Trampolined Style

Steven E. Ganz[*] Indiana University
Daniel P. Friedman[†] Indiana University
Mitchell Wand[‡] Northeastern University

Abstract

A trampolined program is organized as a single loop in which computations are scheduled and their execution allowed to proceed in discrete steps. Writing programs in trampolined style supports primitives for multithreading without language support for continuations. Various forms of trampolining allow for different degrees of interaction between threads. We present two architectures based on an only mildly intrusive trampolined style. Concurrency can be supported at multiple levels of granularity by performing the trampolining transformation multiple times.

1 Introduction

Trampolined style is a way of writing programs such that a single "scheduler" loop, called `trampoline`, manages all transfers of control.[1] Computations are executed in discrete steps. Whenever a computation performs a unit of work, the remaining work is returned to the scheduler.

Trampolining has been applied to interpreters for reflection [2] as well as for process abstractions [6]. A form of trampolining has been applied to arbitrary programs by Tarditi, *et al.*, for proper C implementations of ML tail calls [23]. In the implementation of the Icsla language, Queinnec and De Roure have applied a transformation related to trampolining to arbitrary programs, allowing the multiprocessing primitives to be defined as simple procedures or macros [20].

We present trampolined style through two trampolining architectures. This categorization is by no means to be considered exhaustive or fundamental. Rather, our choice of architectures is a sampling meant to demonstrate the amount of variation possible. In general, the style involves delaying tail calls so that they take place within a loop. The first architecture allows for stepping by having each computation yield a thread after each unit of work is performed. We present three variants of this architecture, for a one-, two- or multi-thread system. Its single-thread variant is used to demonstrate sequential composition, breakpoints and engines. The second architecture allows for dynamic thread creation and termination by having each computation yield a list of threads to be added to the thread queue at each step.

Trampolined style is parameterized over a type constructor T and the definitions of three procedures: `bounce`, `return`, and a scheduler. The parameter to T corresponds to the type of the result of a computation, so a thread of type $\mathsf{T}(\alpha)$ is an intermediate state of a computation returning a value of type α. The type T is defined such that $\mathsf{T}(\alpha) = \alpha + (\mathsf{unit} \to \mathsf{T}(\alpha))$. The `return` and `bounce` procedures then correspond to the injections ι_l and ι_r, respectively.

In this introduction we present several examples of growing complexity that illustrate our style of trampolining. They rely on two very familiar programs: accumulator-style factorial and determining membership in a list of numbers. In these example programs, all tail calls are wrapped in `(bounce (lambda () ...))` and all values in tail position are wrapped in `(return ...)`. Therefore the output type of both programs is $\mathsf{T}(\alpha)$.

---

[*] This work was supported in part by the National Science Foundation under grant CDA-9312614. Author's address: Department of Computer Science, Indiana University, Bloomington, Indiana 47405. `sganz@cs.indiana.edu`.

[†] This work was supported in part by the National Science Foundation under grant CCR-9633109. Author's address: Department of Computer Science, Indiana University, Bloomington, Indiana 47405. `dfried@cs.indiana.edu`.

[‡] This work was supported in part by the National Science Foundation under grants CCR-9629801 and CCR-9804115. Author's address: College of Computer Science, Northeastern University, Boston, Massachusetts 02115. `wand@ccs.neu.edu`.

[1] Compare this to the Kleene Normal Form Theorem [14], which asserts that any computable function over the natural numbers can be represented with a single use of minimization and otherwise only primitive-recursive operations.

⟨*Simple Trampolined Procedures*⟩≡
```
> (define fact-acc
    (lambda (n acc)
      (if (zero? n)
        (return acc)
        (bounce
          (lambda ()
            (fact-acc
              (sub1 n)
              (* acc n)))))))
> (define mem?
    (lambda (n ls)
      (cond
        [(null? ls) (return #f)]
        [(= (car ls) n) (return #t)]
        [else
          (bounce
            (lambda ()
              (mem? n (cdr ls))))])))
```
If we define `return` to be identity and `bounce` to be `(lambda (thunk) (thunk))`, the programs work as expected.

1.1 Simple Trampolining

We first present a single-thread architecture. Our initial definition of `bounce` packages the remaining computation so that it can be resumed by a scheduler. The `return`ed value is packaged to instruct the scheduler to terminate.

We assume a simple record facility. The record types `done` and `doing` correspond to the two variants of threads. The record type `done` represents a complete computation and holds a result value of any type. The record type `doing` represents an incomplete computation and holds a thunk that performs the remaining work.

⟨*Record Definitions*⟩≡
```
(define-record done (value))
(define-record doing (thunk))
```
The procedure `return` creates a `done` thread.

⟨*return Definition*⟩≡
```
(define return
  (lambda (value)
    (make-done value)))
```
and the procedure `bounce` creates a `doing` thread. Its argument is always a delayed computation, modeled by a procedure of no arguments.

⟨*bounce Definition*⟩≡
```
(define bounce
  (lambda (thunk)
    (make-doing thunk)))
```
If a computation has completed, the scheduler, called `pogo-stick`, terminates; otherwise it resumes the computation. Therefore, its type is $\mathsf{T}(\alpha) \rightarrow \alpha$.

⟨*Scheduler for one thread*⟩≡
```
(define pogo-stick
  (lambda (thread)
    (record-case thread
      [done (value) value]
      [doing (thunk) (pogo-stick (thunk))])))
```
The first example calculates the factorial of a number in trampolined style:

⟨*Single-computation Example*⟩≡
```
> (pogo-stick (fact-acc 5 1))
120
```
This computation builds five `doing` records and, as we might expect, one `done` record. We discover further uses of `pogo-stick` in Section 3.

1.2 Interleaved Trampolining

We can arrange for two computations to be interleaved by using `seesaw` instead of `pogo-stick`. They differ in that `seesaw` takes another computation as an additional argument. Execution of the two computations alternates, and `seesaw` terminates when either of the computations does. Thus, its type is $\mathsf{T}(\alpha) \times \mathsf{T}(\alpha) \rightarrow \alpha$.[2]

⟨*Scheduler for two threads*⟩≡
```
(define seesaw
  (lambda (down-thread up-thread)
    (record-case down-thread
      [done (down-value) down-value]
      [doing (down-thunk)
       (seesaw up-thread (down-thunk))])))
```
We demonstrate the use of `seesaw` by calculating the factorial of a number together with a list membership calculation:

⟨*Double-computation Example*⟩≡
```
> (seesaw
    (fact-acc -1 1)
    (mem? 120 '(100 110 120 130)))
#t
```
Although the `fact-acc` computation does not terminate, the `mem?` computation does, which causes `seesaw` to terminate as well. This is made possible by the replacement of the separate loops in `fact-acc` and `mem?` by a single loop in `seesaw`. The `mem?` computation builds two `doing` records and a `done` record. The `fact-acc` computation builds four `doing` records: one before the trampoline is entered, two before the `mem?` computation is completed, and one more before the `done` record is processed by `seesaw`.

A trampoline is a generalization of a seesaw that allows any number of computations to be interleaved. Our first trampoline is the natural extension of `seesaw` to more than two threads. By our choice of argument order for the call to `append`, we stipulate that the thread list behaves like a round-robin queue. Its type is $(\mathsf{list}\ \mathsf{T}(\alpha)) \rightarrow \alpha$.[3]

---

[2] We do not distinguish the types of individual computations, but assume that α includes their union. If we were to distinguish the types of the computations running on `seesaw`, its type would be $(\mathsf{T}(\alpha) \times \mathsf{T}(\beta)) \rightarrow (\alpha + \beta)$. We would then need to have `seesaw` toggle a boolean value with each iteration to determine which thread to process. This would lead us to identify the computation that returned the result.

[3] As with `seesaw`, we can distinguish the types of the computations, giving a type for `trampoline` of $\Pi_i[\mathsf{T}(\alpha_i)] \rightarrow \Sigma_i[\alpha_i]$. Then replace the toggling of a boolean with the incrementing of an integer modulo the number of threads.

$\langle$*trampoline for Multiple Computations*$\rangle\equiv$

```
(define trampoline
  (lambda (thread-queue)
    (record-case (car thread-queue)
      [done (value) value]
      [doing (thunk)
        (trampoline
          (append
            (cdr thread-queue)
            (list (thunk))))])))
```

We test `trampoline` by running two endless factorial computations together with a single list membership computation.

$\langle$*Multi-computation Example*$\rangle\equiv$

```
> (trampoline
    (list
      (fact-acc -1 1)
      (fact-acc -1 1)
      (mem? 120 '(100 110 120 130))))
#t
```

As with the previous example, the `mem?` computation builds two `doing` records and a `done` record, and each `fact-acc` computation builds four `doing` records.

In the remainder of this paper, we present the trampolining transformation in more detail. We define trampolining architectures of increasing complexity and then extend the style in accordance with particular idioms, observing how new control behavior related to multithreading becomes possible. Throughout, we attempt to make this style of programming as natural as possible. Section 2 more formally introduces the trampolining transformation. The main part of the paper follows in Sections 3 and 4 where we describe two trampolining architectures and the operations that can be supported by extending the style in each case. We then show in Section 5 how to vary the granularity of concurrency with multiple iterations of trampolining. In Section 6 we show another way in which our methodology has been used: continuation-passing style (CPS) and the `call/cc` operator. We take a closer look at the history of trampolining in Section 7. Section 8 concludes.

2 The Trampolining Transformation

When is trampolining possible? We have seen that we can trampoline `fact-acc` and `mem?`. What do their definitions have in common? A subexpression is in *tail position* if and only if it has no control context within any immediately enclosing lambda expression, or has no control context at all and is not enclosed in a lambda expression. A program is in *tail form* if and only if all non-primitive applications are in tail position. Any program in tail form can be rewritten in trampolined style. Thus, we can trampoline any program by first rewriting it in CPS [9, 24]. When a CPS program with final continuation `(lambda (`x`)` x`)` is trampolined, the final continuation is rewritten as `(lambda (`x`) (return` x`))`. This is equivalent to passing `return` as the final continuation. This is the *only* occurrence of return in such programs. But we can also rewrite many other programs in trampolined style, as demonstrated by our earlier examples. This is an advantage of our approach over others who have relied on variants of CPS [20]. Wadler's monadic transformation [25], based on Moggi's [17], also produces programs in tail form. Any such programs are amenable to our transformation.

$$
\begin{array}{rcl}
E & ::= & c \mid x \mid (\texttt{lambda}\ (x)\ E) \mid (\texttt{set!}\ x\ S) \\
 & \mid & (c\ S) \mid (\texttt{if}\ S\ E\ E) \mid (\texttt{begin}\ S\ E) \\
 & \mid & (S\ S) \\
S & ::= & c \mid x \mid (\texttt{lambda}\ (x)\ E) \mid (\texttt{set!}\ x\ S) \\
 & \mid & (c\ S) \mid (\texttt{if}\ S\ S\ S) \mid (\texttt{begin}\ S\ S)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T}[c] & = & (\texttt{return}\ c) \\
\mathcal{T}[x] & = & (\texttt{return}\ x) \\
\mathcal{T}[(\texttt{lambda}\ (x)\ E)] & = & (\texttt{return}\ (\texttt{lambda}\ (x)\ \mathcal{T}[E])) \\
\mathcal{T}[(\texttt{set!}\ x\ S)] & = & (\texttt{return}\ (\texttt{set!}\ x\ t[S])) \\
\mathcal{T}[(c\ S)] & = & (\texttt{return}\ (c\ t[S])) \\
\mathcal{T}[(\texttt{if}\ S\ E_1\ E_2)] & = & (\texttt{if}\ t[S]\ \mathcal{T}[E_1]\ \mathcal{T}[E_2]) \\
\mathcal{T}[(\texttt{begin}\ S\ E)] & = & (\texttt{begin}\ t[S]\ \mathcal{T}[E]) \\
\mathcal{T}[(S_1\ S_2)] & = & (\texttt{bounce} \\
 & & \quad (\texttt{lambda}() \\
 & & \quad\quad (t[S_1]\ t[S_2])))
\end{array}
$$

$$
\begin{array}{rcl}
t[c] & = & c \\
t[x] & = & x \\
t[(\texttt{lambda}\ (x)\ E)] & = & (\texttt{lambda}\ (x)\ \mathcal{T}[E]) \\
t[(\texttt{set!}\ x\ S)] & = & (\texttt{set!}\ x\ t[S]) \\
t[(c\ S)] & = & (c\ t[S]) \\
t[(\texttt{if}\ S_1\ S_2\ S_3)] & = & (\texttt{if}\ t[S_1]\ t[S_2]\ t[S_3]) \\
t[(\texttt{begin}\ S_1\ S_2)] & = & (\texttt{begin}\ t[S_1]\ t[S_2])
\end{array}
$$

Figure 1: Trampolining Transformation

We demonstrate how to transform a tail-form expression into trampolined style. Throughout the original program, all tail calls of user-defined procedures are wrapped in `(bounce (lambda () ...))`, and all simple values in tail position are wrapped in `(return ...)`. Figure 1 is a more formal presentation of the trampolining transformation. $\mathcal{T}$ takes a tail-form expression E given by the grammar; the auxiliary transformation t is used for simple expressions S. We use (possibly subscripted) E, S, x, and c as metavariables for tail-position expressions, simple expressions, variables, and constants, respectively. The program resulting from applying $\mathcal{T}$ must be wrapped in `(`*scheduler* `...)` to be properly executed.

The transformation t need not handle applications because programs are presumed to be in tail form. The `call/cc` operator can be treated as a primitive. We use $\mathcal{T}$ over derived forms (such as `let`, `letrec`, and `cond`), although those cases are not specified here.

Primitives accepting higher-order arguments (such as `map`) deserve special mention. Generally, such primitives can easily be implemented in the source language. The natural implementations are not tail recursive. One option is to convert the natural implementations to tail form and then trampoline them. Calls to the primitive are then just treated as tail calls. Another option is to only use the primitives with "safe" higher-order arguments that are guaranteed to terminate. In this case, the arguments should not be trampolined, and the primitive call should be considered as not trampolined in the sense described above. Such primitive calls must not be wrapped in `return` forms (and should not be in tail position).

We need not trampoline an entire program. It may be appropriate to trampoline only a part. If a `lambda` expression is not trampolined, then any applications in which it is the operator must not be wrapped in a `bounce` form (and should not be in tail position). Any such applications are treated as atomic. Other, nested, `lambda` abstractions may still be trampolined.

Not all of the occurrences of `(bouce (lambda () ...))` dictated by the transformation are necessary. Of all tail calls involved in a possible loop, at least one must be an argument to `bounce`. Otherwise, we might spark an unending chain of invocations of a tail-recursive procedure, without control ever being released. No other instances of `bounce` are necessary.

As an aside, we consider what calls to bounce might be necessary to trampoline a program in CPS. In particular, could we get by with just annotating the calls to the continuations? The answer is "no". In many CPS computations (such as `(fact-cps -1 (lambda (x) x))`), the first continuation is not invoked until the calls to the recursive procedure bottom out. The presence of another `bounce` call in the continuation argument is of no help in avoiding the starvation of other threads.

The transformation rules above are incomplete because they leave `bounce` and `return` unspecified. The various definitions of `bounce` and `return` can each be interpreted as equations that complete the transformation rules above. Although we have chosen to define `bounce` and `return` as procedures in implementing the system, a macro-based implementation would be more consistent with that interpretation. Such an implementation would provide simpler transformed programs and would not require the `(lambda () ...)` as part of the user's interface. This applies not only to `bounce` and `return`, but to the extensions to trampolined style presented below.

3 Stepping

In this section, we apply the `pogo-stick` architecture to several problems. Each requires an extension to the style and thus the transformation, allowing for expressions that evaluate to threads in non-tail position. We first demonstrate a tool, less drastic than the CPS transformation, for extending the domain of the trampolining transformation. Next we provide facilities for breakpoints and engines.

3.1 Sequential Composition

The next example calculates the presence of the factorial of a number in a list. We cannot organize it in the obvious way, because the call to `fact-acc` would not be in tail position. That call returns a thread record that can be passed to `pogo-stick` to complete the `fact-acc` computation, returning a number that can be passed to `mem?`. The example runs as follows:

$\langle$*Double pogo-stick Example*$\rangle\equiv$
```
> (pogo-stick
    (mem?
      (pogo-stick (fact-acc 5 1))
      '(100 110 120 130)))
#t
```

For `pogo-stick`, this is fine, but we have already seen generalizations that allow for multiple concurrent threads. This naive strategy would lead to starvation of other computations when the first computation in this sequence runs indefinitely. We see in Section 5 below how this can be handled more safely. Alternatively, we define a sequential composition operator over a procedure written in trampolined style and a thread. The procedure `sequence` takes a procedure of one argument and a thread. It completes the execution of the thread and feeds its result to the procedure. Its type is $((\alpha \to \mathsf{T}(\alpha)) \times \mathsf{T}(\alpha)) \to \mathsf{T}(\alpha)$.

$\langle$*sequence Definition*$\rangle\equiv$
```
(define sequence
  (lambda (f thread)
    (record-case thread
      [done (value)
        (f value)]
      [doing (thunk)
        (bounce
          (lambda ()
            (sequence f (thunk))))])))
```

The second argument to `sequence` is a trampolined expression that yields a thread. Until its computation terminates, each intermediate thread record is dropped to the scheduler as part of another call to `sequence` to ensure the execution of `f`. Upon termination, the value of the computation is passed to `f` and `sequence` is no longer involved.

For generality, we might want the second argument to be a single-parameter procedure, and return a single-parameter procedure. The procedure `seq-comp` satisfies that goal. We can use `sequence` to implement `seq-comp`. The type of `seq-comp` is $((\alpha \to \mathsf{T}(\alpha)) \times (\alpha \to \mathsf{T}(\alpha))) \to (\alpha \to \mathsf{T}(\alpha))$.

$\langle$*seq-comp Definition*$\rangle\equiv$
```
(define seq-comp
  (lambda (f g)
    (lambda (x)
      (sequence f (g x)))))
```

This composition procedure is defined over procedures of one argument, so we curry `fact-acc` and `mem?` before proceeding with our example.

⟨*Curried Trampolined Procedures*⟩≡

```
> (define fact-acc-curry
    (lambda (acc)
      (lambda (n)
        (if (zero? n)
          (return acc)
          (bounce
            (lambda ()
              ((fact-acc-curry
                 (* acc n))
               (sub1 n))))))))

> (define mem?-curry
    (lambda (ls)
      (lambda (n)
        (cond
          ((null? ls) (return #f))
          ((= (car ls) n) (return #t))
          (else
            (bounce
              (lambda ()
                ((mem?-curry (cdr ls))
                 n))))))))
```

Significantly, it is not necessary that these procedures be fully trampolined. Each returns a lambda expression directly, not through `return`. This is acceptable, because we can guarantee that all applications of the inner procedure occur in non-tail position (and are not assumed to be threads). To avoid starvation of other threads, this should not be attempted unless one can also guarantee termination of the non-tail call. Now, the example.

⟨*Simply Trampolined Test*⟩≡

```
> (pogo-stick
    ((seq-comp
       (mem?-curry '(100 110 120 130))
       (fact-acc-curry 1))
     5))
>
#t
```

3.2 Breakpoints

We next show how to extend our protocol to enable the computation to be temporarily halted. We introduce `break` as another interface procedure that intercepts a thread (in tail position) and requests authorization from the console via `resume` before returning it to the scheduler. Alternatively, the user can `return` from the computation at arbitrary breakpoints with a ground value. We assume `syncase`, a simple pattern-matching facility. The type of `break` is $\mathsf{T}(\alpha) \rightarrow \mathsf{T}(\alpha)$.

⟨*Stepping*⟩≡

```
(define break
  (lambda (thread)
    (letrec ([loop
               (lambda ()
                 (printf "% ")
                 (syncase (read)
                   ['(resume)
                     thread]
                   [`(return ,exp)
                     (return exp)]
                   [else
                     (bounce loop)]))])
      (loop))))
```

We now run `fact-acc/break`, a slight variant of `fact-acc`, where we wrap `(break ...)` around the single occurrence of `bounce`.

⟨*Stepping Example*⟩≡

```
> (pogo-stick (fact-acc/break 3 1))
% (resume)
% (resume)
% (resume)
6
```

A more useful breakpoint facility would allow arbitrary nontrampolined expressions to be evaluated while the computation is halted. This could be supported in a language with first-class environments [3]. For this example, we assume `eval-tramp`, a tail-form interpreter that has been trampolined. There is something rather subtle in the uses of `sequence` in this example. We compose an evaluation with its continuation, although the evaluation only yields a thread, perhaps with the evaluation barely begun.

⟨*Enhanced Stepper*⟩≡

```
(define break/env
  (lambda (thread env)
    (letrec
      ([loop
         (lambda ()
           (printf "% ")
           (syncase (read)
             ['(resume)
               thread]
             [`(return ,ret-exp)
               (sequence
                 return
                 (eval-tramp ret-exp env))]
             [input
               (sequence
                 (lambda (v)
                   (printf "~s~n" v)
                   (bounce loop))
                 (eval-tramp input env))]))])
      (loop))))
```

3.3 Engines

Our `pogo-stick` scheduler continues a computation until it has completed. We can modify it to contain a bound, so that a computation is only performed for a fixed number of steps. Such a bounded `pogo-stick` would be similar to an engine [11]. Engines are a mechanism for regulating the progress of a computation by feeding it ticks, much as a car is fed gasoline. Our `make-engine` is a simplification that captures the essence of engines. It takes a thread and returns an engine. It returns the engine directly, and is thus assumed to be used in non-tail position. An engine is invoked by giving it a number of ticks. If the computation completes within that number of ticks, the result is a `done` thread. If the number of ticks is exhausted, the result is a `doing` thread. (These threads must be interpreted by the caller of the engine.) Otherwise, a new engine is created with the remaining computation and passed one fewer ticks. The type of `make-engine` is: $\mathsf{T}(\alpha) \rightarrow \mathsf{Int} \rightarrow \mathsf{T}(\alpha)$.

$\langle$*Engines*$\rangle\equiv$

```
(define make-engine
  (lambda (thread)
    (lambda (ticks)
      (record-case thread
        [done (value) thread]
        [doing (thunk)
          (if (zero? ticks)
            thread
            ((make-engine (thunk))
             (sub1 ticks)))]))))
```

We use an untrampolined `interactive-dotter` program that repeatedly invokes `make-engine` to obtain a scheduler. Each engine is created using a thread of the trampolined program `dot`, which prints an infinite sequence of dots. These engines are invoked with a number of ticks requested from the user console.

$\langle$*Engines Example*$\rangle\equiv$

```
> (define interactive-dotter
    (lambda (thread)
      (printf "~n>> ")
      (let ([input (read)])
        (if (not (zero? input))
          (interactive-dotter
            ((make-engine thread)
             input))))))
> (define dot
    (lambda ()
      (printf ".")
      (bounce dot)))
> (interactive-dotter (bounce dot))

>> 8
........
>> 5
.....
>> 0
```

4 Dynamic Thread Creation

The next form of trampolining extends the technique above by having each tail-position expression evaluate to a list of threads, rather than a single thread. The list represents the computations that must continue as a result of the current execution (including the remainder or result of the current computation) and is appended to the thread queue. In that way, expressions can spawn new threads. It also becomes more convenient to terminate your own thread—just return an empty list. This notion of multitasking allows for communication between processes only through shared variables. There is no mechanism for a child's value to be returned directly to the parent. It is important that this protocol completely protects threads from each other; no entries in the thread queue other than the one currently running can be affected.

For our new architecture, we redefine the type constructor T as $\mathsf{T}(\alpha) = \alpha + (\mathsf{unit} \to (\mathsf{list}\ \mathsf{T}(\alpha)))$.[4]

[4] The two definitions of T differ only in the range of the function, in terms of $\mathsf{T}(\alpha)$. We can make use of this by fixing T as $\mathsf{T}(\alpha) = \alpha + (\mathsf{unit} \to \mathsf{S}(\mathsf{T}(\alpha)))$, where $\mathsf{S}(\mathsf{T}(\alpha))$ represents the information produced at each step in a computation to instruct the scheduler on the state of all computations. Initially S was defined as $\mathsf{S}(\mathsf{T}(\alpha)) = \mathsf{T}(\alpha)$, but it is now redefined as $\mathsf{S}(\mathsf{T}(\alpha)) = (\mathsf{list}\ \mathsf{T}(\alpha))$.

Therefore, the `return` and `bounce` procedures now correspond to the composition of `list` with the injections ι_l and ι_r. They are of types $\alpha \to (\mathsf{list}\ \mathsf{T}(\alpha))$ and $(\mathsf{unit} \to (\mathsf{list}\ \mathsf{T}(\alpha))) \to (\mathsf{list}\ \mathsf{T}(\alpha))$, respectively.

We leave the definitions of `doing` and `done` alone, but modify `return`, `bounce`, and `trampoline` to allow for dynamic creation of threads. The procedures `return` and `bounce` yield control to the next thread on the queue by returning the current thread as the sole element of a list.

$\langle$*return Definition*$\rangle+\equiv$

```
(define return
  (lambda (value)
    (list (make-done value))))
```

$\langle$*bounce Definition*$\rangle+\equiv$

```
(define bounce
  (lambda (thunk)
    (list (make-doing thunk))))
```

We modify the most recent version of `trampoline`. It need not enclose the result of continuing execution in a list, as one has already been created by `return` or `bounce`. Also, we must be wary of the thread list becoming empty. In terms of T, the type of `trampoline` is still $(\mathsf{list}\ \mathsf{T}(\alpha)) \to \alpha$.

$\langle$*trampoline for Dynamic Thread Creation*$\rangle\equiv$

```
(define trampoline
  (lambda (thread-queue)
    (if (pair? thread-queue)
      (record-case (car thread-queue)
        [done (value) value]
        [doing (thunk)
          (trampoline
            (append
              (cdr thread-queue)
              (thunk)))])
      "No thread returned a value")))
```

Since the thread queue (generated by `bounce`) contains only one element, `trampoline` simply invokes that element, yielding another singleton thread queue.

Another response to a `done` record would be to print its `value` and continue processing other threads. A second alternative would be to return the values as a stream, rather than printing them. The order in which the values are generated, and thus their order in the stream would be determined by the behavior of the scheduler, not the order of the original thread queue [10].

Returning a list of threads to trampoline gives us the added flexibility to return lists of zero, or of two or more threads. This potential is exercised by `die` and `spawn`. In `die` below, we simply return the empty list, the identity for the operation `append`. Thus, no computations are added to the list of threads, and so the current computation terminates. It is of type $\mathsf{unit} \to (\mathsf{list}\ \mathsf{T}(\alpha))$.

$\langle$*die Definition*$\rangle\equiv$

```
(define die
  (lambda ()
    '()))
```

Suppose that we need to run the last example of the introduction without `return`. We can accomplish this by letting all of the computations share a variable. Once one of them has a result, it can signal the others to die by setting this variable. We also need to add an initial `cond`-clause to both `fact-acc` and `mem?` to check this variable.

Next, we extend the protocol to enable new threads to be added to the queue. The `spawn` procedure may be applied to the result of `bounce` applications, in order to fork the computation. Up to this point, we do not increase the number of threads beyond those initially provided to the scheduler. This is the case because `die` always reduces that number by one, and `bounce` leaves it unchanged. To build a list of two or more threads, we provide `spawn`. As a result, we can dynamically create arbitrarily long lists of threads. Each tail-position expression now evaluates to an arbitrarily long list of threads. The procedure `spawn` appends these lists. Its type is $(\mathsf{list}\ \mathsf{T}(\alpha)) \times (\mathsf{list}\ \mathsf{T}(\alpha)) \rightarrow (\mathsf{list}\ \mathsf{T}(\alpha))$.

⟨*spawn Definition*⟩≡
```
(define spawn
  (lambda (threads1 threads2)
    (append threads1 threads2)))
```

We redefine `sequence` for the new architecture. It works as before, except that any subcomputations `spawn`ed by the second-argument computation must now also feed their result to the first-argument procedure. The procedure `mapcan` used in its definition is `map` with `cons` replaced by `append`. Its type is $((\alpha \rightarrow (\mathsf{list}\ \mathsf{T}(\alpha))) \times (\mathsf{list}\ \mathsf{T}(\alpha))) \rightarrow (\mathsf{list}\ \mathsf{T}(\alpha))$.

⟨*seqence Definition*⟩≡
```
(define sequence
  (lambda (f threads)
    (mapcan
      (lambda (thread)
        (record-case thread
          [done (value)
            (f value)]
          [doing (thunk)
            (bounce
              (lambda ()
                (sequence f (thunk))))]))
      threads)))
```

Our first example using `spawn` searches for the specific symbol `x` in a deeply nested list of symbols. For every pair, we `spawn` subcomputations to search the `car` and `cdr` separately. If an empty list is encountered, the thread simply dies. If a non-matching symbol is encountered, then the thread `die`s after printing it, preceded by a `^` and thus distinguished from `return`ed values. Finally, if an `x` is discovered, then it is `return`ed, wiping out all remaining computations.

⟨*search-x Definition*⟩≡
```
> (define search-x
    (lambda (t)
      (cond
        [(pair? t)
         (spawn
           (bounce
             (lambda ()
               (search-x (car t))))
           (bounce
             (lambda ()
               (search-x (cdr t)))))]
        [(null? t) (die)]
        [(eqv? t 'x) (return t)]
        [(symbol? t)
         (begin (printf "^~a " t) (die))])))
```

⟨*A search for x in a tree*⟩≡
```
> (trampoline
    (sequence
      (lambda (v)
        (if (eqv? v 'x)
          (return 'yes)
          (return 'no)))
      (search-x '(((a b c d) (x e)) (g h)))))
^a ^g ^b ^h ^c yes
> (trampoline
    (sequence
      (lambda (v)
        (if (eqv? v 'x)
          (return 'yes)
          (return 'no)))
      (search-x '(((a d) (y e)) (g h)))))
^a ^g ^d ^y ^h ^e "No thread returned a value"
```

Since `yes` is not printed on the second invocation of `search-x`, we know that `x` is not in the list and that the waiting wrapper function has not been invoked.

The Fibonacci function provides an example of the use of state for communication between threads. The use of `spawn` here creates a subcomputation for each recursive call. At the base case, the accumulator is incremented and the thread terminated. There are as many threads as the size of the result.

⟨*fib with spawn*⟩≡
```
> (define fib
    (lambda (n)
      (if (<= n 1)
        (begin
          (set! acc (add1 acc))
          (die))
        (spawn
          (bounce
            (lambda ()
              (fib (- n 1))))
          (bounce
            (lambda ()
              (fib (- n 2))))))))
> (define acc 0)
> (trampoline (fib 10))
> acc
89
```

5 Varying the Granularity of Parallelism with Multiple Trampolining

Danvy and Filinski consider how multiple applications of the CPS transformation create multiple embedding contexts, which are then susceptible to multiple control operators [5]. The potential for multiple levels of stepping, from an interpreter perspective, has been referred to by De Roure [6]. To be trampolined a second time, a program (an invocation of a scheduler) must first be reconverted to tail form. Then, the trampolined program, including the scheduler with its queue, becomes a single thread to be run on a higher-level queue. Now, operations such as `return` and `spawn` can be specified to operate at any particular level, by analogy with Danvy and Filinski's work cited above. One might also wish them to take the level at which they should operate as an additional argument. Using such tools, and given a complex application with dependencies at various levels, we can implement it in such a way that multiprocessing is achieved at each level. For example, the main project might spawn tasks 1, 2 and 3, which run on the same queue. Tasks 2 and 3 might have their own queues, to which they spawn subtasks 2a, 2b, 3a, 3b, *etc.* To accomplish this, code task 1 normally, and tasks 2 and 3 in trampolined style as described above. Then, convert each task to tail form and trampoline them.

6 Revisionist History: CPS as a Precedent for Our Methodology

Our methodology can perhaps be better understood by comparing it to the well-known example of the CPS transformation in the presence of `call/cc`, which has inspired our approach. Our examples of the trampolining transformation in the presence of multithreading operators may in turn shed some light on the relationship between CPS and `call/cc`. In this section, we re-enact the invention of `call/cc` through an example involving the `list-index` procedure.

When programs are written in CPS, it becomes reasonable to consider control operators that would have been difficult or impossible to implement in direct style. The `list-index` procedure below returns the index of the first occurrence of a number in a list of numbers, or -1 if no occurences exist. Upon reaching the end of the list, it escapes from the chain of `add1` calls by calling the final continuation directly.

⟨*list-index-cps*⟩≡
```
(define list-index
  (lambda (ls a)
    (let ([final-k (lambda (val) val)])
      (letrec
        ([list-index-cps
           (lambda (ls k)
             (cond
               [(null? ls) (final-k -1)]
               [(= (car ls) a) (k 0)]
               [else (list-index-cps (cdr ls)
                       (lambda (ind)
                         (k (add1 ind))))]))])
        (list-index-cps ls final-k)))))
```

If we were restricted to writing in the language that is the range of the CPS transformation, we would have no choice but to apply `k` (incorrectly) to `-1` in the first `cond` clause. We gain expressiveness by not imposing such a restriction and instead applying `final-k`. We can abstract this invocation of a continuation other than the given one (`k`) by using a language form `call/cc-cps`, which is easy to define as a regular procedure that can be added to a CPS program.

⟨*call/cc in CPS*⟩≡
```
(define call/cc-cps
  (lambda (p k)
    (p (lambda (v ignored-k) (k v)) k)))
```

The `call/cc` operator takes a procedure and applies it to the current continuation. Like all procedures in a CPS program, `call/cc-cps` also takes a continuation argument. Its first argument is applied to a procedure that ignores its own continuation and passes its argument to `call/cc-cps`'s continuation. We can rewrite `list-index-cps` using this abstraction.

⟨*list-index-cps with call/cc-cps*⟩≡
```
(define list-index
  (lambda (ls a)
    (call/cc-cps
      (lambda (final-k k)
        (letrec
          ([list-index-cps
             (lambda (ls k)
               (cond
                 [(null? ls) (final-k -1 k)]
                 [(= (car ls) a) (k 0)]
                 [else
                   (list-index-cps (cdr ls)
                     (lambda (ind)
                       (k (add1 ind))))]))])
          (list-index-cps ls k)))
      (lambda (v) v))))
```

When a λ_v-calculus interpreter is transformed into CPS, the possibility arises of adding new language forms that can take advantage of the new structure of the interpreter [21]. These language forms can provide advanced operations that manipulate the control context, without forcing source programs to conform to CPS.

⟨*interp-cps*⟩≡
```
(define interp-cps
  (lambda (exp env k)
    (syncase exp
      ...
      [`(call/cc ,exp)
        (interp-cps exp env
          (lambda (p)
            (p (lambda (v ignored-k) (k v)) k)))]
      ...)))
```

We first interpret the procedural argument to `call/cc`. The result is applied to a procedure representation of `k`, the current continuation. The current continuation is also passed as the default continuation. Having added `call/cc` to our interpreter, we can rewrite `list-index`.

⟨*list-index*⟩≡
```
(define list-index
  (lambda (ls a)
    (call/cc
      (lambda (final-k)
        (letrec
          ([li
             (lambda (ls)
               (cond
                 [(null? ls) (final-k -1)]
                 [(= (car ls) a) 0]
                 [else
                   (add1 (li (cdr ls) a))]))])
          (li ls))))))
```

Just as CPS makes the continuation visible during computation, trampolining makes the thread queue visible. Following the precedent of CPS above, we have developed a rewriting technique for the new style and standard ways of extending the style to provide multiprocessing capabilities. We have formalized them in a variety of multiprocessing operators. Given any interpreter in tail form, we can trampoline it and add our new operators to a source language so that they can be used without trampolined style. The procedures that gain the power of multitasking by extending the trampolined style in controlled ways then correspond to clauses of the interpreter which do the same.

7 History

There are slightly more involved but similar examples of programming styles providing contexts in which useful operators can be defined. Danvy and Filinski show that their `shift` and `reset` operators can be defined within CPS [5]. Queinnec shows that his `splitter`, `abort`, and `call/pc` can be defined within what he calls Value Transforming Style [19], based on Abstract CPS [8]. Queinnec cannot use standard CPS as his operators are dynamic and rely on the structure of the stack. An extension of Abstract CPS has been used in the Icsla work cited above. Moggi has presented a transformation to monadic style, extended to operate over languages including η and μ [16].

Cooper has shown that arbitrary programs can be rewritten to contain a single loop, using a scheme involving additional boolean variables and a complex loop condition [4]. Much work related to trampolining has already been mentioned. Bawden creates a CPS interpreter which, at each step, returns a list of the continuation and the value [2]. De Roure enhances the interpreter to support multiprogramming primitives [6]. For Tarditi's C implementation of ML [23], programs are transformed to CPS. Then, instead of making each tail call directly, the arguments are stored in an array and the address of the procedure is returned to a main `while` loop. These instances seem to have been independent, and do not use the term "trampolining". Baker appears to have been first to make use of that term in this context [1]. Wand uses a continuation-argument pair as a process representation in a non-preemptive multi-processing system [26]. His continuations, however, are captured using `catch`, a syntactic variant of `call/cc`, rather than relying on the source program being in tail form. A more extensive system for the ML language based on similar principles has been implemented by Morrisett and Tolmach [18].

Haynes, Friedman, and Wand have demonstrated that coroutines can be implemented using `call/cc` [12]. The threads we have introduced differ from coroutines in that the latter require that control be yielded to a particular coroutine, and provide for a value to be communicated to that coroutine. Our threads can be implemented in coroutines by designating one coroutine as a scheduler and requiring other coroutines to yield only to the scheduler, passing a dummy value. At the other extreme are preemptive systems where threads are interrupted by the scheduler. We can implement this using a trampolined interpreter. Dybvig and Hieb have shown that engines can be implemented using `call/cc` [7]. Conversely, Kumar, Bruggeman, and Dybvig have demonstrated that continuations, and in particular partial, composable continuations, can be implemented using threads [15]. Shivers has presented an implementation of threads using multi-continuation CPS [22].

The benefits of the precision gained using partial, composable continuations to capture control context, particularly when multiple threads are running, have been described by Hieb, Dybvig, and Anderson [13]. We agree but have chosen to focus on other aspects at this time. The issue is closely related to the multiple iterations of trampolining.

The Icsla language [20] is perhaps closest to our own, in that it implements multiprocessing primitives through a conversion of programs to a particular style. That style, however, is a variant of Abstract CPS. Our trampolined style is less intrusive on the structure of programs.

8 Conclusion

We have presented trampolined style through two architectures. A transformation to this style provides significant multithreading capabilities without language support for continuations. Programs then have a single loop in which computations are processed in discrete steps. Trampolined expressions evaluate to information (including the remainder of their computation) that is used by the scheduler. Each architecture requires the definition of three procedures, including the scheduler, which must be invoked as the operator in an application of the result of the transformation. For each architecture, we have extended the style in a constrained way, generally through the use of an operator that intercepts the evaluation results of trampolined expressions. Finally, we have demonstrated how each operator could be implemented in a trampolined interpreter, and how programs with the operators could then be rewritten to avoid the use of the trampolined style. In the future, we wish to investigate both the potential for monadic implementations of each of these styles and their semantic properties. We also expect to survey what other domains might benefit from this perspective.

Acknowledgments

We gratefully acknowledge the assistance of Jonathan Sobel and Jonathan G. Rossie for their detailed reading and criticisms of assorted drafts of this paper, which have led us to make considerable clarifications and improvements in the presentation of this final version.

References

[1] Henry Baker. Cons should not cons its arguments, Part II: Cheney on the M.T.A. *SIGPLAN Notices*, 30(9):17–20, September 1995.

[2] Alan Bawden. Reification without evaluation. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 342–351, 1988.

[3] Daniel G. Bobrow and Ben Wegbreit. A model and stack implementation of multiple

environments. *Communications of the ACM*, 16(10):591–603, October 1973.

[4] David Cooper. Böhm and Jacopini's reduction of flow charts. *Communications of the ACM*, 10(8):463,473, August 1967. Letter to the Editor.

[5] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160, Nice (France), June 1990. ACM, ACM Press.

[6] David De Roure. QPL3—continuations, concurrency and communication. Technical Report CSTR 90-20, Department of Electronics and Computer Science, University of Southampton, 1990.

[7] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[8] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 52–62, Snowbird (Utah USA), July 1988.

[9] Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, 1993. Revised from the proceedings of the 1972 ACM Conference on Proving Assertions about Programs.

[10] Daniel P. Friedman and David S. Wise. An indeterminate constructor for applicative programming. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 245–250, Las Vegas (Nevada USA), January 1980.

[11] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2):109–121, 1987.

[12] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. *Computer Languages*, 11(3/4):143–153, 1986.

[13] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson. Subcontinuations. *Lisp and Symbolic Computation*, 6:453–478, 1993.

[14] Stephen Cole Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

[15] Sanjeev Kumar, Carl Bruggeman, and R. Kent Dybvig. Threads yield continuations. *Lisp and Symbolic Computation*, 10(3):223–236, May 1998.

[16] Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings of the* 4^{th} *Annual IEEE Symposium on Logic in Computer Science, LICS '89*, pages 14–23, Pacific Grove (California USA), June 1989. ACM, IEEE.

[17] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[18] J. Gregory Morrisett and Andrew Tolmach. A portable multiprocessor interface for Standard ML of New Jersey. Technical Report CMU-CS-92-155, Carnegie Mellon University, June 1992. Also appears as Princeton University TR-376-92.

[19] Christian Queinnec. Value transforming style. In M. Billaud, P. Castéran, M. M. Corsini, K. Musumbu, and A. Rauzy, editors, *WSA '92—Workshop on Static Analysis*, number 81-82 in Bigre Journal, pages 20–28, Bordeaux (France), September 1992.

[20] Christian Queinnec and David De Roure. Design of a concurrent and distributed language. In Robert H. Halstead, Jr. and Takayasu Ito, editors, *Parallel Symbolic Computing: Languages, Systems, and Applications, (US/Japan Workshop Proceedings)*, volume LNCS 748, pages 234–259, Boston (Massachussetts USA), October 1993.

[21] John C. Reynolds. Definitional interpreters for higher order programming languages. In *Proceedings of the ACM 25th National Conference*, pages 717–740, Boston (Massachussetts USA), August 1972. ACM, ACM Press.

[22] Olin Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, pages 2–1—2–15, Paris (France), January 1997. ACM Press.

[23] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: A standard ML to C compiler. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.

[24] A. van Wijngaarden. Recursive definition of syntax and semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming*, pages 13–24. North-Holland, 1966.

[25] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 61–78, Nice (France), June 1990.

[26] Mitchell Wand. Continuation-based multiprocessing. *Higher-Order and Symbolic Computation*, 12(3), 1999. Reprinted from the proceedings of the 1980 Lisp Conference, pages 19–28.