Languages and Applications*, 2016.

[25] Gordon D. Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science* 9(4), 2013.

[26] Bogdan Popa. Generators from Scratch. 2019. https://defn.io/2019/09/05/racket-generators/