Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

Trampolining Architectures

Steven E. Ganz · Daniel P. Friedman

Received: date / Accepted: date

Abstract Motivated by implementation of multitasking, we define a trampolining architecture as a generalization and extension of a monad. We present seven trampolining architectures, operating over the range of a trampolining translation. The effects of the architectures are cumulative, except for the last two. Some increase the breadth of multitasking facilities provided. Others demonstrate the potential for more efficient implementation. The final two architectures illustrate the full generality of the translation via demonstrations of the direct applicability of trampolining to languages without closures and of support for logic programming, respectively. It is reassuring but perhaps not surprising that such extensions are possible. It is more curious that all but the first three are monads only subject to the identification of yielding control and proceeding immediately with execution, i.e., without regard to intervening execution of other threads. We do not claim that trampolining architectures of similar power cannot be formulated as monads without this relaxation of equality but only that our generalized monads are sufficient to model practical multithreading systems.

Keywords trampolining · monad

1 Introduction

In previous work, Ganz, Friedman, and Wand formalized trampolining as a technique by which multitasking can be implemented directly through program transformations in a language without first-class continuations [10]. Trampolined style is a way of writing programs such that a single "scheduler" loop manages all transfers of control. Computations are executed in discrete steps. After a computation performs a step,

Steven E. Ganz
Genetius, Inc., Emerald Hills, California 94062
Tel.: +408-591-9872
E-mail: steven.ganz@gmail.com

Daniel P. Friedman
Indiana University, Bloomington, Indiana 47405. Tel.: 812-855-4885
Fax: 812-855-4829
E-mail: dfried@cs.indiana.edu

the remaining work is returned to the scheduler. Programs in tail form are converted to trampolined style through the trampolining translation, which inserts applications of `bounce` and thunks around procedure calls. Transformed programs must be run through a scheduler, called `trampoline`. Non-primitive applications are thus delayed so that they take place within the scheduler loop. Properly translated programs satisfy a liveness invariant; they must return control to the scheduler after a bounded amount of computation. Another way of saying this is that any possible infinite loop must include the scheduler loop. This transformation serves to convert non-cooperating code into cooperating code, by forcing it to yield control often enough to avoid starvation of other threads. Filinski independently described multithreading in terms of a layering of the resumption monad on a list monad [9].

In this work, we make two changes to the above framework. First, we remove the tail-form restriction on source-language programs, and translate into a modified monadic metalanguage. The translation is then similar to Moggi's monadic translation [22], but inserts `bounce` forms.[1] Second, we divide the suspended computation expected by `bounce` so that it is not just a thunk, but a more general procedure along with a machine state. The `bounce` forms inserted by the translation are each applied to a `lambda` abstraction over variables identifying the machine state and then references to those variables. The abstractions interrupt a computation and the variable references provide the state with which to resume it. In the process, the code that continues the computation is given access to the machine state through a form of reflection [31].

We thus parameterize trampolined style by a vector $\bar{x}$ of variables and two language forms: `trampoline` and `bounce`, as well as by two monadic operators: `unit` and `extend`.[2] The final component of the Kleisli triple, the object component of the functor, is provided by partially applying a curried type constructor M (indexed by ρ), so that $\mathsf{M}\rho\alpha$ describes an intermediate state of a computation returning a value of type α. We generally define M in terms of a type constructor T, where an object of type $\mathsf{T}\rho\alpha$ is a record describing a thread of return type α. If $\bar{x}$ are of types $\bar{\tau}$, then `trampoline` is of type $(\forall\rho.\bar{\tau}{\to}\mathsf{M}\rho\alpha) \to \bar{\tau}{\to}\alpha$ and `bounce` is of type $(\bar{\tau}{\to}\mathsf{M}\rho\alpha) \to \bar{\tau}{\to}\mathsf{M}\rho\alpha$. The universal quantification over the type variable ρ in the type of `trampoline` ensures that references to the computation being run do not escape into the result value [18]. It is not particularly important to our purpose here; we generally use `trampoline` as the outermost syntactic form. Otherwise, it would serve to ensure that a thread from one `trampoline` is not scheduled by another. More explicitly, in the absense of this mechanism, one could define operators such that

```
(let ((thread (trampoline
                 Code that suspends execution of and returns a thread)))
  (trampoline Code that resumes execution of that thread))
```

would typecheck although it violates the abstraction created by `trampoline`.[3]

Appropriate definitions of the four operators and the type constructor comprise a trampolining architecture, a generalization and extension of a monad. "Appropriate" here includes satisfaction of the standard monadic laws [34, 20] only subject to

[1] It also differs in the reflective operators and minimizes insertion of monadic `let` (`m-let`) constructs.

[2] We use Filinski's formulation [8], referring to η as `unit` and $-^*$ as `extend`, since this reduces the size of our operator definitions. The original paper [10] used uncurried $-^*$ as `sequence` for `extend`, and η as `return` for `unit`.

[3] It should be possible for the two occurrences of `trampoline` to enforce different architectures with different representations of threads.

the identification of yielding control and proceeding immediately with execution, i.e., without regard to any intervening execution of other threads. If we let $=$ represent the congruence closure of reduction and $\approx$ represent the congruence closure of:

$$= \cup \{(\texttt{bounce},\ \texttt{id}_{\bar{\tau}\to\mathsf{M}\rho\alpha})\}$$

then the monad laws are expressible as[4,5]:

$$(\circ\ (\texttt{extend}\ recvr)\ \texttt{unit}) \approx recvr \tag{1}$$

$$(\texttt{extend}\ \texttt{unit}) \approx \texttt{id}_{M\rho\alpha} \tag{2}$$

$$(\texttt{extend}\ (\circ\ (\texttt{extend}\ f)\ g)) \approx (\circ\ (\texttt{extend}\ f)\ (\texttt{extend}\ g)) \tag{3}$$

The relaxation is somewhat compensated for by the following additional constraints of "appropriateness" that must hold over the unmodified reduction relation. The first is a requirement that `bounce` commute with `extend`. The result of appending a receiving procedure *recvr* onto a suspended computation should still be a suspended computation. Often it will be the suspension of the original computation with *recvr* appended, but we will find it useful in Section 5.2 to be more permissive. In the common case, we have a fourth law[6]:

$$\begin{array}{l}(\circ\ (\texttt{extend}\ recvr)\ (\texttt{bounce}\ susp)) = \\ (\texttt{bounce}\ (\circ\ (\texttt{extend}\ recvr)\ susp))\end{array} \tag{4}$$

The second additional constraint asserts that some number of `bounce` forms can be eliminated when set just inside a `trampoline` form.

$$\exists i > 0.(\circ\ \texttt{trampoline}\ \texttt{bounce}^i) = \texttt{trampoline} \tag{5}$$

where $\texttt{bounce}^i$ indicates the i-fold composition of `bounce`.

We are able to claim that a trampolining architecture is a generalization of a monad because one can always define $\bar{x} = \epsilon$ and `bounce` to be $\texttt{id}_{1\to\mathsf{M}\rho\alpha}$. `trampoline` corresponds to a monad's `run`.

The creation of a trampolining architecture allows for the possibility of the definition of additional operators. Several trampolining architectures were presented in prior work [10]: three variations of a simple trampolining architecture based on the resumption monad [24] and an architecture based on a combination of the resumption and list monads. The former allowed for the implementation of breakpoints and engines in a one-, two- or multi-thread system by having each computation yield a thread after each unit of work is performed. The latter allowed for dynamic thread creation and termination through `spawn` and `die` operators. This was accomplished by having each computation yield a list of threads to be added to the thread queue at each step.

A goal of this paper is to show that these techniques are more generally applicable. To this end, we demonstrate an assortment of architectures that provide, in addition to variations on the functionality expected of any multitasking system, support for

[4]For brevity and in deference to their category-theoretic primacy, we refer to a binary compose primitive `(lambda (x y) (lambda (z) (x (y z))))` as $\circ$ and to an identity primitive `(lambda (x) x)` as `id`. Because of the associativity of composition, we freely use the composition operator with arbitrary arity. Where it may be helpful, we subcript `id` with the type over which it operates.

[5]We assume throughout that variables in types or formulae that are not bound explicitly or by context are assumed to be bound by an implicit outermost universal quantification.

[6]Here, any $\bar{x}$ may occur free in *comp*, but we assume without loss of generality that $\bar{x}$ do not occur free in *recvr*.

another paradigm and the basis for a more efficient implementation. We begin in Section 2 by reviewing and generalizing the trampolining translation, and then proceed to review trampolining architectures for stepping through a computation and for dynamic creation and termination of threads, as a starting point for our enhancements. We next present a series of trampolining architectures with the purpose of demonstrating additional functionality or the potential for more improved efficiency. These architectures allow more fair scheduling of threads than under the first architecture, recycling of thread records, synchronous communication between threads, the avoidance of closure introduction, and logic programming. The architecture for avoiding closure introduction suggests an enhanced trampolining translation (which we describe only informally) to convert variable references to make use of the machine state.

A more theoretically compelling goal of this paper is to reflect on the relevance of the monadic model to these architectures. Thus, after each trampolining architecture, we state as a theorem the extent to which that architecture conforms to the monadic laws. We find that only our beginning architectures for dynamic creation and termination and more fair scheduling of threads in Section 3 are monads outright in the category that equates equivalent terms of our programming language. Section 4.1 considers alternative versions of those architectures that satisfy the laws under a more permissive notion of equality ($\approx$) that does not distinguish the presence of bounce forms.[7] The remaining architectures in Sections 4 and 5 satisfy the monadic laws only with this more pervasive notion of equality.

The architectures in Section 5 demonstrate the full generality of the trampolining translation and its potential for reflection in making the implementation state available to user code. The architecture for closure avoidance makes a conventional machine state available. The architecture for logic programming provides access to the substitution.

Throughout, we use the Scheme programming language and conform to R^{6}RS [6], from which we make use of record definitions. Our code examples are basically self-contained; exceptions such as stream operators, a syntactic record analyzer, a unique symbol generator, and a simple macro definition construct should provide little difficulty. We assume familiarity with the definition of monads and their use in functional programming [34], but not with category theory more generally.

2 The Trampolining Translation

Before proceeding to present trampolining architectures, we restate the trampolining transformation [10] to operate over arbitrary programs, not necessarily in tail form. We use (possibly subscripted) E, S, C, x, and c as metavariables for expressions, simple expressions, complex expressions, variables, and constants, respectively. The first three of these syntactic categories are defined in the grammar in Figure 1. Expressions consist of constants, variables, lambda abstractions, assignments, applications of constant functions, conditionals, sequences, general applications, and reflections into a thread's state (modifying that state). Explicit reifications are not needed because the implementation is always available as $\bar{x}$. Expressions are partitioned into simple and

[7] Although the monad laws do not hold in the category of the programming language for these architectures, those laws do hold in the category that differs in identifying `bounce` morphisms with the relevant identity morphisms. However, the additional equation is not without import; adding sufficient equations would render the theory trivial, as, e.g., in the case of Fexprs [35].

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

Fig. 1 Source Language Grammar

$$
\begin{array}{rcl}
S^{\bar{x}} & ::= & c \mid x \mid (\texttt{lambda}\ (x)\ C^{\bar{x}}) \mid (\texttt{set!}\ x\ S^{\bar{x}}) \\
& \mid & (c\ S^{\bar{x}}) \mid (\texttt{if}\ S^{\bar{x}}\ S^{\bar{x}}\ S^{\bar{x}}) \mid (\texttt{begin}\ S^{\bar{x}}\ S^{\bar{x}}) \\
C^{\bar{x}} & ::= & (\texttt{unit}\ S^{\bar{x}}) \mid (\texttt{let}\ (\overline{(x\ S^{\bar{x}})})\ C^{\bar{x}}) \mid (\texttt{m-let}\ (x\ C^{\bar{x}})\ C^{\bar{x}}) \\
& \mid & (\texttt{if}\ S^{\bar{x}}\ C^{\bar{x}}\ C^{\bar{x}}) \mid (\texttt{begin}\ S^{\bar{x}}\ C^{\bar{x}}) \mid (\texttt{m-begin}\ C^{\bar{x}}\ C^{\bar{x}}) \\
& \mid & ((\texttt{bounce}\ (\texttt{lambda}\ (\bar{x})\ C^{\bar{x}}))\ \bar{x})
\end{array}
$$

Fig. 2 Target Language Grammar

complex expressions; the former exclude general applications and reflections, except in the body of lambda abstractions.

The syntactic categories of complex and simple expressions are redefined for the target language of the translation in the grammar in Figure 2. These syntactic categories are superscripted by a sequence of variable names $\bar{x}$, representing the internal machine state. Simple expressions are otherwise unchanged except that `lambda` expressions now have a complex body. Complex expressions no longer include either `reflect` or `set!`, `if`, or `begin` expressions with complex subexpressions in head position. They do include the monadic forms `unit`, `m-let`, and `m-begin`, the former of which wraps a simple expression and the latter two of which do allow for complex subexpressions in nontail position. Complex expressions are narrowly defined to also include only `let` forms that bind the designated variables $\bar{x}$ and `bounce` forms applied to an abstraction over $\bar{x}$ and then references to $\bar{x}$.

In both the source and target languages, we assume that constants include boolean literals. The target language is intended to be a subset of Scheme (augmented with definitions that we will provide) and so we will make reference to evaluation and reduction in Scheme with respect to this language.

Figure 3 presents the trampolining transformation through two mutually recursive translations, both indexed by $\bar{x}$: a main translation $\mathcal{E}^{\bar{x}}[]$, on expressions E yielding a $C^{\bar{x}}$and an auxiliary translation $\mathcal{S}^{\bar{x}}[]$ yielding an $S^{\bar{x}}$. Throughout the original program, all calls of user-defined procedures are wrapped in `((bounce (lambda (`$\bar{x}$`) [])) `$\bar{x}$`)`, and all simple values in tail position are wrapped in `(unit [])`. When complex expressions occur in nontail position, a `(m-let (`x `[]) []) `form is inserted, with the complex expression forming the right-hand side and the evaluation context of the complex expression, with x substituted for the hole, forming the body. In the case of `begin`, however, we forgo variable binding and translate to `(m-begin [] [])`. The intension is that `m-let` and `m-begin` operate over computations; they are defined in terms of `extend` in Figure 5. `reflect` allows user code to specify implementation data as simple source language terms by binding $\bar{x}$ to them over the scope of the code. The main translation $\mathcal{E}$ and an auxiliary transformation $\mathcal{S}$ are used to recur on subexpressions: $\mathcal{E}$ on arbitrary expressions and $\mathcal{S}$ only on simple expressions. This translation is nondeterministic; it could

$$
\begin{aligned}
\mathcal{E}^{\bar{x}}[E] &\in C^{\bar{x}} \\
\mathcal{E}^{\bar{x}}[c] &= (\texttt{unit}\ c) \\
\mathcal{E}^{\bar{x}}[x] &= (\texttt{unit}\ x) \\
\mathcal{E}^{\bar{x}}[(\texttt{lambda}\ (x)\ E)] &= (\texttt{unit}\ (\texttt{lambda}\ (x)\ \mathcal{E}^{\bar{x}}[E])) \\
\mathcal{E}^{\bar{x}}[(\texttt{set!}\ x\ S)] &= (\texttt{unit}\ (\texttt{set!}\ x\ \mathcal{S}^{\bar{x}}[S])) \\
\mathcal{E}^{\bar{x}}[(\texttt{set!}\ x\ C)] &= (\texttt{m-let}\ (\texttt{v}\ \mathcal{E}^{\bar{x}}[C])\ \mathcal{E}^{\bar{x}}[(\texttt{set!}\ x\ \texttt{v})]) \\
\mathcal{E}^{\bar{x}}[(c\ S)] &= (\texttt{unit}\ (c\ \mathcal{S}^{\bar{x}}[S])) \\
\mathcal{E}^{\bar{x}}[(c\ C)] &= (\texttt{m-let}\ (\texttt{v}\ \mathcal{E}^{\bar{x}}[C])\ \mathcal{E}^{\bar{x}}[(c\ \texttt{v})]) \\
\mathcal{E}^{\bar{x}}[(\texttt{if}\ S\ E_1\ E_2)] &= (\texttt{if}\ \mathcal{S}^{\bar{x}}[S]\ \mathcal{E}^{\bar{x}}[E_1]\ \mathcal{E}^{\bar{x}}[E_2]) \\
\mathcal{E}^{\bar{x}}[(\texttt{if}\ C\ E_1\ E_2)] &= (\texttt{m-let}\ (\texttt{v}\ \mathcal{E}^{\bar{x}}[C])\ \mathcal{E}^{\bar{x}}[(\texttt{if}\ \texttt{v}\ E_1\ E_2)]) \\
\mathcal{E}^{\bar{x}}[(\texttt{begin}\ S\ E)] &= (\texttt{begin}\ \mathcal{S}^{\bar{x}}[S]\ \mathcal{E}^{\bar{x}}[E]) \\
\mathcal{E}^{\bar{x}}[(\texttt{begin}\ C\ E)] &= (\texttt{m-begin}\ \mathcal{E}^{\bar{x}}[C]\ \mathcal{E}^{\bar{x}}[E]) \\
\mathcal{E}^{\bar{x}}[(S_1\ S_2)] &= ((\texttt{bounce}\ (\texttt{lambda}\ (\bar{x})\ (\mathcal{S}^{\bar{x}}[S_1]\ \mathcal{S}^{\bar{x}}[S_2])))\ \bar{x}) \\
\mathcal{E}^{\bar{x}}[(C\ E)] &= (\texttt{m-let}\ (\texttt{v}\ \mathcal{E}^{\bar{x}}[C])\ \mathcal{E}^{\bar{x}}[(\texttt{v}\ E)]) \\
\mathcal{E}^{\bar{x}}[(E\ C)] &= (\texttt{m-let}\ (\texttt{v}\ \mathcal{E}^{\bar{x}}[C])\ \mathcal{E}^{\bar{x}}[(E\ \texttt{v})]) \\
\mathcal{E}^{\bar{x}}[(\texttt{reflect}\ (\overline{S})\ E)] &= (\texttt{let}\ (\overline{(x\ \mathcal{S}^{\bar{x}}[S])})\ \mathcal{E}^{\bar{x}}[E])
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}^{\bar{x}}[S] &\in S^{\bar{x}} \\
\mathcal{S}^{\bar{x}}[c] &= c \\
\mathcal{S}^{\bar{x}}[x] &= x \\
\mathcal{S}^{\bar{x}}[(\texttt{lambda}\ (x)\ E)] &= (\texttt{lambda}\ (x)\ \mathcal{E}^{\bar{\texttt{x}}}[E]) \\
\mathcal{S}^{\bar{x}}[(\texttt{set!}\ x\ S)] &= (\texttt{set!}\ x\ \mathcal{S}^{\bar{\texttt{x}}}[S]) \\
\mathcal{S}^{\bar{x}}[(c\ S)] &= (c\ \mathcal{S}^{\bar{\texttt{x}}}[S]) \\
\mathcal{S}^{\bar{x}}[(\texttt{if}\ S_1\ S_2\ S_3)] &= (\texttt{if}\ \mathcal{S}^{\bar{\texttt{x}}}[S_1]\ \mathcal{S}^{\bar{\texttt{x}}}[S_2]\ \mathcal{S}^{\bar{\texttt{x}}}[S_3]) \\
\mathcal{S}^{\bar{x}}[(\texttt{begin}\ S_1\ S_2)] &= (\texttt{begin}\ \mathcal{S}^{\bar{\texttt{x}}}[S_1]\ \mathcal{S}^{\bar{\texttt{x}}}[S_2])
\end{aligned}
$$

Fig. 3 Trampolining Translation

be made deterministic by selecting an order of evaluation for applications. Technically, it would still be nondeterministic, but we assume applications of constant functions are always translated as such and not as more general applications. The program resulting from applying $\mathcal{E}^{\bar{x}}[E]$ must be wrapped in `(trampoline (lambda (`$\bar{x}$`) []))` to be properly executed, where E may reference the state of the computation through $\bar{x}$.

Continuation operators such as `call/cc` can be made available as constant functions. We use $\mathcal{E}$ over derived forms (such as `let`, `letrec`, and `cond`), although those cases are not specified here. The translation is easily extended to translate-in-place the operators that are supported by each trampolining architecture. Further comments regarding the translation are provided in the original presentation [10].

As a simple example, we present a trampolined function (with $\bar{x} = \epsilon$) that computes the factorial function. Given an initial function:

```
(define fact
  (lambda (n)
    (if (zero? n)
      1
      (* n (fact (sub1 n))))))
```

```
(define-record-type done (fields (immutable value)))
(define-record-type doing (fields (immutable thunk)))

(define unit make-done)

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

(define extend
  (lambda (recvr)
    (lambda (thread)
      (record-case thread
        (done (value) (recvr value))
        (doing (thunk) ((bounce (∘ (extend recvr) thunk))))))))

(define tramp
  (lambda (thread)
    (record-case thread
      (done (value) (stream-cons value (make-empty-stream)))
      (doing (thunk) (tramp (thunk))))))

(define trampoline (∘ tramp make-doing))
```

Fig. 4 Stepping Architecture

the trampolined version is:

```
(define fact-tramp
  (lambda (n)
    (if (zero? n)
      (unit 1)
      (m-let (v ((bounce (lambda () (fact-tramp (sub1 n))))))
        (unit (* n v))))))
```

and it can be run as:

```
(trampoline (lambda () (fact-tramp m)))
```

3 Monadic Trampolining Architectures

3.1 A Stepping Architecture

We first present a single-threaded architecture in Figure 4. The Stepping Architecture has each expression generated by $\mathcal{E}$ evaluate to a thread. Thus, we define the type constructor M, representing computations, as $\mathsf{M}\rho\alpha = \mathsf{T}\rho\alpha$.

For this architecture, we define the type constructor T, representing threads, as $\mathsf{T}\rho\alpha = \alpha + (1 \to \mathsf{M}\rho\alpha)$. The record types `done` and `doing` correspond to the two thread variants. The record type `done` represents a completed computation and holds a result value of any type α. The record type `doing` represents an incomplete computation and holds a thunk that performs the remaining work.

The procedure `unit` creates a `done` thread holding its argument, the result of a computation. This packaging instructs the scheduler to emit the value. Similarly, `bounce` creates a `doing` thread to package a computation so that it can be resumed by the

```
(rewrites-as (m-let (?var ?rhs) ?body) ((extend (lambda (?var) ?body)) ?rhs))

(rewrites-as (m-let* () ?body) ?body)
(rewrites-as (m-let* ((?var0 ?rhs0) (?var1 ?rhs1) ...) ?body)
  (m-let (?var0 ?rhs0) (m-let* ((?var1 ?rhs1) ...) ?body)))

(rewrites-as (m-begin ?exp1) ?exp1)
(rewrites-as (m-begin ?exp1 ?exp2 ?exp3...)
  (m-let (dummy ?exp1) (m-begin ?exp2 ?exp3...)))
```

Fig. 5 Derived Forms for All Architectures

scheduler. The `unit` and `bounce` procedures correspond to the injections ι_l and ι_r, respectively. In the case of `bounce`, however, the injection is delayed. For this architecture and all succeeding ones until Section 5, we set $\bar{x} = \epsilon$. Thus, a suspended computation is modeled by a procedure of no arguments, and `bounce` takes and returns a thunk. The two procedures are of types $\alpha \to \mathsf{M}\rho\alpha$ and $(1 \to \mathsf{M}\rho\alpha) \to 1 \to \mathsf{M}\rho\alpha$, respectively.

We now define a sequential composition operator. `extend` takes an initial computation, i.e., a thread, after a trampolined procedure `recvr` of one argument. If the initial computation has completed, work continues by passing its result to `recvr`. Otherwise, the doing record is rebuilt with `recvr` recursively extended to access the thread produced by the thunk. The type of `extend` is $(\alpha_1 \to \mathsf{M}\rho\alpha_2) \to \mathsf{M}\rho\alpha_1 \to \mathsf{M}\rho\alpha_2$. We thus have an implementation of the Resumption monad [21,24], with ι_l serving as the `unit`.

The definitions in Figure 5 allow us to use `m-let`, `m-let*`, or `m-begin` in place of `extend`, for a slightly more pleasant syntax. `m-begin` is convenient when the second computation is independent of the result of the first; in this case no variable need be declared. We assume a macro facility that accepts a sequence of clauses to be tried in order, each containing a pattern and an expansion. Variables introduced in the expansion are hygienic, i.e., they will not clash with surrounding names [16].

The scheduler, called `trampoline`, runs a computation. In anticipation of our remaining architectures, it makes the results available in a stream, with remaining processing delayed until additional results are requested. At this point, however, only a singleton stream is returned. We assume throughout definitions of `make-empty-stream` and `stream-cons`. The type of `trampoline` is $(\forall\rho.1 \to \mathsf{M}\rho\alpha) \to \mathsf{Stream}\ \alpha$. The type variable α here is bound to the type of result that might be produced by the computation. `trampoline` uses a helper procedure `tramp` of type $(\forall\rho.\mathsf{T}\rho\alpha) \to \mathsf{Stream}\ \alpha$. `tramp` terminates upon any completed thread, emitting the result. Upon an imcomplete thread, it steps by recurring on the thread produced by invoking the thunk.

We can run `fact-tramp` through our Stepping Architecture.

```
> (trampoline (lambda () (fact-tramp 5)))
[120]
```

We present streams as lists delimited with '[' and ']'. We use an elipsis ('...') after a stream element to indicate an infinite sequence of that element. We use '***' at the end of a stream to indicate that the computation diverges with no additional results generated.

Our `trampoline` scheduler continues a computation until it has completed. We can replace it with one that interactively adjusts a bound, so that a computation is only performed for a fixed number of steps at a time. Such a bounded `trampoline` would demonstrate some of the power of an engine [13]. 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. `interactive-trampoline` still creates a `doing` thread, but passes it to a specialized `tramp` routine returned by `make-engine`. `make-engine` expects a number of ticks, and is initially invoked with `0`. The resulting routine's processing is interrupted when the number of ticks reaches zero, and the user is interactively queried as to whether to supply more. If more ticks are provided, the computation continues. Otherwise, it terminates without producing a value.

```
(define interactive-trampoline
  (lambda (thunk)
    ((make-engine 0) (make-doing thunk))))

(define make-engine
  (lambda (ticks)
    (lambda (thread)
      (record-case thread
        [done (value) (stream-cons value (make-empty-stream))]
        [doing (thunk)
          (if (zero? ticks)
            (begin
              (printf ``~nHow many more? '')
              (let ([ticks (read)])
                (if (zero? ticks)
                  (make-empty-stream)
                  ((make-engine ticks) (thunk)))))
            ((make-engine (sub1 ticks)) (thunk)))]))))
```

We can now step through `fact-tramp`:

```
> (interactive-trampoline (lambda () (fact-tramp 10)))
How many more? 4
How many more? 4
How many more? 4
[3628800]
>
```

As it stands, this would allow us to probe an unknown trampolined function for a result without risking divergence. Similar technology could form the basis of a debugger.

Theorem 3.1 *In Stepping,* Mρ*,* `unit`*, and* `extend` *form a monad.*
The following laws hold:

1. `(∘ (extend` *recvr*`) unit)` = *recvr*
2. `(extend unit)` = `id`
3. `(extend (∘ (extend` f`) `g`))` = `(∘ (extend` f`) (extend` g`))`

Proof For the first law, because the initial computation is `done`, we need only consider the `done` case in `extend`, and the left side reduces to the right.

For the second law, the situation is more complex. It is the `recvr` that is `unit`, so `extend` needs to handle either kind of thread. `(extend unit)` equals:

```
(lambda (thread)
  (record-case thread
    (done (value)
     (make-done value))
    (doing (thunk)
     (make-doing (compose (extend unit) thunk)))))
```

For the `done` case, we certainly have an identity function. But for the `doing` case, we can only say that `(extend unit)` = `id` if `(extend unit)` = `id`. One might be tempted to perform an induction, perhaps on the number of `doing` records encountered before hitting a `done` record. Aside from the awkwardness and failure to generalize cleanly to our other architectures, this approach has the more serious limitation that we may never hit a `done` record. We feel strongly that two expressions should not be considered equivalent merely because both engender non-terminating computations. We therefore instead give an argument by coinduction.

Koutavas and Wand consider using bisimulations to reason about contextual equality in higher-order imperative programs using a big-step semantics [17]. We take a similar approach here, using intuitive arguments about the semantics of Scheme. One might be concerned that the use of big-step semantics would restrict us to considering only terminating programs. In fact, this is not the case. In our trampolined programs, there are no loops except that of the scheduler, so the expressions that we need to compare will always converge; we need only ensure that they converge to corresponding values.

Relations R explicitly associate values. Such a value relation can be extended to an expression relation R^* that matches expressions that differ only by R-related values, and then restricted back to the subrelation R^*_{val} consisting only of values.

Koutavas and Wand define a set $\mathcal{X}$ of tuples, each composed of a pair of stores and a value relation R, in order to demonstrate a bisimulation. We elide the description of stores because they are not central to the argument, and allow $mathcalX$ to indicate a set of value relations. It seems clear that the techniques of Felleisen and Hieb [7] should allow the argument to proceed in the more general case including both `set!` and `call/cc`.

They enumerate conditions under which $\mathcal{X}$ demonstrates a bisimulation. We will get by with only the constraint for abstractions[8], which we are able to simplify, both because, as mentioned, we elide state and because we will not need their induction hypotheses. In fact, it is still more complex than we need, as our $\mathcal{X}$ will contain only a single relation R.
Given the definitions:

$$R \vdash_{\mathcal{X}} e \sqsubseteq e' = \quad e \Downarrow w \quad \Rightarrow \quad \exists w', Q: \; e' \Downarrow w' \;\wedge\; Q \in \mathcal{X} \;\wedge\; (w, w') \in Q^*_{\mathsf{val}} \;\wedge\; R \subseteq Q$$
$$R \vdash_{\mathcal{X}} e \sqsupseteq e' = \quad e' \Downarrow w' \quad \Rightarrow \quad \exists w, Q: \; e \Downarrow w \;\wedge\; Q \in \mathcal{X} \;\wedge\; (w, w') \in Q^*_{\mathsf{val}} \;\wedge\; R \subseteq Q$$

If $(\texttt{(lambda (}x\texttt{) }e\texttt{)}, \texttt{(lambda (}x\texttt{) }e'\texttt{)}) \in R$ and $(v, v') \in R^*_{\mathsf{val}}$,
then $R \vdash_{\mathcal{X}} \texttt{((lambda (}x\texttt{) }e\texttt{) }v\texttt{)} \sqsubseteq \texttt{((lambda (}x\texttt{) }e'\texttt{) }v'\texttt{)}$
$R \vdash_{\mathcal{X}} \texttt{((lambda (}x\texttt{) }e\texttt{) }v\texttt{)} \sqsupseteq \texttt{((lambda (}x\texttt{) }e'\texttt{) }v'\texttt{)}$

[8] They require that each pair in R be of the same syntactic form, and that some constraint hold, determined by the choice of syntactic form. We describe above the constraint for abstractions. For constants, the values must be equal. Product terms in R must be of the same structure and corresponding contained values must be R^*_{val}-related.

The goal is a set of relations on abstractions $\mathcal{X}$ that satisfies the constraint. Following Koutavas and Wand, we consider values formed by closing the expressions that we wish to demonstrate equivalent, and notice that these abstractions are equivalent iff the open expressions were. In this case, however, the expressions contain no free variables but we still enclose each in a thunk to obtain values:

`(lambda () (extend unit))` and `(lambda () id)`

We form a value relation R consisting of the single pair matching the two abstractions. The idea is to let R relate the two abstractions above and then work instead with an application of each abstraction to a value, assuming the values to be related by R^*_{val}. We can thus assume that in the expressions we wish to show equivalent, any free variables denote values. In this case, again, we have only applications of thunks.

Both applications converge. By the constraint on $\mathcal{X}$, we need to show that the results of applying the two thunks are related by R^*_{val}. We show instead by equational reasoning that the results of applying the two thunks are equivalent to terms related by R^*_{val}.[9].

Consider the application:

```
((lambda () (extend unit)))
```

This converges to:

```
(lambda (thread)
  (record-case thread
    (done (value) (make-done value))
    (doing (thunk)
      ((bounce (∘ (extend make-done) thunk))))))
```

One obtains from equational reasoning that the above is equivalent to:

```
(lambda (thread)
  (record-case thread
    (done (value) (make-done value))
    (doing (thunk)
      (make-doing (∘ ((lambda () (extend unit))) thunk)))))
```

But on the right we have:

```
((lambda () id))
```

This of course converges to `id`, which is equivalent to:

```
(lambda (thread)
  (record-case thread
    (done (value) (make-done value))
    (doing (thunk) (make-doing (∘ ((lambda () id)) thunk)))))
```

These differ only by the two abstractions in R, and are thus related by R^*_{val}, satisfying the constraint on abstractions.

Each of these terms is an application of a common context $C[]$:

[9] In the proofs of successive architectures, these terms are not necessarily `lambda` abstractions or contstants; one may either infer an expansive notion of values or allow the terms to be related by R^*

```
(lambda (thread)
  (record-case thread
    (done (value) (make-done value))
    (doing (thunk) (make-doing (∘ [] thunk)))))
```

to the two original applications. Many subsequent proofs will have similar structure and we provide this context in lieu of the earlier details.

For the third law, unlike the second, our expressions have free variables that we must bind with our abstractions:

```
(lambda (f g) (extend (∘ (extend f) g)))
```

and

```
(lambda (f g) (∘ (extend f) (extend g)))
```

We postulate pairs of values $v_1^{\mathtt{f}}$, $v_1^{\mathtt{g}}$; $v_2^{\mathtt{f}}$, $v_2^{\mathtt{g}}$, respectively related by R^*_{val}, to which these abstractions are applied in the hope of getting related results. For either pair of values $v^{\mathtt{f}}$ and $v^{\mathtt{g}}$, we have $C_{v^{\mathtt{f}},v^{\mathtt{g}}}[] =$

```
(lambda (thread)
  (record-case thread
    (done (value) ((extend v^f) (v^g value)))
    (doing (thunk) ((bounce (∘ [] thunk))))))
```

Theorem 3.2 *Stepping is a trampolining architecture.*

Proof Along with Theorem 3.1, show (for Law 4):

```
(∘ (extend recvr) (bounce (lambda () comp))) =
(bounce (lambda () ((extend recvr) comp)))
```

and (for Law 5), setting $i = 1$:

```
(∘ trampoline bounce) = trampoline
```

These follow easily from the definitions. We will proceed similarly for each architecture until Section 4.3 for Law 5 and Section 5.2 for Law 4.

3.2 Dynamic Thread Creation and Termination

The trampolining architecture for dynamic thread creation and termination has each expression generated by $\mathcal{E}$ evaluate to a list of threads. The list represents the threads engendered by the current execution (including the remainder or result of the current thread) and is appended to the thread queue. Thus, we define the type constructor M, representing computations, as $\mathsf{M}\rho\alpha = \mathsf{List}\ \mathsf{T}\rho\alpha$. T is still defined such that $\mathsf{T}\rho\alpha = \alpha + (1 \to \mathsf{M}\rho\alpha)$. Our notion of computation thus involves the type operators of the list and resumption monads [21,24]. By returning a list with more than one element, expressions can spawn new threads. It is also convenient to terminate your own thread—just return an empty list. Under this notion of multitasking there is no mechanism for a child's value to be returned directly to the parent. It is, however, possible for threads to communicate through shared variables. This protocol otherwise completely protects

```
(define-record-type done (fields (immutable value)))
(define-record-type doing (fields (immutable thunk)))

(define unit (∘ unit_List make-done))

(define bounce (lambda (thunk) (lambda () (unit_List (make-doing thunk)))))

(define extend
  (lambda (recvr)
    (extend_List
      (lambda (thread)
        (record-case thread
          (done (value) (recvr value))
          (doing (thunk) ((bounce (∘ (extend recvr) thunk)))))))))

(define tramp
  (lambda (tq)
    (if (pair? tq)
      (record-case (car tq)
        (done (value) (stream-cons value (tramp (cdr tq))))
        (doing (thunk) (tramp (append (cdr tq) (thunk)))))
      (make-empty-stream))))

(define trampoline (∘ tramp list make-doing))
```

Fig. 6 Dynamic Threads Architecture

threads from each other; no entries in the thread queue other than the one currently running can be directly affected.

The Dynamic Threads Architecture is presented in Figure 6. `unit` and `bounce` now return the thread as the sole element of a list. They differ from the Stepping Architecture only in that the injection is followed immediately by the List monad's `unit`.[10]

We now define `extend` so as to ensure that `recvr` is performed after every result produced by an initial computation. We use the List monad's `extend` operator[11] to collect the results. When applied to a function that postpends `recvr` onto a thread, it yields a function over a list of threads that can serve to receive the result of applying a suspended computation. From each thread in the list we obtain another list of threads; these are appended together by $\texttt{extend}_{\mathsf{List}}$.

The scheduler `trampoline` must now allocate control among all of the threads in process, any number of which may emit a result. The type variable α in the type of trampoline now signifies a union over all of the result types that might be produced by the computation. The helper procedure `tramp` now handles a thread queue and is thus of type $(\forall\rho.\mathsf{List}\ (\mathsf{T}\rho\alpha)) \rightarrow \mathsf{Stream}\ \alpha$. It applies a round-robin scheduling algorithm. `tramp` examines the first thread on the queue and if its computation has completed, it extends the stream of results; otherwise it takes a step and places any resulting threads at the end of the queue before proceeding. In contrast to the Stepping Architecture, on emitting a value, it continues processing any remaining threads but terminates upon an empty thread queue. Since our operators return lists of threads, `tramp` can directly `append` the result of applying `thunk`.

[10] `list`, in Scheme

[11] curried mapcan, in Lisp

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

(define spawn
  (lambda (thunk1 thunk2) (list (make-doing thunk1) (make-doing thunk2))))
```

Fig. 7 Dynamic Threads Operators

At this point, though, `tramp` always obtains a singleton list of threads. Since the initial thread queue contains only one element, `trampoline` simply invokes that element, yielding another singleton thread queue. Yet, 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 Figure 7. In `die`, we simply return the empty list, the identity for the operation `append`. Thus, nothing is added to the list of threads, and so the current computation terminates. It is of type $1 \to \mathsf{M}\rho\alpha$.

Notice that `spawn` delays each thunk as if `bounce` had been called. This will be convenient in Section 4.3, but has implications on the translation. Although we could simply translate `spawn` in place, it would be more expedient to avoid generating additional `bounce` forms around procedure calls that serve as its operands. Next, we extend the protocol to enable new threads to be added to the queue. To fork the computation, we may apply the `spawn` operator to two thunks. Each tail-position expression now evaluates to an arbitrarily long list of threads. The procedure `spawn` simply packages the thunks as a list of doing records. Its type is $((1 \to \mathsf{M}\rho\alpha_1) \times (1 \to \mathsf{M}\rho\alpha_2)) \to \mathsf{M}\rho(\alpha_1 \cup \alpha_2)$.

We demonstrate the operators by running an endless factorial computation together with a terminating one.

```
> (trampoline
    (lambda ()
      (spawn (lambda () (fact-tramp -1))
             (lambda () (fact-tramp 5)))))
[120 ***]
```

Although the first computation does not terminate, the second does, which causes `trampoline` to generate a result. This is made possible by the replacement of the separate loops in each `fact-tramp` computation by a single loop in `trampoline`.

The Fibonacci function provides an example of communication between threads by way of shared state in the form of an accumulator. The use of `spawn` here forks the computation between the two recursive calls. The algorithm relies on the equality betwen the value of the Fibonacci function and the number of times a base case is reached in the standard recursive definition (provable by a simple inductive argument). We `spawn` a new thread with each recursive call. At the base case, the accumulator is incremented and the thread terminated. This interaction between threads is asynchronous in the sense that we can not rely on the order in which the various threads will update the accumulator, only on the fact that they will all eventually do so. The stretches between calls to `spawn` are guaranteed to be atomic. Any desired synchronization beyond that must be provided by the programmer.

```
> (define fib-asynch
    (lambda (n)
      (if (< n 2)
        (begin
          (set! acc (add1 acc))
          (die))
        (spawn (lambda () (fib-asynch (- n 1)))
               (lambda () (fib-asynch (- n 2)))))))
> (define acc 0)
> (trampoline (lambda () (fib-asynch 10)))
[]
> acc
89
```

We would prefer, however, to be able to run multiple instances of `fib-asynch` on `trampoline` and retrieve separate results. That is clearly not possible as long as the accumulator is global. To surmount this limitation, we arrange that a separate accumulator be allocated for each instance of `fib-asynch`.

```
> (define instantiate-fib-asynch
    (lambda ()
      (let ((acc 0))
        (cons
          (letrec ((fib-asynch
                     (lambda (n)
                       (cond
                         ((< n 2)
                          (set! acc (add1 acc))
                          (die))
                         (else
                          (spawn (lambda () (fib-asynch (- n 1)))
                                 (lambda () (fib-asynch (- n 2)))))))))
            fib-asynch)
          (lambda () acc)))))
> (define fib1 (instantiate-fib-asynch))
> (define fib2 (instantiate-fib-asynch))
> (trampoline (lambda () (spawn (lambda () ((car fib1) 5))
                                (lambda () ((car fib2) 10)))))
[]
> ((cdr fib1))
8
> ((cdr fib2))
89
```

Theorem 3.3 *In Dynamic Threads,* Mρ*,* `unit`*, and* `extend` *form a monad.*

Proof For the first law, the left side again reduces to the right. For the second, the context $C[] =$

```
(List
  (lambda (thread)
    (record-case thread
      (done (value) (make-done value))
      (doing (thunk) (make-doing (∘ [] thunk))))))
```

where List is the functor of the List monad,[12] reflecting the fact that our computations are not threads but lists of threads.

For the third law, we have $C_{v^{\mathtt{f}},v^{\mathtt{g}}}[] =$

```
(extend_List
  (lambda (thread)
    (record-case thread
      (done (value) ((extend v^f) (v^g value)))
      (doing (thunk) ((bounce (∘ [] thunk)))))))
```

Theorem 3.4 *Dynamic Threads is a trampolining architecture.*

3.3 A More Fair Architecture

The above architecture is fair in the sense of Ramanujam and Lodaya [27] that every thread that is not done is eventually allowed to continue processing. However, this is of little comfort to a process that finds its share of processor time diminishing due to being run alongside another process that continually spawns siblings. If we were to run `(fact-tramp 9)` in parallel with `(fib-asynch 10)` in the above example, the factorial computation would terminate after the Fibonacci computation had completed, although the latter requires far more steps of processing. At issue is the round-robin scheduling algorithm and the organization of threads as a simple queue.

The Fair Threads Architecture in Figure 8 allows for a "per stirpes" notion of fairness wherein if a process spawns siblings, these must live off of its own resources and not those of any existing threads. The motivation is that a thread should not gain a greater share of processing time by spawning more threads. Our notion of fairness is an extension of that of Nagle, who presented an algorithm in the context of packet-switching [23].

Computations in this architecture correspond to trees of computations under the Stepping Architecture. This is accomplished by adding a new variant to the type constructor T, representing threads, so that $\mathsf{T}\rho\alpha = \alpha + (1 \to \mathsf{M}\rho\alpha) + \mathsf{List}\ \mathsf{T}\rho\alpha$. The new record type `multi` corresponds to a thread that has been split into siblings, represented as a list of thread records.

Because lists of siblings are now internalized in threads, we redefine the type constructor M, representing computations, as simply $\mathsf{M}\rho\alpha = \mathsf{T}\rho\alpha$, as in the Stepping architecture. `unit` and `bounce` also revert to their definitions under the earlier architecture.

[12] curried map, in Scheme

```
(define-record-type done (fields (immutable value)))
(define-record-type doing (fields (immutable thunk)))
(define-record-type multi (fields (immutable siblings)))

(define unit make-done)

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

(define extend
  (lambda (recvr)
    (Y (lambda (p)
         (lambda (thread)
           (record-case thread
             (done (value) (recvr value))
             (doing (thunk) ((bounce (∘ (extend recvr) thunk))))
             (multi (siblings) (make-multi ((List p) siblings)))))))))

(define tramp
  (lambda (thread)
    (tramp-step thread
      (lambda (res . value?)
        (let ((get-rest
               (if (and (multi? res) (null? (multi-siblings res)))
                 make-empty-stream
                 (lambda () (tramp res)))))
          (if (null? value?)
            (get-rest)
            (stream-cons (car value?) (get-rest))))))))

(define trampoline (∘ tramp make-doing))

(define tramp-step
  (lambda (thread k)
    (record-case thread
      (done (value) (k (make-multi '()) value))
      (doing (thunk) (k res))
      (multi (siblings)
       (if (null? siblings)
         (k (make-multi '()))
         (tramp-step (car siblings)
           (lambda (res . value?)
             (apply k
               (make-multi
                 (if (and (multi? res) (null? (multi-siblings res)))
                   (cdr siblings)
                   (append (cdr siblings) (list res))))
               value?)))))))))
```

Fig. 8 Fair Threads Architecture

Our `extend` routine must now recursively append the `recvr` computation to any nested threads. It does so using a `Y` combinator.[13,14,15] That combinator is used to

[13]We use several such combinators. Smullyan provides a good introduction [32] to combinatory logic [30].

[14]The use of the applicative–order `Y` combinator:

```
(lambda (f) ((lambda (x) (f (x x))) (lambda (x) (f (x x)))))
```

```
(define die (lambda () (make-multi '())))

(define spawn
  (lambda (thunk1 thunk2 . thunkRest)
    (make-multi (apply list (make-doing thunk1) (make-doing thunk2)
                  (map make-doing thunkRest)))))
```

Fig. 9 Fair Threads Operators

define `p` as a recursive "thread extender" that operates not just on a single thread but upon the entire thread tree. In the case of a `multi` thread, we must recur on the siblings. The recursive thread extender also serves to receive the result of applying a suspended computation.

`tramp` now operates over a thread queue in the form of a tree. A helper procedure `tramp-step` performs a step of processing for a single thread. If this involves descending into the tree, all ancestor thread queues are considered to have been processed and must be rotated. That rotation occurs in a continuation argument `k`, which accepts the thread that was just processed, and any value that it might have produced. Continuations are called directly when the caller knows the number of arguments or via `apply` when it is passing along what it was given. It will be useful later to ensure that the `done` thread that led to the emitted value not be included in the result thread. Results generated from recursive calls to siblings are propagated upwards. It is only the final continuation that collects results in a stream.

This modification, with our current definitions and assuming n threads and a maximum branching factor of b, reduces the time complexity of each scheduler step from $O(n)$ to $O(b \log_b n)$, assuming balanced trees. If we optimized out the queue rotation (perhaps by using circular mutable queues), the time complexity for balanced trees would drop to $O(\log_b n)$, but in the worst case would return to $O(n/b)$. However, if we amortize this cost over a full cycle of the scheduler with the same threads, it would approach a constant $(b/(b-1))$ for large n. In effect, diving into the thread tree is not very expensive because the deeper a thread appears in it, the less often it needs to be executed.

`die` and `spawn` in Figure 9 now return `multi` records. We gain expressivity by allowing `spawn` to take additional operands, as processor attention will be divided equally among them. `spawn` now wraps its arguments in `list` to create the nested thread queue.

We now see that in a situation similar to the one referenced above, the factorial computation terminates at a point where the Fibonacci computation has incremented the accumulator only once.

```
> (define acc 0)
> (stream-car
    (trampoline (lambda () (spawn (lambda () (fact-tramp 50))
                                  (lambda () (fib-asynch 10))))))
30414093201713378043612608166064768844377641568960512000000000000
```

would not indicate an efficient implementation, but we always apply it to a nested abstraction so that it is easily replaced with `letrec`.

[15] Explicit recursion is not necessary here, as we could replace the reference to `p` with `(extend recvr)`. But it is useful for the proofs and sets us up for the next architecture.

```
> acc
1
```

Theorem 3.5 *In Fair Threads,* Mρ*,* `unit`*, and* `extend` *form a monad.*

Proof For the first law, the left side again reduces to the right.

For the second law, the context $C[] =$

```
(Y (lambda (p)
     (lambda (thread)
       (record-case thread
         (done (value) (make-done value))
         (doing (thunk) (make-doing (∘ [] thunk)))
         (multi (siblings) (make-multi ((List p) siblings)))))))
```

It reflects the new definition of `extend`, including the clause for the new variant.

For the third law, the context $C_{v^{\mathtt{f}},v^{\mathtt{g}}}[] =$

```
(Y (lambda (p)
     (lambda (thread)
       (record-case thread
         (done (value) ((extend v^f) (v^g value)))
         (doing (thunk) (make-doing (∘ [] thunk)))
         (multi (siblings) (make-multi ((List p) siblings)))))))
```

Theorem 3.6 *Fair Threads is a trampolining architecture.*

4 Quasi-Monadic Trampolining Architectures

4.1 Retracing Our Steps

So far, all seems to be going well. We have described several trampolining architectures of increasing levels of sophistication, all conforming to the monad laws. But let's look a little more deeply.

Recall the definition of `extend` from the Stepping Architecture. The `done` case immediately applied the `recvr` function to the `value` of the `done` thread. Had we instead defined it so as to create an intermediate `doing` thread that would perform that application, i.e., `(make-doing (lambda () (recvr value)))`, we would obtain the first procedure in Figure 10. We could also replace the `bounce` operator in the `doing` case by `make-doing`, to emphasize that each thread is replaced by another. The third procedure in Figure 10 results from making similar changes to the `extend` of the Fair Threads Architecture.

For the Dynamic Threads Architecture, computations (including those engendered by `extend`) are represented as lists of threads. Instead of allowing each thread in the initial computation to generate a list of threads and then flattening, we might have simply replaced each thread in the initial computation with one that knows to perform the `recvr` function. We would thus apply the List functor, rather than $\mathtt{extend}_{\mathsf{List}}$. The changes to the two record cases are the same as for the other two architectures, but the new code describes a single thread where the old described a list of threads.

Stepping:
```
(define extend
  (lambda (recvr)
    (lambda (comp)
      (record-case comp
        (done (value)
          (make-doing (lambda () (recvr value))))
        (doing (thunk)
          (make-doing (∘ (extend recvr) thunk)))))))
```

Dynamic Threads:
```
(define extend
  (lambda (recvr)
    (List
      (lambda (thread)
        (record-case thread
          (done (value) (make-doing (lambda () (recvr value))))
          (doing (thunk)
            (make-doing (∘ (extend recvr) thunk))))))))
```

Fair Threads:
```
(define extend
  (lambda (recvr)
    (Y (lambda (p)
         (lambda (thread)
           (record-case thread
             (done (value)
               (make-doing (lambda () (recvr value))))
             (doing (thunk)
               (make-doing (∘ (extend recvr) thunk)))
             (multi (siblings)
               (make-multi ((List p) siblings)))))))))
```

Fig. 10 Alternate `extend` Definitions

These changes seem harmless enough. But consider how they impact our proofs. We demonstrate the first monad law for the Stepping Architecture. The proof of this law did not involve reasoning by coinduction. We start with:

```
(∘ (extend recvr) unit)
```

Plugging in the definitions of our monadic operators yields:

```
(∘ (lambda (thread)
     (record-case thread
       [done (value)
        (make-doing (lambda () (recvr value)))]
       [doing (thunk)
        (make-doing (∘ (extend recvr) thunk))]))
 make-done)
```

Because we know that by the time we reach the `record-case` we will have a `done` thread, we can ignore the `doing` case:

```
(∘ (lambda (thread)
     (record-case thread
       [done (value)
        (make-doing (lambda () (recvr value)))]))
 make-done)
```

Simplifying:

```
(lambda (value)
  (make-doing (lambda () (recvr value))))
```

This is purported to be equivalent to *recvr*, but in fact it is not. If, for example, *recvr* were `unit`, the two expressions would evaluate to different kinds of threads. But these threads would differ, in effect, only by a single step of `trampoline`.

What does this say about our trampolining architectures? Similarly to the joke about the patient who complains that his arm hurts whenever he "does *this*", moving it into some contorted position, a "doctor" might respond, "So don't do *that*". But there are several differences.

First, the alternate versions of `extend` are not contorted. In fact, for the Dynamic Threads Architecture, the alternate version is probably more intuitive. One could argue that it is the original version that was contorted in order to satisfy the monad laws.

Second, we will see in the next section that something corresponding to the alternate versions of `extend` will provide opportunities that are not available with the approach of the original version.

The alternate versions of `extend` in Figure 10 seem to work as well as the originals. Do we want to declare that they are wrong, just because they do not fit within the confines of our model? What other trampolining architectures might we overlook by doing so?

4.2 Recycling Thread Records

It is inefficient to continually allocate and deallocate thread records. If we make the current thread available, the computation can reuse it. This is a simple example of the technique that Sobel and Friedman applied to continuations [33]. A system that recycled thread records in this way would still fit our intuitive notion of trampolining. In this section, we see that it fits our formal definition as well, at least after loosening the monadic requirement.

For this section, we make use of the record system's ability to update record fields. We merge our three record types `done`, `doing`, and `multi` into a single record type `thread`. In lieu of using the implicit record tags, we can identify the kind of thread by an explicit tag. We introduce the symbols `done`, `doing`, and `multi` for this purpose. Our threads now must hold a union of all of the fields of any variant.[16] Our thread type constructor T is defined such that $\mathsf{T}\rho\alpha \cong \mathsf{Ref}\ (\alpha + (1 \to \mathsf{M}\rho\alpha) + \mathsf{List}\ \mathsf{T}\rho\alpha)$, i.e., it is isomorphic to a reference type of its type in the Fair Threads Architecture, relative to the definition of computations. To mitigate the readability issues that this causes, we define the familiar predicates and accessors. Record abstractions and other code that will be used for several of our architectures are presented in Figure 11.

[16] It is possible to make do with a more compact representation that avoids a dedicated tag field.

The rest of the Thread Recycling Architecture is in Figure 12, beginning with the `thread` record type definition and the `doing` thread constructor—the only one that we will need. The computations engendered by our operators are now represented by a procedure, rather than by a thread record directly. The procedure expects a thread record, representing the current computation, and through its references, mutates it to take a single step. The computation type constructor M is defined as $\mathsf{M}\rho\alpha = \mathsf{T}\rho\alpha{\to}1$. Thus, as a computation proceeds, the thread record can be preserved. We are, however, still allocating and deallocating thunks but address this in Section 5.1. Our operators now build computations that receive the current thread and mutate it.

The `unit` procedure now returns a value by converting the current thread to a `done` thread and placing the value to be returned in the `value` field. The `bounce` procedure likewise converts the thread to `doing` and updates the `thunk`. Resetting the thread type is not necessary for correct operation but will be important in obtaining our desired result for this architecture.

We modify `extend` considerably. The result of applying the `B!` combinator[17] to a recursive "thread extender" expects the initial computation (in the form of a procedure expecting a thread) and then the current thread. It applies the former to the latter, and then the thread extender to the result. It is precisely what is needed to receive the result of a suspended computation. Our result computation, when provided with the current thread, will thus process it in place to take a single step, converting it to a `doing` thread that performs `recvr` when `done`.

Observe that we could not use the original approach of Section 3 here. Because each case of the thread extender is performed only for effect, were we to directly apply `recvr` to `(done-value thread)` in the `done` case, the resulting computation would never be executed.

The `trampoline` operator constructs the initial thread record. `tramp` differs only in using the new `thread` representation. `tramp-step` must now modify `thread` in place.

Consider the threads with which our computations are invoked. We invoke computations initially in `tramp-step`, where a `doing` thread is provided. The only other place where we invoke a computation is in `B!`, where there are two occurrences. For the first occurrence, the `doing` thread from `tramp-step` will be used. However, for the second occurrence, this may no longer be a `doing` thread, as it may have been mutated by the first computation in the sequence. The second computation in the sequence may receive a thread of any type, but this computation will always be the recursive thread-extender in `extend`. `extend` is thus unique among our computations in that it should expect to receive arbitrary threads and not just `doing` threads.

In Figure 13 we redefine the dynamic thread creation operator for the new architecture. `spawn` now converts the current thread from `doing` to `multi` and explicitly creates a separate sibling `thread` record for each new operand. The termination operator is defined in Figure 11.

The tests of previous sections continue to run. They should perform far fewer allocations on an appropriate implementation.

Theorem 4.1 *In Thread Recycling,* $\mathsf{M}\rho$*,* `unit`*, and* `extend` *form a monad, subject to the equality* `bounce` = `id`*.*

[17] We defined the `B!` combinator in Figure 11 as an imperative version of the `B` combinator, which in turn is a curried ∘ that can be expressed as `(lambda (f) (lambda (g) (lambda (x) (f (g x)))))`.

```
(define done (gensym ``done''))
(define doing (gensym ``doing''))
(define multi (gensym ``multi''))

(define done? (lambda (thread) (eq? (thread-tag thread) done)))
(define done-value thread-value)
(define done-value-set! thread-value-set!)
(define to-done! (lambda (thread) (thread-tag-set! thread done)))

(define doing? (lambda (thread) (eq? (thread-tag thread) doing)))
(define doing-proc thread-proc)
(define doing-proc-set! thread-proc-set!)
(define to-doing! (lambda (thread) (thread-tag-set! thread doing)))

(define multi? (lambda (thread) (eq? (thread-tag thread) multi)))
(define multi-sibs thread-sibs)
(define multi-sibs-set! thread-sibs-set!)
(define to-multi! (lambda (thread) (thread-tag-set! thread multi)))

(define unit
  (lambda (value)
    (lambda (thread)
      (to-done! thread)
      (done-value-set! thread value))))

(define B!
  (lambda (comp2)
    (lambda (comp1)
      (lambda (thread)
        (comp1 thread)
        (comp2 thread)))))

(define tramp
  (lambda (thread)
    (tramp-step thread
      (lambda (res . value?)
        (let ((get-rest
               (if (and (multi? res) (null? (multi-sibs res)))
                 make-empty-stream
                 (lambda () (tramp res)))))
          (if (null? value?)
            (get-rest)
            (stream-cons (car value?) (get-rest)))))))))

(define die
  (lambda ()
    (lambda (thread)
      (to-multi! thread)
      (multi-sibs-set! thread '()))))
```

Fig. 11 Common Code for Thread Recycling, Communicating Threads, and Closure Avoidance Architectures

```
(define-record-type thread
  (fields (mutable tag) (mutable value) (mutable proc) (mutable sibs)))
(define make-doing (lambda (thunk) (make-thread doing (void) thunk (void))))

(define bounce
  (lambda (thunk)
    (lambda ()
      (lambda (thread)
        (to-doing! thread)
        (doing-proc-set! thread thunk)))))

(define extend
  (lambda (recvr)
    (B! (Y (lambda (p!)
             (lambda (thread)
               (cond
                 ((done? thread)
                  (let ((value (done-value thread)))
                    (to-doing! thread)
                    (doing-proc-set! thread
                      (lambda () (recvr value)))))
                 ((doing? thread)
                  (doing-proc-set! thread
                    (∘ (extend recvr) (doing-proc thread))))
                 ((multi? thread) (for-each p! (multi-sibs thread)))))))))

(define trampoline (∘ tramp make-doing))

(define tramp-step
  (lambda (thread k)
    (cond
      ((done? thread)
       (let ((value (done-value thread)))
         (to-multi! thread)
         (multi-sibs-set! thread '())
         (k thread value)))
      ((doing? thread)
       (((doing-proc thread)) thread)
       (k thread))
      ((multi? thread)
       (if (null? (multi-sibs thread))
         (k thread)
         (tramp-step (car (multi-sibs thread))
           (lambda (res . value?)
             (multi-sibs-set! thread
               (if (and (multi? res) (null? (multi-sibs res)))
                 (cdr (multi-sibs thread))
                 (append (cdr (multi-sibs thread)) (list res))))
             (apply k thread value?))))))))
```

Fig. 12 Thread Recycling Architecture

Proof We use the absence of the tags from our set of values. We also require that invariants regarding the tags hold over threads at computation boundaries: that if `(doing?` *thread*`)`, then `(doing-proc` *thread*`)` is a thunk and that if `(multi?` *thread*`)`, then `(multi-sibs` *thread*`)` is a list of threads. For the first monad law, the left side reduces to

```
(define spawn
  (lambda (thunk1 thunk2 . thunkRest)
    (lambda (thread)
      (to-multi! thread)
      (multi-sibs-set! thread
        (apply list (make-doing thunk1) (make-doing thunk2)
          (map make-doing thunkRest))))))
```

Fig. 13 Thread Recycling Operator

```
(lambda (value) ((bounce (lambda () (recvr value)))))
```

Applying the equality `((bounce` *thunk*`))` = (*thunk*) and η reduction yields `recvr`.

For the second monad law, reduction of the left side uses the equality. We have $C[] =$

```
(B! (Y (lambda (p!)
         (lambda (thread)
           (cond
             ((doing? thread)
              (doing-proc-set! thread (∘ [] (doing-proc thread))))
             ((multi? thread) (for-each p! (multi-sibs thread))))))))
```

The `bounce` equality is not needed for the third law. We have $C_{v^{\mathtt{f}},v^{\mathtt{g}}}[] =$

```
(B! (Y (lambda (p!)
         (lambda (thread)
           (cond
             ((done? thread)
              (let ((value (done-value thread)))
                (to-doing! thread)
                (doing-proc-set! thread
                  (lambda () ((extend v^f) (v^g value))))))
             ((doing? thread)
              (doing-proc-set! thread (∘ [] (doing-proc thread))))
             ((multi? thread) (for-each p! (multi-sibs thread))))))))
```

Theorem 4.2 *Thread Recycling is a trampolining architecture.*

4.3 Synchronous Communication

Here we build on our previous architectures to implement more complex interaction between threads. We intend that a thread can start a computation that will run concurrently with itself, and at any point request values from that computation. Moreover, any thread, and only threads, spawned (perhaps indirectly) by that computation may contribute results, and results may be requested any number of times. This architecture is fundamentally different from those that preceded it in that it introduces an asymmetric relationship between running threads—one is considered to be the parent and the other the child.

The usage unfolds as follows. A parent may start a child computation. Both will begin running. Any values produced by the child will be held. The parent may *touch*

the child, i.e., request the result of the child's computation. If the child has produced a waiting value, the parent receives the result. Otherwise, the parent is put to sleep until a result is available, at which point the parent is awakened. When the parent terminates, all its descendents (but not siblings via `spawn`) terminate with it. This notion of inter-thread communication is related to that of futures [11]. Futures were independently applied to a system with trampolining by Manolescu [19]; perhaps this is evidence that the two techniques are a good fit. The communication is functional in that it does not rely on explicit side effects in user code, as did the threads presented earlier (although such side effects are still available). However, our futures are general in that the child thread may return multiple values.

Our nested queue representation from Section 3.3 enables a thread to have access to all possible results that might be produced by a child (via spawned threads). We must, however, allow for threads to wait on a value from any particular child. The updating of thread records in-place from Section 4.2 then allows us, with a single pointer comparison, to recognize a thread producing a value for a waiting parent. A disadvantage of this approach is that touching of a child thread by any thread other than its parent is not well-defined.

Values are still propagated upward implicitly through the siblings hierarchy. The upward propagation from children must be explicitly directed by the parent. We arbitrarily decide that a parent with any children splits time 50/50 between itself and the children, collectively. `trampoline` divides time evenly among the children.

The Communicating Threads Architecture is presented in Figures 11, 14, and 15. We redefine the type constructor T, representing threads, such that it is now isomorphic to a reference to the product of a boolean toggle and a sum over four variants. The first, `done` threads, and third, `multi` threads, are the same as in the Recycling Threads architecture. `doing` threads are modified by inclusion of a list of children. The fourth variant is `waiting` threads, representing threads that can no longer continue without a result from their child. Each of these includes the list of children as well as the designated child thread on which it is waiting. The `doing` thread's thunk is replaced by a procedure of one argument with which to awaken the `waiting` thread upon a result from the child. The boolean toggle is intended to regulate the alternation of computation between a thread and its children: when true the scheduler will process the children and when false the parent.

$$
\begin{array}{l}
\mathsf{T}\rho\alpha \cong \\
\mathsf{Ref}(\ 2 \times \alpha + \\
\qquad (1 \to \mathsf{M}\rho\alpha) \times \mathsf{List}\ \exists\alpha.\mathsf{T}\rho\alpha + \\
\qquad \mathsf{List}\ \mathsf{T}\rho\alpha + \\
\qquad \exists\alpha_1 (\alpha_1 \to \mathsf{M}\rho\alpha) \times \mathsf{List}\ \exists\alpha.\mathsf{T}\rho\alpha \times \mathsf{T}\rho\alpha_1)
\end{array}
$$

The full type expresses that there is no necessary relation between the parent and child types and that the awakening procedure mediates between them. It does not express that the thread being waited on of type $\mathsf{T}\rho\alpha_1$ must belong to the list of children of type $\mathsf{List}\ \exists\alpha.\mathsf{T}\rho\alpha$.

The type constructor M, representing computations, continues to be defined as $\mathsf{M}\rho\alpha = \mathsf{T}\rho\alpha \to 1$.

`extend` must now handle the `waiting` case by appending the receiving procedure `recvr` to the awakening procedure `(waiting-proc thread)`.

The helper procedure `tramp-step` must now handle the case of a value being emitted from a child whose parent is waiting on it by awakening the parent. The toggle

```
(define-record-type thread
  (fields (mutable tag) (mutable value) (mutable proc)
    (mutable sibs) (mutable toggle)
    (mutable children) (mutable waiting-on)))
(define make-doing (lambda (thunk children)
                     (make-thread doing (void) thunk (void) #t children (void))))
(define doing-children thread-children)
(define doing-children-set! thread-children-set!)

(define waiting (gensym "waiting"))
(define waiting? (lambda (thread) (eq? (thread-tag thread) waiting)))
(define waiting-proc thread-proc)
(define waiting-proc-set! thread-proc-set!)
(define waiting-waiting-on thread-waiting-on)
(define waiting-waiting-on-set! thread-waiting-on-set!)
(define to-waiting! (lambda (thread) (thread-tag-set! thread waiting)))

(define bounce
  (lambda (thunk)
    (lambda ()
      (lambda (thread)
        (to-doing! thread)
        (doing-proc-set! thread thunk)))))

(define extend
  (lambda (recvr)
    (B! (Y (lambda (p!)
             (lambda (thread)
               (cond
                 ((done? thread)
                  (let ((value (done-value thread)))
                    (to-doing! thread)
                    (doing-proc-set! thread (lambda () (recvr value)))))
                 ((doing? thread)
                  (doing-proc-set! thread
                    (∘ (extend recvr) (doing-proc thread))))
                 ((multi? thread) (for-each p! (multi-sibs thread)))
                 ((waiting? thread)
                  (waiting-proc-set! thread
                    (∘ (extend recvr) (waiting-proc thread))))))))))

(define trampoline (lambda (thunk) (tramp (make-doing thunk '()))))
```

Fig. 14 Communicating Threads Architecture, I

is used to alternate computation between children and parent. In the latter case, processing follows the model of Section 4.2, but must now handle the cases of `done` and `waiting` threads. `done` threads must be handled because they could now be left in the queue pending a touch.

In the former case, it is after a recursive call on a child that we must accept a result from the thread that we are waiting on, set ourselves to no longer be `waiting`, and pass the value to the stored procedure. This takes place in a continuation built to receive the result of the recursive call on the first child. That continuation is also responsible for rotating the children queue. The `done` thread that led to the emitted value not being included in the result thread ensures that each produced value will only be consumed once. The ultimate consumer is the stream of observable results. A `seek-value?` flag

```
(define tramp-step
  (lambda (thread k)
    (letrec
      ((ts (lambda (thread seek-value? k)
             (thread-toggle-set! thread (not (thread-toggle thread)))
             (if (and (or (doing? thread) (waiting? thread))
                      (thread-toggle thread)
                      (not (null? (thread-children thread))))
               (tramp-children thread k ts)
               (cond
                 ((done? thread)
                  (if seek-value?
                    (let ((value (done-value thread)))
                      (to-multi! thread)
                      (multi-sibs-set! thread '())
                      (k thread value))
                    (k thread)))
                 ((doing? thread)
                  (((doing-proc thread)) thread)
                  (if (and seek-value? (done? thread))
                    (let ((value (done-value thread)))
                      (to-multi! thread)
                      (multi-sibs-set! thread '())
                      (k thread value))
                    (k thread)))
                 ((multi? thread)
                  (if (null? (multi-sibs thread))
                    (k thread)
                    (ts (car (multi-sibs thread)) seek-value?
                      (lambda (res . val?)
                        (multi-sibs-set! thread
                          (if (and (multi? res) (null? (multi-sibs res)))
                            (cdr (multi-sibs thread))
                            (append (cdr (multi-sibs thread)) (list res))))
                        (apply k thread val?)))))
                 ((waiting? thread) (k thread)))))))
      (ts thread #t k))))

(define tramp-children
  (lambda (thread k ts)
    (ts (car (thread-children thread))
      (and (waiting? thread)
           (eq? (waiting-waiting-on thread) (car (thread-children thread))))
      (lambda (res . val?)
        (let ((rest (cdr (thread-children thread))))
          (thread-children-set! thread
            (if (and (multi? res) (null? (multi-sibs res)))
              rest
              (append rest (list res))))
          (if (null? val?)
            (k thread)
            (let ((proc (waiting-proc thread)))
              (to-doing! thread)
              (doing-proc-set! thread (lambda () (proc (car value?))))
              (k thread)))))))))
```

Fig. 15 Communicating Threads Architecture, II

```
(define spawn
  (lambda (thunk1 thunk2 . thunkRest)
    (lambda (thread)
      (let ((children (doing-children thread)))
        (to-multi! thread)
        (multi-sibs-set! thread
          (apply list (make-doing thunk1 children) (make-doing thunk2 children)
            (map (lambda (thunk) (make-doing thunk children)) thunkRest)))))))

(define-record-type just (fields (immutable value)))
(define-record-type none (fields))

(rewrites-as (f-let (?touch-var ?rhs) ?body)
  (lambda (thread)
    (let ((child-thread (make-doing (lambda () ?rhs) '())))
      (doing-children-set! thread
        (cons child-thread (doing-children thread)))
      (let ((?touch-var (touch child-thread)))
        (?body thread)))))

(define extract-val!
  (lambda (sibs)
    (let ((rest (cdr sibs)))
      (if (null? rest)
        (make-none)
        (let ((next (car rest)))
          (cond
            ((done? next) (let ((value (done-value next)))
                            (set-cdr! sibs (cdr rest))
                            (make-just value)))
            ((doing? next) (extract-val! rest))
      ((multi? next)
       (if (done? (car (multi-sibs next)))
         (begin
           (multi-sibs-set! next (cdr (multi-sibs next)))
           (make-just (done-value (car (multi-sibs next)))))
         (let ((extract-res (extract-val! (multi-sibs next))))
           (record-case extract-res
             (none () (extract-val! rest))
             (just (value) extract-res)))))))))))
```

Fig. 16 Communicating Threads Operators, I

is used to identify when the value of a `done` thread should be propagated upward; it is set by default, passed unchanged to siblings, and set upon recurring on a child only if the parent is waiting for that child's value. Because of the latter condition, if we obtain a value from recurring on a child, `thread` must have been `waiting`.

Operators for the Communicating Threads Architecture are in Figures 16 and 17. `trampoline` (back in Figure 14) and `spawn` are changed to call the thread constructor with additional arguments. `spawn` must also now reference the children from each sibling, rather than the `doing`/`multi` thread. That is facilitated by the insertion of a `doing` record for each sibling.

We also need an enhanced version of the `spawn` operator, which we call `f-let`. It is defined as a macro. `f-let` treats one argument, the `?body` computation, as a parent and the other, `?rhs`, as a child. `?touch-var` is bound over `?body` to a procedure that retrieves a result from the right-hand-side computation. Like `spawn`, `f-let` must create

```
(define touch
  (lambda (touched-thread)
    (lambda ()
      (lambda (touching-thread)
        (cond
          ((done? touched-thread)
           (to-done! touching-thread)
           (done-value-set! touching-thread (done-value touched-thread)))
          ((multi? touched-thread)
           (if (not (null? (multi-sibs touched-thread)))
             (if (done? (car (multi-sibs touched-thread)))
               (begin
                 (to-done! touching-thread)
                 (done-value-set! touching-thread
                   (done-value (car (multi-sibs touched-thread))))
                 (multi-sibs-set! touched-thread
                   (cdr (multi-sibs touched-thread))))
               (let ((val? (extract-val! (multi-sibs touched-thread))))
                 (record-case val?
                   (none ()
                    (to-waiting! touching-thread)
                    (waiting-waiting-on-set! touching-thread touched-thread)
                    (waiting-proc-set! touching-thread unit))
                   (just (value)
                    (to-done! touching-thread)
                    (done-value-set! touching-thread value)))))))
          ((or (doing? touched-thread) (waiting? touched-thread))
           (to-waiting! touching-thread)
           (waiting-waiting-on-set! touching-thread touched-thread)
           (waiting-proc-set! touching-thread unit)))))))
```

Fig. 17 Communicating Threads Operators, II

a new thread record to prevent overwriting; this must be done for the child. After building the child thread, `f-let` defines `?touch-var` as a thunk and applies the `?body` computation to the current thread. `f-let` expands to a representation of a computation, i.e., an expression of type $\mathsf{M}\rho\alpha$.

The `?touch-var` thunk definition uses `touch`. If the touched thread is already `done`, then the touching thread is also `done` and its value becomes the value of the touched thread. If the touched thread has been broken into siblings, we recursively search for a `done` thread among them. If the search fails, the parent sets itself as `waiting-on` the child and its initial awakening procedure to `unit`. This may later be extended by `extend`. `touch` is of type $\mathsf{T}\rho\alpha \to (1 \to \mathsf{M}\rho\alpha)$.

`extract-val!` is used to extract any value that may have already been produced by the child thread queue. Its argument, `sibs`, is non-empty and its car doesn't contain any value.

If a parent touches a child whose sibling hierarchy becomes empty, the parent will remain waiting indefinitely. It would be only a minor change, however, to provide failure code on the `f-let` binding and use that if the parent touches a child that is (or becomes) guaranteed not to emit any more values.[18]

[18] This would involve storing this failure code on the thread record alongside the awakening procedure that accepts a value.

We demonstrate our new operators with an example: a revised version of the Fibonacci function, making use of interaction between `f-let` and `touch`.

```
(define fib-synch
  (lambda (n)
    (if (< n 2)
      (unit 1)
      (f-let (touch1 (fib-synch (sub1 (sub1 n))))
        (f-let (touch2 (fib-synch (sub1 n)))
          (m-let* ((f1 (touch1))
                   (f2 (touch2)))
            (unit (+ f1 f2))))))))

> (trampoline (lambda () (fib-synch 10)))
[89]
```

An additional example demonstrates the ability of the parent to touch the same child computation twice and obtain successive results.

```
> (trampoline
    (lambda ()
      (f-let (touch (spawn (lambda () (fact 5)) (lambda () (fact 10))))
        (m-let* ((first (touch))
                 (second (touch)))
          (unit (+ first second))))))
[3628920]
```

Theorem 4.3 *In Communicating Threads,* Mρ*,* `unit`*, and* `extend` *form a monad, subject to the equality* `bounce` = `id`.

Proof The proof is similar to that of Theorem 4.1. The main difference is that in the coinductive proofs, there are now two holes in the context instead of one.

For the second monad law, we have $C[] =$

```
(B! (Y (lambda (p!)
         (lambda (thread)
           (cond
             ((doing? thread)
              (doing-proc-set! thread (∘ [] (doing-proc thread))))
             ((multi? thread) (for-each p! (multi-sibs thread)))
             ((waiting? thread)
              (waiting-proc-set! thread
                (∘ [] (waiting-proc thread)))))))))
```

For the third law we have $C_{v^{\mathtt{f}},v^{\mathtt{g}}}[] =$

```
(B! (Y (lambda (p!)
         (lambda (thread)
           (cond
             ((done? thread)
              (let ((value (done-value thread)))
                (to-doing! thread)
                (doing-proc-set! thread
                  (lambda () ((extend v^f) (v^g value))))))
             ((doing? thread)
              (doing-proc-set! thread (∘ [] (doing-proc thread))))
             ((multi? thread) (for-each p! (multi-sibs thread)))
             ((waiting? thread)
              (waiting-proc-set! thread
                (∘ [] (waiting-proc thread)))))))))
```

Theorem 4.4 *Communicating Threads is a trampolining architecture.*

Proof For Law #5, we now must allow two occurrences of `bounce`, reflecting our toggling of evaluation of a thread and its children. It can be shown that:

$$(\circ\ \texttt{trampoline}\ \texttt{bounce}^2) = \texttt{trampoline}$$

5 Reflective Trampolining Architectures

Our previous architectures have fixed $\bar{x} = \epsilon$, i.e., represented a suspended computation as a thunk. In this section, we show how the full generality of the trampolining translation can be used to make the state of a computation available to a user's code. We take as a first example the application of familiar techniques of compilation of functional languages and show that the resulting programs still fall within the purview of our definition of a trampolining architecture. Next, we show a more sophisticated application of the more general translation involving logic programming.

In making the state of a computation available to a user's code, the generalized `bounce` operator provides a form of behavioral reflection. This form of reflection exists in the context of a multi-threaded system, but reifies only the state of a single thread. The `bounce` operator can mediate any difference between the internal representation (stored on a `doing` thread) and the reified representation (our $\bar{v}$). Our `reflect` operator allows source-language terms to be inserted into a thread's implementation.

5.1 Avoiding the Introduction of Closures

We now restate our notion of suspended computation in terms of a reified representation of the machine architecture. We can thus effectively avoid adding closures to programs through our translation. It is no surprise that programs with closures can be represented in a language without them. In this section, we use the modification of programs to avoid free variables as an example of the generalized trampolining translation. As such, multitasking computations can be expressed in terms of machine architecture in

a modular quasi-monadic language. The reflective capability is used to allow user code to be stated in terms of that machine architecture.

Reynolds, through his defunctionalization technique [28], shows that we can replace occurrences of `lambda` expressions with constructions of records holding the values of free variables. Applications of such functions are replaced with code that does a record case analysis, performing the `lambda` body appropriate for the given record type. Where these closures are created and immediately applied, a single instance can be recycled.[19] This is acceptable for closures that are introduced by our trampolining operators (of the form `(lambda (thread) [])`).[20] There are a fixed number of these, and they have a fixed structure, independent of the computations being trampolined. But in the case of thunks introduced by our translation, it would involve a lot of processing of the user's code that wouldn't otherwise be necessary. It should be possible to take a machine-level non-cooperating program (and by extension a non-cooperating program in any language) and trampoline it directly under our translation.

We use the generalized form of our translation to model a stack architecture within transformed programs. Thunks, formerly representing suspended computations, are replaced with a quintuplet including a code-pointer register, a local stack, a frame-pointer register, a stack-pointer register, and a free-values register. Thus we have $\bar{x} =$ `stack`, `fp`, `sp`, `fv`. The code-pointer register `cp` is represented by a procedure accepting the local stack and contents of the other three registers, with no free variables. The frame-pointer register `fp` indicates the base of the topmost frame on the stack, and is used to resolve bound variable references. The stack-pointer register `sp` indicates the next unused stack location. The free-values register `fv` is used to resolve free variable references, due to already existing closures. An augmented transformation would be required for programs originally written with named variable references to make use of this information instead. We describe the necessary changes below.

We build the Closure Avoidance Architecture in Figures 11, 18, and 19 from the code of Section 4.2. The representation of computations remains unchanged from that section except for the structure of thread records.

Suspended computations are now divided into a full machine state and a function expecting that machine state. Thus, `bounce` has access to all state elements and records them in its thread. `bounce` is thus useful in making this context information available to users' code. The stack and free-values register refer to their own state and so do not need to be stored with each `bounce`, but we store them anyway for the sake of the upcoming proofs. The `extend` operator is unchanged, except that our code pointers take four arguments.

The `trampoline` operator accepts a code pointer instead of a thunk. It must now create a `thread` record of the correct format. The new thread is provided with the given code pointer and the other parameters are initialized: a stack vector of pre-set size, zero for the frame and stack pointers, and an empty free-values vector. For `doing` threads, `tramp-step` now simply applies the `cp` instead of the `thunk`.

Operators for the Closure Avoidance Architecture are in Figure 20. `spawn` obtains the stack and register values for new thread records from the local thread record. It

[19] This is similar to our treatment of the thread records in Section 4.2, but here we share a record throughout the system, as opposed to one per thread.

[20] We assume that this can be done without demonstrating it in the code below.

```
(define-record-type thread
  (fields (mutable tag) (mutable value) (mutable proc)
    (mutable sibs) (immutable stack)
    (mutable fp) (mutable sp) (immutable fv)))
(define stack-size 1000)

(define doing-stack thread-stack)
(define doing-stack-set! thread-stack-set!)
(define doing-fp thread-fp)
(define doing-fp-set! thread-fp-set!)
(define doing-sp thread-sp)
(define doing-sp-set! thread-sp-set!)
(define doing-fv thread-fv)
(define doing-fv-set! thread-fv-set!)
(define make-doing (lambda (cp stack fp sp fv)
                     (make-thread doing (void) cp (void) stack fp sp fv)))

(define bounce
  (lambda (cp)
    (lambda (stack fp sp fv)
      (lambda (thread)
        (to-doing! thread)
        (doing-proc-set! thread cp)
        (doing-stack-set! thread stack)
        (doing-fp-set! thread fp)
        (doing-sp-set! thread sp)
        (doing-fv-set! thread fv)))))

(define extend
  (lambda (recvr)
    (B! (Y (lambda (p!)
             (lambda (thread)
               (cond
                 ((done? thread)
                  (let ((value (done-value thread)))
                    (to-doing! thread)
                    (doing-proc-set! thread
                      (lambda (stack fp sp fv) (recvr value)))))
                 ((doing? thread)
                  (doing-proc-set! thread
                    (∘ (extend recvr) (doing-proc thread))))
                 ((multi? thread) (for-each p! (multi-sibs thread)))))))))
```

Fig. 18 Closure Avoidance Architecture, I

copies the stack for the new thread. This is inefficient; it would be preferable to have it share the existing stack and start new branches from that.[21]

We assume that the translation is revised to work as follows. Variable references are replaced with frame references, or, only if a closure already exists, with closure references. Arguments to procedures must be explicitly placed on the current frame for tail calls, or on a new frame for nontail calls. Had we also wished to eliminate the need for a stack architecture in the host system, we could explicitly pass a return address on the stack and convert the program to tail form. For tail calls, arguments are loaded into positions indexed from the `fp` and the `fp` remains unchanged. Reloading

[21] That approach, however, involves a complexity. If a stack unwinds and then grows again, the shared portion must not be overwritten.

```
(define trampoline
  (lambda (cp) (tramp (make-doing cp (make-vector stack-size) 0 0 (vector)))))

(define tramp-step
  (lambda (thread k)
    (cond
      ((done? thread)
       (let ((value (done-value thread)))
         (to-multi! thread)
         (multi-sibs-set! thread '())
         (k thread value)))
      ((doing? thread)
       (((doing-proc thread)
         (doing-stack thread)
         (doing-fp thread)
         (doing-sp thread)
         (doing-fv thread))
        thread)
       (k thread))
      ((multi? thread)
       (if (null? (multi-sibs thread))
         (k thread)
         (tramp-step (car (multi-sibs thread))
           (lambda (res . value?)
             (multi-sibs-set! thread
               (if (and (multi? res) (null? (multi-sibs res)))
                 (cdr (multi-sibs thread))
                 (append (cdr (multi-sibs thread)) (list res))))
             (apply k thread value?)))))))))
```

Fig. 19 Closure Avoidance Architecture, II

```
(define spawn
  (lambda (cp1 cp2 . cpRest)
    (lambda (thread)
      (let ((sibs
             (apply list
               (make-doing cp1
                 (doing-stack thread)
                 (doing-fp thread)
                 (doing-sp thread)
                 (doing-fv thread))
               (make-doing cp2
                 (vector-copy (doing-stack thread))
                 (doing-fp thread)
                 (doing-sp thread)
                 (doing-fv thread))
               (map (lambda (cp)
                      (make-doing cp
                        (vector-copy (doing-stack thread))
                        (doing-fp thread)
                        (doing-sp thread)
                        (doing-fv thread)))
                 cpRest))))
      (to-multi! thread)
      (multi-sibs-set! thread sibs)
      thread))))
```

Fig. 20 Closure Avoidance Operator

of arguments is not necessary in the case of recursive calls with unchanged arguments. Care must be taken not to overwrite values needed as other arguments. For nontail calls, arguments are written to locations indexed from the `sp`, and the `sp` becomes the new `fp`. All arguments must be loaded, including the case of recursive calls with unchanged arguments, but there is no danger of overwriting. In either case, the `sp` is set to the first position above the last argument. Notice that in the definition of `extend`, the new code pointer after completing execution of the initial computation simply applies `recvr` to `value`. When a call to `extend` is the result of `m-let`, references to the `stack`, `fp`, `sp`, and `fv` within `recvr` are fixed at the point of the `m-let`. Thus, when the `?rhs` of the `m-let` represents a nontail call that updates the stack and increments the registers, these will be restored upon returning by way of `recvr`. There is no danger of the free variable values being restored because they are accessed by indirection through a vector that is never copied.

The representation for existing closures changes, so that a closure created by evaluating

`(lambda (x ...)` *exp*`)`

is represented by the result of evaluating

```
(let ((fv (vector trivExp ...)))
  (lambda (stack fp sp)
    modifiedExp))
```

where each *trivExp* describes a free variable of *exp* in terms of the context of the `lambda` form and *modifiedExp* is the result of translating *exp*. This treatment of free variables requires techniques familiar to those who deal in compilation of functional languages [3]. Our representation of closures is still procedural. We would like for the number of closures to remain the same, in that `fv` should appear free in the transformed `lambda` form if and only if the original `lambda` form had at least one free variable. To this end, in the case of a `lambda` form with no free variables, we drop the `let` form and pass an empty vector to `bounce` in place of `fv`.

Finally, we present an example of a trampolined program (the translation enhanced to modify variable references) in terms of a machine state.[22] In the example, two threads start from a common base stack, but grow it in alternate directions in mapping a closure over different lists. Notice the difference in representation between tail calls, as in `map-times-tramp`, and non-tail calls, as in `map-tramp`. Also notice the creation of a closure and reference to the free variable in `map-times-tramp`.

```
> (define map-times-orig
    (lambda (n ls)
      (map (lambda (m) (* n m)) ls)))

> (define map-orig
    (lambda (f ls)
      (if (null? ls)
        '()
        (cons (f (car ls)) (map-orig f (cdr ls))))))

> (define map-times-tramp
```

[22] We use `v-ref` for `vector-ref` and `v-set!` for `vector-set!`.

```
    (lambda (stack fp sp)
      (v-set! stack (+ fp 0)
        (let ((fv (vector (v-ref stack (+ fp 0)))))
          (lambda (stack fp sp)
            (unit (* (v-ref fv 0)
                     (v-ref stack (+ fp 0))))))))
      (v-set! stack (+ fp 1) (v-ref stack (+ fp 1)))
      (map-tramp stack fp sp)))

> (define map-tramp
    (lambda (stack fp sp)
      (if (null? (v-ref stack (+ fp 1)))
        (unit '())
        (m-let*
          ((res ((bounce
                   (lambda (stack fp sp fv)
                     (v-set! stack (+ sp 0) (car (v-ref stack (+ fp 1))))
                     ((v-ref stack (+ fp 0)) stack sp (+ sp 1))))
                 stack fp sp (vector)))
           (rest ((bounce
                    (lambda (stack fp sp fv)
                      (v-set! stack (+ sp 0) (v-ref stack (+ fp 0)))
                      (v-set! stack (+ sp 1)
                        (cdr (v-ref stack (+ fp 1))))
                      (map-tramp stack sp (+ sp 2))))
                  stack fp sp (vector))))
          (unit (cons res rest))))))

> (trampoline (lambda (stack fp sp fv)
                (v-set! stack (+ fp 0) 3)
                (spawn
                  (lambda (stack fp sp fv)
                    (v-set! stack (+ fp 1) '(1 2))
                    (map-times-tramp stack fp (+ fp 2)))
                  (lambda (stack fp sp fv)
                    (v-set! stack (+ fp 1) '(3 4))
                    (map-times-tramp stack fp (+ fp 2))))))
[(3 6) (9 12)]
```

Theorem 5.1 *In Closure Avoidance,* Mρ*,* `unit`*, and* `extend` *form a monad, subject to the equality* `bounce` = `id`*.*

Proof The proof uses techniques similar to those of Theorem 4.1.

Theorem 5.2 *Closure Avoidance is a trampolining architecture.*

Proof For Laws #4 and #5, we again show, respectively:

(∘ (extend *recvr*) (bounce *cp*)) =
(bounce (∘ (extend recvr) *cp*))

and

```
(∘ trampoline bounce) = trampoline
```

Both follow without much difficulty.

5.2 Logic Programming

Since logic programming involves trying multiple paths, any of which could either yield an answer or not terminate, trampolining is a natural solution. A trampolining architecture can be used to implement a logic programming system that interleaves the execution of all search paths, thus terminating whenever possible and eventually reaching any solution. We no longer build cumulatively upon our work to this point; one can think of the Logic Programming architecture as an extension of the architecture for Dynamic Thread Creation and Termination of Section 3.2. As in that architecture, the computation type constructor M is defined such that $\mathsf{M}\rho\alpha = \mathsf{List}\ \mathsf{T}\rho\alpha$, with α now representing the type of substitutions. However, we now set $\bar{x} =$ `subst`. In so doing, we allow user computations to reflectively access or even update the substitution.

The thread type constructor T is redefined such that $\mathsf{T}\rho\alpha = \alpha + (\mathsf{G}\rho\alpha \times \alpha)$. The type of goals is in turn defined as $\mathsf{G}\rho\alpha = \alpha \to \mathsf{M}\rho\alpha$. A goal takes a substitution and refines it in various ways as it generates subgoals. Success occurs when a refined substitution is offered with no further subgoal. Failure occurs when neither refined substitutions nor subgoals are made available.

Similar to the case of Section 3.2, the `unit` procedure accepts a substitution and builds a singleton list of a `done` record. Thus, `unit` itself is a goal. `bounce` packages a goal and a substitution as a suspended computation. This pair plays the role of the thunk in the previous architectures. The type of `bounce` is now $\mathsf{G}\rho\alpha \to \alpha \to \mathsf{M}\rho\alpha$.

We now have two ways to feed the result of one computation to another. The first is very similar to the `extend` operator of Section 3.2. Note, however, that the `recvr` parameter now takes the form of a goal.

The second is obtained by taking advantage of this property of `recvr` by swapping it with the suspended goal each time computations are composed. It makes use of the property that substitutions form a lattice with a refinement relation made up of goals; one can swap the suspended goal with the receiver because the order in which goals are applied to a substitution is irrelevant. In a chain of calls to `interleave`, processing will alternate "depthwise" between `goal` (where `comp` $= \iota_r$`(goal, subst)`) and `recvr`, before `comp` produces any value. This will allow us to anticipate failures due to `recvr`, even if `comp` diverges without producing a substitution.[23]

`trampoline` must now take a goal rather than a thunk and form a suspension using the empty substitution, which we define to be the empty list. `trampoline` is thus of type $(\forall\rho.\mathsf{G}\rho\alpha) \to \mathsf{Stream}\ \alpha$. The thread queue is processed by the helper procedure `tramp`, which is still of type $(\forall\rho.\mathsf{List}\ \mathsf{T}\rho\alpha) \to \mathsf{Stream}\ \alpha$.

We now identify some ways of obtaining goals. We have already mentioned that `unit` is a goal; we now call it `succeed` since it always passes its substitution through unchanged, without generating any subgoals. `fail`, by contrast, neither accepts its given substitution nor generates any subgoals. It corresponds to `die` of Section 3.2.

[23] We could redefine the macros `m-let` and `m-begin` to use the `interleave` form but instead leave them to be interpreted based on context.

```
(define-record-type done (fields (immutable subst)))
(define-record-type doing (fields (immutable goal) (immutable subst)))

(define unit (∘ unit_List make-done))

(define bounce (lambda (goal) (lambda (subst) (unit_List (make-doing goal subst)))))

(define extend
  (lambda (recvr)
    (extend_List
      (lambda (thread)
        (record-case thread
          (done (subst) (recvr subst))
          (doing (goal subst) ((bounce (∘ (extend recvr) goal)) subst)))))))

(define interleave
  (lambda (recvr)
    (extend_List
      (lambda (thread)
        (record-case thread
          (done (subst) (recvr subst))
          (doing (goal subst) ((bounce (∘ (interleave goal) recvr)) subst)))))))

(define empty-subst '())

(define trampoline (lambda (goal) (tramp (list (make-doing goal empty-subst)))))

(define tramp
  (lambda (tq)
    (if (pair? tq)
      (record-case (car tq)
        (done (subst) (stream-cons subst (tramp (cdr tq))))
        (doing (goal subst) (tramp (append (cdr tq) (goal subst)))))
      (make-empty-stream))))
```

Fig. 21 Logic Programming and Interleaved Logic Programming Architectures

Unifying two terms either leads to success with a new substitution or to failure. We do not specify the details of unification [1] here. A pair of binary operators over goals, `allSeq` and `allInt`, make use of `extend` and `interleave`, respectively. A corresponding binary `any` operator is similar to `spawn`, but also takes the form of a goal operator. This will allow a computation to succeed if any disjunct does, even if others diverge. Three more operators provide iterated transitive closures of a goal over any of these binary operators.

We make use of `trans` to provide a transitive closure of a goal. A more general form of recursion over goals is available with `gletrec`. In each case a `bounce` is required to return control to the scheduler.

We can now see what these operators do.[24] `allInt` provides a solution (substitution) whenever `allSeq` does, but not vice versa; interleaving is necessary to find a substitution when the first sequenced computation would diverge. $\texttt{any}^+$ can never fail and $\texttt{allInt}^+$ (and thus $\texttt{allSeq}^+$) can never succeed. Neither `failing`, defined as ($\texttt{any}^+$ `fail`) nor `succeeding`, defined as ($\texttt{allInt}^+$ `succeed`) will terminate; the former cannot fail because of the illusory hope that some subsequent computation might suc-

[24] The reader is referred back to the descriptions of stream notations in Section 3.2.

```
(define succeed unit)

(define fail (lambda (subst) '()))

(define any
  (lambda (goal1 goal2)
    (lambda (subst) (list (make-doing goal1 subst) (make-doing goal2 subst)))))

(define allSeq (lambda (goal1 goal2) (∘ (extend goal2) goal1)))

(define allInt (lambda (goal1 goal2) (∘ (interleave goal2) goal1)))

(define trans
  (lambda (op)
    (lambda (goal) (op goal (bounce (lambda (subst) ((trans op) goal) subst))))))

(define any+ (trans any))
(define allSeq+ (trans allSeq))
(define allInt+ (trans allInt))

(rewrites-as (gletrec ((x g1)...) g2) (letrec ((x (bounce g1))...) g2))
```

Fig. 22 Logic Programming and Interleaved Logic Programming Operators

ceed, while the latter cannot succeed because of the illusory hope that some subsequent computation might fail. On the other hand, `always` (defined as (`any`$^+$ `succeed`)) creates an infinite stream of substitutions, while (`allInt`$^+$ `fail`) and (`allSeq`$^+$ `fail`) just fail. In fact, `allSeq` is able to fail whenever this is possible based on the first sequential step of computation, as in `(allSeq fail failing)` and `(allSeq fail always)`, but must diverge whenever the first sequential step does, as in `(allSeq failing fail)`. By contrast, even if the first sequential step of computation does not terminate (`trampoline` continually generates goals), we can still fail using `allInt` whenever this is possible based on any sequential step of computation, as additionally in `(allInt failing fail)` and `(allInt always fail)`. The former fails at once although the inital computation diverges, an obstacle that `(allSeq failing fail)` never surmounts. The latter fails at once for all possible successes of the inital computation, where `(allSeq always fail)` would diverge "breadthwise", trying every such possible success.

Examples such as `(any always succeeding)` and `(any failing always)` demonstrate that `any` is not hampered by nonterminating computations in returning an infinite stream of successes. If, however, only one success is possible, as in `(any succeeding succeed)` or `(any succeed failing)`, we will produce the single substitution and then diverge.

```
> (trampoline succeed)
[()]

> (trampoline fail)
[]

> (define succeeding (allInt+ succeed))
> (trampoline succeeding)
[***]
```

```
> (define failing (any+ fail))
> (trampoline failing)
[***]

> (define always (any+ succeed))
> (trampoline always)
[()...]

> (trampoline (allInt+ fail))
[]

> (trampoline (allSeq fail failing))
[]

> (trampoline (allSeq fail always))
[]

> (trampoline (allSeq failing fail))
[***]

> (trampoline (allInt failing fail))
[]

> (trampoline (allSeq always fail))
[***]

> (trampoline (allInt always fail))
[]

> (trampoline (any always succeeding))
[()...]

> (trampoline (any failing always))
[()...]

> (trampoline (any succeeding succeed))
[() ***]

> (trampoline (any succeed failing))
[() ***]
```

Some may wonder what this all has to do with logic programming. To clarify the connection, we make a few changes to the language.

1. Generalize constants c to $(c\ \ E\ldots)$.
2. Drop `set!`, `begin`, and `if` from the language.
3. Define an operator `unify` taking two terms and a substitution to a modified substitiution and use it in:

```
(define ==
  (lambda (v w)
    (lambda (s)
      (cond
        ((unify v w s) => succeed)
        (else (fail s))))))
```

4. Allow unbound variables to self-evaluate.
5. Generalize `lambda` to take the form `(lambda` P E`)` where $P ::= (c\ P \ldots) \mid x$
   Interpret an application of `(lambda` P E_1`)` to E_2 as

   `(m-let (v` E_2`) ((interleave (==` P `v))` E_1`))`

Theorem 5.3 *In Logic Programming,* Mρ*,* `unit`*, and* `extend` *form a monad.*

Proof The proof is entirely analogous to that of Theorem 3.3. For the second law, the context $C[] =$

```
(List
  (lambda (thread)
    (record-case thread
      (done (subst) (make-done subst))
      (doing (goal subst) (make-doing (∘ [] goal) subst)))))
```

For the third law, the context $C_{v^{\mathtt{f}},v^{\mathtt{g}}}[] =$

```
(extend_List
  (lambda (thread)
    (record-case thread
      (done (subst) ((extend v^f) (v^g subst)))
      (doing (goal subst) ((bounce (∘ [] goal)) subst)))))
```

Theorem 5.4 *Logic Programming is a trampolining architecture.*

Proof For Law #4 we show:

```
(∘ (extend recvr) (bounce goal)) =
(bounce (∘ (extend recvr) goal))
```

It again follows from the definitions.

Lemma 5.1 *In Interleaved Logic Programming,* `interleave` *can resolve goals in any order, subject to the equality* `bounce` $=$ `id`*.*

```
(∘ (interleave goal2) goal1) ≈
(∘ (interleave goal1) goal2)
```

Proof The proof uses the `bounce` equality initially to obtain `((bounce` *goal*1`)` *subst*`)` and then reapplies it in the opposite direction to obtain `((∘ (interleave` *goal*1`)` *goal*2`)` *subst*`)`.

Lemma 5.2

```
(∘ (interleave f) (interleave g)) =
(extend_List
  (lambda (thread)
    (record-case thread
      (done (subst) ((interleave f) (g subst)))
      (doing (goal subst)
       ((bounce (∘ (interleave (∘ (interleave goal) g)) f))
        subst)))))
```

Proof The proof is direct.

Theorem 5.5 *In Interleaved Logic Programming,* Mρ, `unit`, *and* `interleave` *form a monad, subject to the equality* `bounce` = `id`.

Proof The first monad law is straightforward. The second monad law uses the first on the `doing` clause. The third monad law is by coinduction. On the left side, simply apply Lemma 5.2. On the right side, apply Lemma 5.2 and then Lemma 5.1 to the innermost `interleave` in the `doing` clause. We have $C_{v^{\mathtt{f}},v^{\mathtt{g}}}[] =$

```
(extend_List
  (lambda (thread)
    (record-case thread
      (done (subst) ((interleave v^f) (v^g subst)))
      (doing (goal subst) ((bounce (∘ [] v^f)) subst)))))
```

with f = `goal` in the next iteration. Interestingly, the sides of the equation reverse with each context descended, so that the value relation R must be symmetric.

Theorem 5.6 *Interleaved Logic Programming is a trampolining architecture.*

Proof For Law #4 we show:

```
(∘ (interleave (bounce recvr)) (bounce goal)) =
(bounce (bounce (∘ (interleave recvr) goal)))
```

Theorem 5.7 *The monad of the Interleaved Logic Programming architecture is commutative, subject to the equality* `bounce` = `id`.

```
(∘ (interleave f) (interleave g)) ≈
(∘ (interleave g) (interleave f))
```

Proof The proof is by coinduction. Apply Lemma 5.2 to each side, then Lemma 5.1 to each side at both `interleave`s of the `doing` clause. Now apply Lemma 5.1 again to the `done` clause on the left side. We have $C_{v^{\mathtt{f}},v^{\mathtt{g}}}[] =$

```
(extend_List
  (lambda (thread)
    (record-case thread
      (done (subst) ((interleave v_g) (v_f subst)))
      (doing (goal3 subst) ((bounce (∘ [] goal3)) subst)))))
```

6 Related Work

The history of trampolining has been surveyed before [10]. Our approach is similar to that of Queinnec and De Roure, who also implemented multitasking primitives through program transformations [26]. Related work on concurrency facilities using monads has been done in Haskell. Hudak and Jones implement channels in Haskell [14]. Scholz discusses synchronous interaction using streams [29].

Several independent presentations of similar ideas around the time of our earlier work [10] attest to the fact that models of trampolining were "in the air". We have already mentioned Filinski's presentation [9] of the Dynamic Threads Architecture as a layered monad. Claessen presented a "Poor Man's Concurrency Monad" [4]. While unconcerned with conversion of programs or placement of "bounces", he described a monad transformer for modelling multithreading, making his work in many ways similar to that described in Section 3.2 above. Our use of `done` records, however, allows multi-threaded computations to terminate and a value to be returned without relying upon side effects in a prior monad.[25] Our support for `bounce` gives the user of the language (or designer of the translation) flexibility in specifying when transfers of control will occur, again without relying on effects in a prior monad. Subsequently, Papaspyrou [24] presented a function `step` that converts a non-interleaved computation into an interleaved one using a resumption monad transformer.

Harrison has provided a monadic implementation supporting asynchronous interthread broadcast communication and a separate synchronization mechanism, based on a "reactive" resumption monad [12]. He does not support a notion of fairness similar to ours, recycle thread records, nor provide a symmetric spawn operator such that a thread might receive values from any descendent.

Kiselyov, Shan, et. al., describe a monad transformer for interleaving execution of logic programs [15]. Their implementations are also continuation-based; in this case a 2-continuation (for success and failure) approach and one based on delimited continuations. By comparison, our implementation in Section 5.2 is both simple and transparent. Unlike their `>>-` operator, our `interleave` supports recognition of a failure in the dependent conjunct. It fixes depthwise concurrency (between a computation and its receiving procedure) into the `interleave` operator while leaving breadthwise interleaving to the placement of `bounce` forms.

Bawden [2] cps'es an interpreter and then trampolines it by pausing upon applications of continuations. The resulting interpreted language can then be made to support reflective capabilities. In previous formulations of reflection, including Smith's [31], it is by writing and reifying a piece of interpreter that one obtains access to the processor state, and the continuation is treated as just another element of that state. Bawden's formulation, in which procedures are to be converted between user-level and implementation-level forms, similarly treats the continuation as processor state available to any procedure that a user may want to reflect. It also ties the reflective interface to user-defined procedures. For us, by contrast, the continuation is only made available to user code by a specific design decision. In fact, it is almost the opposite—creating a suspension procedure and passing it to `bounce` provides access to the processor state as the procedure's arguments. Because our translation is based on the monadic translation rather than a cps translation, that suspension procedure need not be a continuation. Our reflective interface is not dependent on user-defined procedures or any other lan-

[25] Claessen's `Stop` record is more similar to our `die` operator.

guage feature. We do not attempt to model any tower of interpreters in this work. As previously intimated [10], this could be achieved with multiple applications of the translation. Any operator, not just `bounce`, could then refer to the level at which it is intended to be applied.

We make the substitution accessible in logic programs as an example of how reflective principles can be applied to a trampolining-based implementation of this paradigm. By comparison with previous work on reflection in logic programming that allows a user to make assertions in terms of a provability predicate (for example, by Costantini and Lanzarone [5]), this approach is quite low-level. However, we see no reason why it couldn't be extended to provide more expressive reflective capabilities. Pientka provides a type system for a language with first-class substitutions [25].

7 Conclusion

We have reviewed and refined the trampolining translation and presented several trampolining architectures within the monadic approach. We then proceeded to present additional architectures that, while loosening the strictures of a monadic model, have at the same time reaffirmed the viability of the monadic approach to concurrency in the implementation of realistic systems. We presented an array of concurrency and logic programming facilities and demonstrated the potential for their efficient implementation in a wide variety of languages. The architectures presented in Sections 4.2 through 5.1 differ from those in Section 3 by making use of state in their implementations. User programs relying on the operators we can provide with those architectures may, however, use fewer explicit side-effects (compare the two definitions of the Fibonacci function using multiple threads). The architecture in Section 5.2 supports interleaving of both conjunctions and disjunctions over goals. Both architectures in Section 5 demonstrate the ability to make thread-level implementation information available to user code.

Applying these concepts to processing over muliple cores, rather than interleaved processing on a single core is beyond the scope of this paper. We point out only that the threads on the queue can be scheduled in any order and, in fact, in parallel. A complication is that atomicity would not longer come for free; a mechanism for protecting access to shared resources (including shared memory) would be needed. Ideally, the scheduler could group computations in a way that would facilitate access to these shared resources. Our enhanced notion of fairness would be complicated by data-parallel architectures, where a computation can spawn multiple subcomputations with limited crowding-out of siblings.

The trampolining architectures that we have presented allow for the seamless integration of synchronous and asynchronous forms of interthread communicaton and for a simple yet powerful model of logic programming. They deal as well with issues of efficiency of implementation. We have characterized precisely the extent to which these architectures are monadic, i.e., without regard to the distribution of `bounce` forms (yielding control) throughout a computation. This can be thought of as compensating for the difference between the trampolining translation and the monadic translation, i.e., that the former inserts `bounce` forms. We found that the need for this generalization is due in one case to the recycling of thread records across the `extend` form and in the case of logic programming to the periodicity inherent in the interleaving `extend` form. Is this a totally negative result, suggesting that other techniques be applied to

multitasking systems? Or does this point to a useful generalization that is worthy of study in its own right? What is clear at this point is that a range of abstractions useful in representing computation are somewhat but not completely amenable to the monadic framework.

Acknowledgements We are indebted to a referee of an earlier draft of this article, who recognized some of the subtlety of the relationship between trampolining architectures and monads. We are indebted as well to Chung-chieh Shan for an insight that made the elegant code in Section 5.2 possible. That section also benefited from discussions with Will Byrd regarding appropriate termination behavior for logic programs. Discussions sponsored by both the Silicon Valley Patterns Group and the SF Bay Area Categories and Types Group were helpful. Finally, thanks to Jiho Kim for careful reading that resulted in catching errors and raising issues, which improved the final result.

References

1. Baader, F., Snyder, W.: Unification theory. In: J.A. Robinson, A. Voronkov (eds.) Handbook of Automated Reasoning, chap. 8. Elsevier Science and MIT Press (2001)
2. Bawden, A.: Reification without evaluation. In: LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming, pp. 342–349. ACM, New York, NY, USA (1988). DOI http://doi.acm.org/10.1145/62678.62726
3. Cardelli, L.: Compiling a functional language. In: Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, pp. 208–217. ACM Press, Austin, TX, USA (1984)
4. Claessen, K.: A poor man's concurrency monad. Journal of Functional Programming **9**(3), 313–324 (1999)
5. Costantini, S., Lanzarone, G.A.: A metalogic programming approach: Language, semantics and applications. Journal of Experimental & Theoretical Artificial Intelligence **6**(3), 239–287 (1994)
6. Dybvig, R.K., Flatt, M., van Stratten, A.: Revised^6 report on the algorithmic language scheme (2007). `http://www.r6rs.org/versions/r6rs.pdf`
7. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoretical Computer Science **103**, 235–271 (1992)
8. Filinski, A.: Representing monads. In: Proceedings of the 21st Symposium on Principles of Programming Languages, pp. 446–457. ACM Press, New York (1994)
9. Filinski, A.: Representing layered monads. In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages, pp. 175–188. ACM Press (1999)
10. Ganz, S.E., Friedman, D.P., Wand, M.: Trampolined style. In: Proceedings of the 4th International Conference on Functional Programming, *SIGPLAN Notices*, vol. 34.9, pp. 18–27. ACM Press, Paris, France (1999)
11. Halstead, R.: Multilisp: A language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems **7**(4), 501–538 (1985)
12. Harrison, W.L.: The essence of multitasking. In: Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology, *Lecture Notes in Computer Science*, vol. 4019, pp. 158–172. Springer, Kuressaare, Estonia (2006)
13. Haynes, C.T., Friedman, D.P.: Abstracting timed preemption with engines. Computer Languages **12**(2), 109–121 (1987)
14. Jones, M., Hudak, P.: Implicit and explicit parallel programming in Haskell. Tech. Rep. RR-982, Yale University, Department of Computer Science (1993)
15. Kiselyov, O., Shan, C., Friedman, D.P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers (functional pearl). In: Proceedings of the 10th International Conference on Functional Programming, *SIGPLAN Notices*, vol. 40.9, pp. 192–203. ACM Press, Paris, France (2005)
16. Kohlbecker, E.E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: Symposium on LISP and Functional Programming, pp. 151–161. ACM (1986)

17. Koutavas, V., Wand, M.: Small bisimulations for reasoning about higher-order imperative programs. In: Proceedings of the 33rd Symposium on Principles of Programming Languages, pp. 141–152. ACM Press (2006)
18. Launchbury, J., Jones, S.L.P.: State in Haskell. Lisp and Symbolic Computation **8**(4), 293–341 (1995)
19. Manolescu, D.A.: Workflow enactment with continuation and future objects. In: Proceedings of the Conerence on Object-Oriented Programming, Systems, Languages, and Applications, *SIGPLAN Notices*, vol. 37.11, pp. 40–51. ACM, ACM Press, Seattle, WA, USA (2002)
20. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989, pp. 14–23. IEEE Computer Society Press, Washington, DC, USA (1989)
21. Moggi, E.: An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Department of Computer Science, Edinburgh University (1990)
22. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1), 55–92 (1991)
23. Nagle, J.: On packet switches with infinite storage. IEEE Transactions on Communications **35**(4), 435–438 (1987)
24. Papaspyrou, N.: A resumption monad transformer and its applications in the semantics of concurrency. In: Proceedings of the 3rd Panhellenic Logic Symposium (2001)
25. Pientka, B.: Type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: Proceedings of the 19th Symposium on Principles of Programming Languages, pp. 371–382. ACM Press, San Francisco, CA, USA (2008)
26. Queinnec, C., DeRoure, D.: Design of a concurrent and distributed language. In: J. Robert H. Halstead, T. Ito (eds.) Parallel Symbolic Computing: Languages, Systems, and Applications (US/Japan Workshop Proceedings), *Lecture Notes in Computer Science*, vol. 748, pp. 234–259. Springer, Cambridge, MA, USA (1993)
27. Ramanujam, R., Lodaya, K.: Proving fairness of schedulers. In: R. Parikh (ed.) Proceedings of the Conference on Logic of Programs, *Lecture Notes in Computer Science*, vol. 193, pp. 284–301. Springer, Brooklyn, NY, USA (1985)
28. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the 25th ACM National Conference, pp. 717–740. ACM Press, New York, NY, USA (1972)
29. Scholz, E.: Imperative streams — a monadic combinator library for synchronous programming. In: Proceedings of the 3rd International Conference on Functional Programming, *SIGPLAN Notices*, vol. 34.1, pp. 261–272. ACM Press, Baltimore, MD, USA (1998)
30. Schönfinkel, M.: ber die Bausteine der mathematischen Logik, vol. 92, pp. 305–316. Springer Verlag (1924). Translated by Stefan Bauer-Mengelberg as "On the building blocks of mathematical logic" in Jean van Heijenoort, 1967. A Source Book in Mathematical Logic, 18791931. Harvard Univ. Press: 35566
31. Smith, B.C.: Reflection and semantics in a procedural language. Ph.D. thesis, Massachusetts Institute of Technology (1982)
32. Smullyan, R.: To Mock a Mockingbird and Other Logic Puzzles: Including an Amazing Adventure in Combinatory Logic. Knopf (1985)
33. Sobel, J., Friedman, D.P.: Recycling continuations. In: Proceedings of the 3rd International Conference on Functional Programming, *SIGPLAN Notices*, vol. 34.1, pp. 261–272. ACM Press, Baltimore, MD, USA (1998)
34. Wadler, P.: The essence of functional programming. In: Proceedings of the 19th Symposium on Principles of Programming Languages, pp. 1–14. ACM Press, Albuquerque, NM, USA (1992)
35. Wand, M.: The theory of Fexprs is trivial. Lisp and Symbolic Computation **10**(3), 189–199 (1998)