The Essence of Multitasking*

William L. Harrison

Department of Computer Science, University of Missouri,
Columbia, Missouri, USA.

Abstract. This article demonstrates how a powerful and expressive abstraction from concurrency theory—monads of resumptions—plays a dual rôle as a programming tool for concurrent applications. The article demonstrates how a wide variety of typical OS behaviors may be specified in terms of resumption monads known heretofore exclusively in the literature of programming language semantics. We illustrate the expressiveness of the resumption monad with the construction of an exemplary multitasking kernel in the pure functional language Haskell.

1 Introduction

Many techniques and structures have emigrated from programming language theory to programming practice (e.g., types, CPS, etc.), and this paper advocates that resumption monads make this journey as well. This work demonstrates how a natural (but, perhaps, under-appreciated) computational model of concurrency is used to construct multi-threaded concurrent applications suitable for formal verification. The expressiveness of resumption monads is illustrated by the construction of an exemplary multitasking operating system kernel with process forking, preemption, message passing, and synchronization constructs all requiring about fifty lines of Haskell 98 code[1]. And, because this machinery may be generalized as monad transformers, the functionality described here may be reused and refined easily.

The literature involving resumption monads [21, 23, 11, 22, 18, 17] focuses on their use in elegant and abstract mathematical semantics for programming languages. The current work advocates resumption monads as a useful abstraction for concurrent functional programming as well. **The contributions of this work are twofold:** (1) the formulation of typical concurrent operating system behaviors in terms of structures known heretofore in theoretical semantics literature and (2) a substantial case study illustrating this formulation within a higher-order functional programming language. The purpose of the case study, in part, is to provide an exposition so that the interested reader may grasp the theoretical literature more readily.

* This research supported in part by subcontract GPACS0016, System Information Assurance II, through OGI/Oregon Health & Sciences University.

[1] All the code presented in this paper is available online [13].

A *resumption* [27] is stream-like construction similar to a continuation in that both tell what the "rest of the computation" is. However, resumptions are considerably less powerful than continuations—the *only* thing one may model with resumptions is multitasking computation. This conceptual economy makes concurrent applications structured with resumption monads easy to comprehend, modify, extend, and reason about. Specifically, we demonstrate how to construct a multitasking operating system kernel based on three monads and their operations (written here in categorical style):

$$
\begin{array}{lll}
St\ A &= Sto\ \rightarrow\ A \times Sto & \text{— state} \\
R\ A &= \mu X.\,(A\ +\ (St\ X)) & \text{— state+concurrency} \\
Re\ A &= \mu X.\,(A\ +\ (Req \times (Rsp \rightarrow St\ X))) & \text{— state+concurrency+interactive i/o}
\end{array}
$$

St is the familiar state monad, while *R* and *Re* are resumption monads providing what we call *basic* and *reactive* concurrency about which we will say much more below.

The structure of this article is as follows. After reviewing the related work below and the necessary background in Section 2, Section 3 describes in detail how resumption monads may be used to model multitasking concurrency. Section 4 presents a resumption-monadic semantics for a concurrent language extended with "signals"; a thread may signal the kernel to fork, suspend, preempt, print, send or receive a message, and acquire or release a semaphore and Section 5 describes the kernel on which these threads execute. Section 6 summarizes the work and outlines future directions.

Related Work. Functional languages are well-known for promoting mathematical reasoning about programs, and, perhaps because of this, there has been considerable research into their use for concurrent software such as OS kernels. The present work has this pedigree, yet fundamentally differs from it in at least one key respect: we explicitly encapsulate all effects necessary to the kernel with monads: input/output, shared state and multitasking concurrency.

The concurrency models underlying previous applications of functional languages to concurrent system software fall broadly into four camps. The first camp [16, 31, 32, 5] assumes the existence of a non-deterministic choice operator to accommodate "non-functional" situations where more than one action is possible, such as a scheduler choosing between two or more waiting threads. However, such a non-deterministic operator risks the loss of an important reasoning principle of pure languages—referential transparency—and considerable effort is made to minimize this danger. Non-determinism may be incorporated easily into the kernel presented here via the non-determinism monad, although such non-determinism is of a different, but closely related, form.

The second model uses "demand-driven concurrency" [3, 30] in which threads are mutually recursive bindings whose lazy evaluation simulates multitasking concurrency. Interleaving order is determined (in part) by the interdependency of these bindings. However, the demand-driven approach requires some alteration of the underlying language implementation to completely determine thread scheduling. Thread structure is entirely implicit—there are no atomic actions *per*

se. Demand determines the extent to which a thread is evaluated—rather like the "threads" encoded by computations in the lazy state monad [19]. Thread structure in the resumption-monadic setting is explicit—one may even view a resumption monad as an abstract data type for threads. This exposed thread structure allows deterministic scheduling without changing the underlying language implementation as with demand-driven concurrency.

The third camp uses CPS to implement thread interleaving. Concurrent behavior may be modeled with first-class continuations [4, 34, 9, 33] because the explicit control over evaluation order in CPS allows multiple threads to be "interwoven" to produce any possible execution order. Claessen presents a formulation of this style using the CPS monad transformer [4], although without exploiting the full power of first-class continuations—i.e., he does not use *callcc* or *shift* and *reset*. While it is certainly possible to implement the full panoply of OS behaviors with CPS, it is also possible to implement much, much more—most known effects may be expressed via CPS [7]. This expressiveness can make programs in CPS difficult to reason about, rendering CPS less attractive as a foundation for software verification. Resumptions can be viewed as a disciplined use of continuations which allows for simpler reasoning.

The last camp uses a program-structuring paradigm for multi-threading called trampoline-style programming [10]. Programs in trampoline-style are organized around a single scheduling loop called a "trampoline." One attractive feature of trampolining is that it requires no appeal to first-class continuations. Of the four camps, trampolining is most closely related to the resumption-monadic approach described here. In [10], the authors motivate trampolining with a type constructor equivalent to the functor part of the basic resumption monad (described in Section 3.1 below), although the constructor is never identified as such.

The previous research relevant to this article involves those applications of functional languages where the concurrency model is explicitly constructed rather than inherited from a language implementation or run-time platform. There are many applications of functional languages to system software that rely on concurrency primitives from existing libraries or languages [12, 1]; as the modeling of concurrency is not their primary concern, no further comparison is made. Similarly, there are many concurrent functional languages—concurrent versions of ML, Haskell, and Erlang—but their concurrency models are built-in to their run-time systems and provide no basis of comparison to the current work. It may be the case, however, that the resumption-monadic framework developed here provides a semantic basis for these languages.

Resumptions are a denotational model of concurrency first introduced by Plotkin [27]; excellent introductions to this non-monadic form of resumptions are due to Schmidt [29] and Bakker [2]. Moggi was the first to observe that the categorical structure known as a monad was appropriate for the development of modular semantic theories for programming languages [21]. In his initial development, Moggi showed how a sequential theory of concurrency could be expressed in the resumption monad. The particular formulation of the basic re-

sumption monad we use is due to Papaspyrou [23, 22], although other equivalent formulations exist [21, 6, 8].

2 Review: Monads

Monads are algebras just as groups or rings are algebras; that is, a monad is a type constructor (functor) with associated operators obeying certain equations—the well-known "monad laws" [20]. There are several formulations of monads, and we use one familiar to functional programmers called the Kleisli formulation: a monad M is given by an eponymous type constructor M and the *unit* operator, **return** $: a \to M\ a$, and the *bind* operator, (>>=) $: M\ a \to (a \to M\ b) \to M\ b$. We assume of necessity that the reader possesses familiarity with monads and their uses in modeling effects. Readers requiring further background should consult the references [21, 20].

The State Monad in Haskell. We represent the monadic constructions here in the pure functional language Haskell 98 [25], although we would be equally justified using categorical notation or any other higher-order functional programming language.

A monad in Haskell typically consists of a data type declaration (defining the computational "raw materials" encapsulated by the monad) and definitions for the overloaded symbols (**return**) and (>>=) [25]. The state monad St, containing a single threaded state $Sto = Loc \to Int$, is declared:

$$\begin{array}{ll} \textbf{data}\ St\ a = ST\ (Sto \to (a, Sto)) & \textbf{return}\ v = ST\ (\lambda s.\ (v, s)) \\ deST\ (ST\ x) = x & (ST\ x) \texttt{>>=} f = ST(\lambda s.\ let\ (y, s') = (x\ s) \\ & \qquad\qquad in\ deST\ (f\ y)\ s') \end{array}$$

The state monad has operators for updating the state, u, getting the state, g, and reading a particular location, *getloc*:

$$\begin{array}{ll} u : (Sto \to Sto) \to St\ () & u\ \delta = ST\ (\lambda\ s.\ ((), \delta\ s)) \\ g : St\ Sto & g = ST\ (\lambda s.\ (s, s)) \\ getloc : Loc \to St\ Int & getloc\ x = g \texttt{>>=} \lambda\ \sigma.\ \textbf{return}\ (\sigma\ x) \end{array}$$

Here, () is both the single element unit type and its single element. The "null" bind operator, $(\texttt{>>}) : M\ a \to M\ b \to M\ b$, is useful when the result of >>='s first argument is ignored: $x \texttt{>>}\ y = x \texttt{>>=} \lambda\ \_.\ y$.

Notational Convention. We suppress details of Haskell's concrete syntax when they are unnecessary to the presentation (in particular, instance declarations and class predicates in types). Haskell 98 reverses the standard use of (::) and (:) in that (::) stands for "has type" and (:) for list concatenation in Haskell 98. We will continue to use the standard interpretation of these symbols.

3 Concurrency Based on Resumptions

Two formulations of resumption monads are used here–what we call *basic* and *reactive* resumption monads. Both occur, in one form or another, in the literature [21, 23, 22, 6, 8]. The *basic* resumption monad (Section 3.1) encapsulates a notion of multitasking concurrency; that is, its computations are stream-like and may be woven together into single computations representing any arbitrary schedule. The *reactive* resumption monad (Section 3.2) encapsulates multitasking concurrency as well, but, in addition, affords a request-and-response interactive notion of computation which, at a high-level, resembles the interactions of threads within a multitasking operating system.

To motivate resumptions, let's compare them with a natural model of concurrency known as the "trace model" [28]. The trace model views threads as (potentially infinite) streams of atomic operations and the meaning of concurrent thread execution as the set of all their possible thread interleavings. Imagine that we have two simple threads $a = [a_0, a_1]$ and $b = [b_0]$, where a_0, a_1, and b_0 are "atomic" operations, and, if it is helpful, think of such atoms as single machine instructions. According to the trace model, the concurrent execution $a \| b$ of threads a and b is denoted by the set[2] of all their possible interleavings:

$$\mathit{traces}\,(a \parallel b) = \{[a_0, a_1, b_0], [a_0, b_0, a_1], [b_0, a_0, a_1]\} \qquad (\ddagger)$$

This means that there are three distinct possible execution traces of $(a \parallel b)$, each of which corresponds to an interleaving of the atoms in a and b. Non-determinism in the trace model is reflected in the fact that $\mathit{traces}(a \parallel b)$ is a set consisting of multiple interleavings.

The trace model captures the structure of concurrent thread execution abstractly and is well-suited to formal characterizations of properties of concurrent systems (e.g., liveness). However, a gap exists between this formal model and an executable system: traces are streams of events, and each event is itself a place holder (i.e., what do the events a_0, a_1, and b_0 actually do?). Resumption monads bridge this gap because they are both a formal, trace-based concurrency model and may be directly realized and executed in a higher-order functional language.

The notion of computation provided by resumption monads is that of sequenced computation. A resumption computation has a stream-like structure in that it includes both a "head" (corresponding to the next action to perform) and a "tail" (corresponding to the rest of the computation)—very much like the execution traces in (‡). We now describe the two forms of resumption monads in detail.

3.1 Sequenced Computation & Basic Resumptions

This section introduces sequenced computation in monadic style, discussing the monad that combines resumptions with state. The monad combining resump-

[2] This set is also *prefix-closed* in Roscoe's model, meaning that it includes all prefixes of any trace in the set. For the purposes of this exposition, we ignore this consideration.

tions with state is:

$$\begin{array}{ll}
\textbf{data}\ R\ a & = \mathit{Done}\ a \mid \mathit{Pause}\ (\mathit{St}\ (R\ a)) \\
\textbf{return} & = \mathit{Done} \\
(\mathit{Done}\ v)\ \texttt{>>=}\ f & = f\ v \\
(\mathit{Pause}\ r)\ \texttt{>>=}\ f & = \mathit{Pause}\,(r\ \texttt{>>=}_{St}\ \lambda\kappa.\,\textbf{return}_{St}\,(\kappa\ \texttt{>>=}\ f))
\end{array} \qquad (*)$$

Here, the bind operator for R is defined recursively using the bind and unit for the state monad (written above as $\texttt{>>=}_{St}$ and $\textbf{return}_{St}$, respectively). Some stateful computation—i.e., within "$r\ \texttt{>>=}_{St}\ldots$"—takes place.

Returning to the trace model example from the beginning of this section, we can now see that R-computations are quite similar to the traces in (‡). The basic resumption monad has lazy constructors *Pause* and *Done* that play the rôle of the lazy list constructors cons (::) and nil ([]) in the traces example. If the atomic operations of a and b are computations of type $\mathit{St}\ ()$, then the following computations of type $R\ ()$ are the set of possible interleavings:

$\mathit{Pause}\ (a_0\ \texttt{>>}\ \textbf{return}\ (\mathit{Pause}\ (a_1\ \texttt{>>}\ \textbf{return}\ (\mathit{Pause}\ (b_0\ \texttt{>>}\ \textbf{return}\ (\mathit{Done}\ ())))))))$
$\mathit{Pause}\ (a_0\ \texttt{>>}\ \textbf{return}\ (\mathit{Pause}\ (b_0\ \texttt{>>}\ \textbf{return}\ (\mathit{Pause}\ (a_1\ \texttt{>>}\ \textbf{return}\ (\mathit{Done}\ ())))))))$
$\mathit{Pause}\ (b_0\ \texttt{>>}\ \textbf{return}\ (\mathit{Pause}\ (a_0\ \texttt{>>}\ \textbf{return}\ (\mathit{Pause}\ (a_1\ \texttt{>>}\ \textbf{return}\ (\mathit{Done}\ ())))))))$

where >> and **return** are the bind and unit operations of the *St* monad. While the stream version implicitly uses a lazy cons operation ($h :: t$), the monadic version uses something similar: $\mathit{Pause}\ (h\ \texttt{>>}\ \textbf{return}\ t)$. The laziness of *Pause* allows infinite *computations* to be constructed in R just as the laziness of cons in ($h :: t$) allows infinite *streams* to be constructed.

3.2 Reactive Concurrency

We now consider a refinement to the R monad allowing computations to signal requests and receive responses in a manner like the interaction between an operating system and processes. Processes executing in an operating system are interactive; processes are, in a sense, in a continual dialog with the operating system. Consider what happens when such a process makes a system call. (1.) The process sends a request signal q to the operating system for a particular action (e.g., a process fork). Making this request may involve blocking the process (e.g., making a request to an I/O device would typically fall into this category) or it may not (e.g., forking). (2.) The OS, in response to the request q, handles it by performing some action(s). These actions may be privileged (e.g., manipulating the process wait list), and a response code c will be generated to indicate the status of the system call (e.g., its success or failure). (3.) Using the information contained in c, the process continues execution.

How might we represent this dialog? Assume we have data types of requests and responses:

$\textbf{data}\ \mathit{Req} = \mathit{Cont} \mid \langle\text{other requests}\rangle$
$\textbf{data}\ \mathit{Rsp} = \mathit{Ack} \mid \langle\text{other responses}\rangle$

Both *Req* and *Rsp* are required to have certain minimal structure; the continue request, *Cont*, signifies merely that the computation wishes to continue, while the acknowledge response, *Ack*, is an information-free response. The following monad, *Re*, "adds" the raw material for interactivity to the monad R as follows:

$$\mathbf{data}\ Re\ a = D\ a \mid P\ (Req, Rsp \to (St(Re\ a)))$$

We coin the term *reactive* resumption to distinguish *Re* from R and use D and P instead of "*Done*" and "*Pause*", respectively. The notion of concurrency provided by *Re* formalizes the process dialog example described above. A paused *Re*-computation has the form $P(q, r)$, where q is a request signal in *Req* and r, if provided with a response code from *Rsp*, is the rest of the computation. The operations for *Re* are defined:

$$\begin{array}{ll}
\mathbf{return} & = D \\
D\ v \mathbin{>\!\!>\!=} f & = f\ v \\
P\ (q, r) \mathbin{>\!\!>\!=} f & = P\ (q,\ \lambda\ rsp\ .\ (r\ rsp) \mathbin{>\!\!>\!=}_{St} \lambda\ \kappa\ .\ \mathbf{return}_{St}\ (\kappa \mathbin{>\!\!>\!=} f))
\end{array}$$

In this article, we use a particular definition of the request and response data types *Req* and *Rsp* which correspond to the services provided by the operating system (more will be said about the use of these in Section 5):

$$\begin{array}{ll}
\mathbf{type}\ Message & = Int \\
\mathbf{type}\ PID & = Int \\
\mathbf{data}\ Req & = Cont \mid Sleep_q \mid Fork_q\ Process \mid Bcst_q\ Message \\
 & \quad \mid Rcv_q \mid V_q \mid P_q \mid Prnt_q\ String \\
 & \quad \mid PID_q \mid Kill_q\ PID \\
\mathbf{data}\ Rsp & = Ack \mid Rcv_r\ Message \mid PID_r\ PID
\end{array}$$

Note that both *Req* and *Rsp* have *Cont* and *Ack*. The kernel in Section 5 will use the response *Ack* for several different requests. *Process* is defined in the next section.

Reactive resumption monads have two non-proper morphisms. The first, *step*, recasts a stateful computation as a resumption computation[3]:

$$\begin{array}{l}
step\ \ :\ St\ a\ \to\ Re\ a \\
step\ x = P\ (Cont,\ \lambda Ack.\ x \mathbin{>\!\!>\!=}_{St} (\mathbf{return}_{St}\ \circ\ D))
\end{array}$$

The definition of *step* shows why we require that *Req* and *Rsp* have a particular shape including *Cont* and *Ack*, respectively; namely, there must be at least one request/response pair for the definition of *step*. Another non-proper morphism for *Re* allows a computation to raise a signal; its definition is:

$$\begin{array}{ll}
sig\ \ :\ Req\ \to\ Re\ Rsp & sig_i\ \ :\ Req\ \to\ Re\ () \\
sig\ q = P(q,\ \mathbf{return}_{St} \circ \mathbf{return}_{Re}) & sig_i\ q = P\ (q, \lambda_.\ \mathbf{return}_{St}(\mathbf{return}_{Re}\ ()))
\end{array}$$

Furthermore, there are certain cases where the response to a signal is intentionally ignored, for which we use sig_i.

[3] For R, *step* is defined similarly: $step\ x = Pause(x \mathbin{>\!\!>\!=}_{St} (\mathbf{return}_{St} \circ Done))$.

4 The Language of Threads

This section formulates an abstract syntax for kernel processes. Operating systems texts typically define threads as lightweight processes executed in the same address space[4]. Events are abstract machine instructions—they read from and write to locations and signal requests to the operating system. Processes are infinite sequences of events, although it is straightforward to include finite (i.e., terminating) processes as well, but it suffices for our presentation to assume non-terminating, infinite processes.

$$
\begin{array}{lll}
\mathit{Process} & = & \mathit{Event}\,;\,\mathit{Process} \\
\mathit{Event} & = & \mathit{Loc}\texttt{:=}\mathit{Exp} \mid \texttt{bcast(}\mathit{Exp}\texttt{)} \mid \texttt{recv(}\mathit{Loc}\texttt{)} \mid \texttt{print(}\mathit{String}\texttt{,}\mathit{Exp}\texttt{)} \\
 & & \mid \texttt{sleep} \mid \texttt{fork(}\mathit{Process}\texttt{)} \mid \texttt{P} \mid \texttt{V} \mid \texttt{kill(}\mathit{Exp}\texttt{)} \\
\mathit{Exp} & = & \mathit{Int} \mid \mathit{Loc} \mid \texttt{pid}
\end{array}
$$

The *Exp* language is self-explanatory except for the `pid` expression that returns the process identifier of the calling process. The *Event* language has a simple assignment statement, l`:=`e, which evaluates its right-hand side, $e \in \mathit{Exp}$, and stores it in the location, $l \in \mathit{Loc}$, on the left-hand side. The language includes broadcast and receive primitives: `bcast(`e`)` and `recv(`l`)`. The event `bcast(`e`)` broadcasts the value of expression e, while `recv(`l`)` receives an available message in location l. There is also a process spawning primitive, `fork(`p`)`, producing a child process p executing in the same address space. The language has a single semaphore with test release operations, `P` and `V`. Finally, there is a process killing primitive, `kill(`*pid*`)`, that terminates the process with identifier *pid* (if such a process exists). Where the language and its semantics differ from previous work [22] is the inclusion of signals; that is, programs may request intervention from the kernel.

Figure 1 defines expressions, events, and processes with $\mathcal{E}[\![-]\!]$, $\mathcal{A}[\![-]\!]$, and $\mathcal{P}[\![-]\!]$, respectively. In most respects, this is a conventional store passing semantics in monadic form, the difference being that individual *St* actions (e.g., *getloc* x) are lifted to *Re* via the *step* function. *step* creates an "atomic" action out of a single *St* action, and $\mathcal{A}[\![-]\!]$ "chains together" one or two such actions. For example, $\mathcal{A}[\![\texttt{P}]\!]$ is the single kernel signal $(sig_i P_q)$, while $\mathcal{A}[\![x\texttt{:=}e]\!]$ chains together "$\mathcal{E}[\![e]\!]$" and "*store* x" with `>>=`. The meaning of a process, $\mathcal{P}[\![p]\!]$, is the infinite "chaining-together" of its event chains. These semantics are similar to published resumption-monadic language semantics [22] for CSP-like languages, differing only in the inclusion of signals (i.e., requests made with with *sig* and sig_i to be handled by the kernel).

5 The Kernel

This section describes the structure and implementation of a kernel providing a variety of services typical to an operating system. For the sake of comprehensibility, we have intentionally made this kernel simple; the goal of the present work

[4] We use the terms "thread" and "process" interchangeably throughout.

$$
\begin{array}{l}
[l{\mapsto}v] : (Loc{\to}Int){\to}Loc{\to}Int \\
[l{\mapsto}i]\ \sigma\ n = \begin{cases} i & l = n \\ \sigma\ n & l \neq n \end{cases} \\
store \quad : \ Loc{\to}Int{\to}Re\ a \\
store\ l\ i = (step{\circ}u)\ ([l{\mapsto}i]) \\
\mathcal{E}[\![-]\!] \quad : \ Exp \ \to \ Re\ Int \\
\mathcal{E}[\![i]\!] \quad = \mathbf{return}\ i \\
\mathcal{E}[\![x]\!] \quad = step\ (getloc\ x) \\
\mathcal{E}[\![\texttt{pid}]\!] = sig\ PID_q\ \texttt{>>=}\ (\mathbf{return} \circ prj) \\
\quad \mathbf{where}\ \ prj\ (PID_r\ pid) \ = \ pid \\
\mathcal{P}[\![-]\!] \quad : \ Process \ \to \ Re\ () \\
\mathcal{P}[\![e; p]\!] = \mathcal{A}[\![e]\!]\ \texttt{>>}\ \mathcal{P}[\![p]\!]
\end{array}
\qquad
\begin{array}{l}
\mathcal{A}[\![-]\!] \quad : \ Event \ \to \ Re\ () \\
\mathcal{A}[\![x\texttt{:=}e]\!] = \ \mathcal{E}[\![e]\!]\ \texttt{>>=}\ store\ x \\
\mathcal{A}[\![\texttt{print(}m\texttt{,}e\texttt{)}]\!] = \ \mathcal{E}[\![e]\!]\ \texttt{>>=}\ print \\
\quad \mathbf{where} \\
\qquad out\ m\ v = m{+\!\!+}\text{“: ”}{+\!\!+}show\ v \\
\qquad print \quad = sig_i \circ Prnt_q \circ (out\ m) \\
\mathcal{A}[\![\texttt{sleep}]\!] \quad = \ sig_i\ Sleep_q \\
\mathcal{A}[\![\texttt{fork(}p\texttt{)}]\!] \ = \ sig_i\ (Fork_q\ p) \\
\mathcal{A}[\![\texttt{bcast(}x\texttt{)}]\!] = \ \mathcal{E}[\![x]\!]\ \texttt{>>=}\ (sig_i \ \circ \ Bcst_q) \\
\mathcal{A}[\![\texttt{recv(}x\texttt{)}]\!] \ = \ sig\ Rcv_q\ \texttt{>>=}\ (store\ x) \circ prj \\
\quad \mathbf{where}\ \ prj\ (Rcv_r\ m) \ = \ m \\
\mathcal{A}[\![\texttt{P}]\!] \quad = \ sig_i\ P_q \\
\mathcal{A}[\![\texttt{V}]\!] \quad = \ sig_i\ V_q \\
\mathcal{A}[\![\texttt{kill(}e\texttt{)}]\!] \ = \ \mathcal{E}[\![e]\!]\ \texttt{>>=}\ (sig_i \ \circ \ Kill_q)
\end{array}
$$

Fig. 1. Semantics of Expressions, Events, and Processes. All monad operations belong to the *Re* monad.

is to demonstrate how typical operating system services may be represented using resumption monads in a straightforward and compelling manner. It should be clear, however, how more powerful or expressive operating system behaviors may be captured as refinements to this system.

The structure of the kernel is given by the global system configuration and two mutually recursive functions representing the scheduler and service handler. The system configuration consists of a snapshot of the operating system resources; these resources are a thread waiting list, a message buffer, a single semaphore, an output channel, and a counter for generating new process identifiers. The system configuration is specified by:

$$
\begin{array}{lll}
\mathbf{type}\ System \ = \ ([(PID, Re\ ())], & \text{— waiting list} \\
\qquad\qquad\quad [Message], & \text{— message buffer} \\
\qquad\qquad\quad Semaphore, & \text{— } Semaphore{=}Int\text{, 1 initially} \\
\qquad\qquad\quad String, & \text{— output channel} \\
\qquad\qquad\quad PID) & \text{— identifier counter}
\end{array}
$$

The first component is the waiting list consisting of a list of pairs: (pid, t). Here, pid is the unique process identifier of thread t. The second component is a message buffer where messages are assumed to be single integers and the buffer itself is a list of messages. Threads may broadcast messages, resulting in an addition to this buffer, or receive messages from this buffer if a message is available. There is a single semaphore, and individual threads may acquire or release this lock. The semaphore implementation here uses busy waiting, although one could readily refine this system configuration to include a list of processes blocked waiting on the semaphore. The fourth component is an output channel (merely a *String*) and the fifth is a counter for generating process identifiers.

The types of a scheduler and service handler are:

$$\begin{aligned} &\mathit{sched} &&: \mathit{System} \to R\,() \\ &\mathit{handler} &&: \mathit{System} \to (\mathit{PID}, \mathit{Re}\,()) \to R\,() \end{aligned}$$

A *sched* morphism takes the system configuration (which includes the waiting list), picks the next thread to be run, and calls the *handler* on that thread. The *sched* and *handler* morphisms translate reactive computations—i.e., those interacting threads typed in the *Re* monad present in the wait list—into a single, interwoven scheduling typed in the basic R monad. The range in the typings of *sched* and *handler* is $R\,()$ precisely because the requested thread interactions have been mediated by *handler*.

From the waiting list component of the system configuration, the scheduler chooses the next thread to be serviced and passes it, along with the system configuration, to the service handler. The service handler performs the requested action and throws the remainder of the thread and the system configuration (possibly updated reflecting the just-serviced request) back to `sched`. The scheduler/handler interaction converts reactive *Re* computations representing threads into a single basic R computation representing a particular schedule.

There are many possible choices for scheduling algorithms—and, hence, many possible instances of *sched*—but for our purposes, round robin scheduling suffices:

$$\begin{aligned} &\mathit{rrobin} : \mathit{System} \to R\,() \\ &\mathit{rrobin}\ ([], \_, \_, \_, \_) &&= \mathit{Done}\,() \quad \text{— stop when no threads} \\ &\mathit{rrobin}\ (((i,t) :: w), mq, s, o, g) &&= \mathit{handler}\ (w, mq, s, o, g)\ (i, t) \end{aligned}$$

The handler fits entirely in Figure 2. A call $(\mathit{handler}\ \mathit{sys}\ (i, P(q, r)))$ responds to query q based on the contents of *sys* and follows the same pattern:

$$P(q, r) \ \to\ \mathit{Pause}(\langle \mathit{St}\ \text{action} \rangle\ ;\ \mathbf{return}_{St}\ (\mathit{rrobin}\ \mathit{sys}'))$$

Here, the "$\langle \mathit{St}$ action$\rangle$" is a (possibly empty) *St* computation determined by r and sys' is the *System* configuration reflecting changes to kernel resources necessary to handling q. Each *handler* branch is discussed in detail below and the labels (a.)-(l.) refer to lines within Figure 2.

Basic Operation. When *handler* encounters a thread which is completed (i.e., the thread is a computation of the form $D\_$), it simply calls the scheduler with the unchanged system configuration (a.). If the thread wishes to continue (i.e., it is of the form $P(\mathit{Cont}, r)$), then *handler* acknowledges the request by passing *Ack* to r (b.). As a result, the first atom in r is scheduled, and the rest of the thread (written $\mathit{next}\ (i, \kappa)$ above) is passed to the scheduler.

Dynamic Scheduling. A thread may request suspension with the Sleep_q signal (c.); the handler changes the Sleep_q request to a *Cont* and reschedules the thread. The effect of this is to delay the thread by one scheduling cycle. An obvious refinement of this service would include a counter field within the Sleep_q request and use this field to delay the thread through multiple cycles.

```
handler : System → (PID, Re ()) → R ()
handler (w, mq, s, o, g) (i, t) =
case t of
(a.)  (D v)              → rrobin (w, mq, s, o, g)
(b.)  (P(Cont, r))       → Pause (r Ack >>=St λκ. returnSt (next (i, κ))
        where
          next t = rrobin (w++[t], mq, s, o, g)
(c.)  (P(Sleep_q, r))    → Pause (returnSt next)
        where
          next = rrobin (w++[(i, P(Cont, r))], mq, s, o, g)
(d.)  (P(Fork_q p, r))   → Pause (returnSt next)
        where
          parent = (i, cont r Ack)
          child  = (g, P⟦p⟧)
          next   = rrobin (w++[parent, child], mq, s, o, g + 1)
(e.)  (P(Bcst_q m, r))   → Pause (returnSt next)
        where
          next = rrobin (w++[(i, cont r Ack)], mq++[m], s, o, g)
(f.)  (P(Rcv_q, r)) | (mq == [])  → Pause (returnSt next)
        where
          next = rrobin (w++[(i, P(Rcv_q, r))], [], s, o, g)
(g.)  (P(Rcv_q, r)) | otherwise   → Pause (returnSt next)
        where
          next = rrobin (w++[(i, cont r (Rcv_r m))], ms, s, o, g)
          m    = head mq
          ms   = tail mq
(h.)  (P(Prnt_q msg, r))  → Pause (returnSt next)
        where
          next = rrobin (w++[(i, P(Cont, r))], mq, s, o++msg, g)
(i.)  (P(P_q, r))         → Pause (returnSt next)
        where
          next      = if s > 0 then goahead else tryagain
          goahead   = rrobin (w++[(i, P(Cont, r))], mq, s − 1, o, g)
          tryagain  = rrobin (w++[(i, P(P_q, r))], mq, s, o, g)
(j.)  (P(V_q, r))         → Pause (returnSt next)
        where
          next = rrobin (w++[(i, cont r Ack)], mq, s + 1, o, g)
(k.)  (P(PID_q, r))       → Pause (returnSt next)
        where
          next = rrobin (w++[(i, cont r (PID_r i))], mq, s, o, g)
(l.)  (P(Kill_q j, r))    → Pause (returnSt next)
        where
          next          = rrobin (wl', mq, s, o, g)
          wl'           = filter (exit j) (w++[(i, cont r Ack)])
          exit i (pid, t) = pid /= i

cont      : (Rsp → St (Re a)) → (Rsp → Re a)
cont r rsp = P (Cont, λAck. r rsp)
```

Fig. 2. The Request Handler

A thread may request to spawn a new thread (d.). The child process is $(g, \mathcal{P}[\![p]\!])$ for new identifier g. Then, both parent and child thread are added back to the waiting list. We introduce the "continue" helper function, *cont*, that takes a partial thread, r, and a response code, *rsp*, and creates a thread which receives and continues on the response code *rsp*. Another useful service (á la Unix `fork` system call) would include a response $Fork_r\ Bool$ in *Rsp* to distinguish child and parent processes.

Asynchronous Message Passing. For a thread to broadcast m (e.), the message is simply appended to the message queue. When a Rcv_q signal occurs and the message queue is empty, then the thread must wait (f.) and so is put back on the thread list. Note that, rather than busy-waiting for a message, the message queue could contain a "blocked waiting list" for threads waiting for the arrival of messages, and, in that scenario, the handler could wake a blocked process whenever a message arrives. If there is a message m in the message queue, then it is passed to the thread (g.).

Printing. When a print request $(Prnt_q\ msg)$ is signaled (h.), then the string *msg* is appended to the output channel *out* and the rest of the thread, $P(Cont, r)$, is passed to the scheduler. An alternative could use the "interactive output" monad formulation for R: $R\ A = \mu X.\ (A + (String \times S\ X))$ instead of encoding the output channel as the string o.

Synchronization Primitives. Requesting the system semaphore (i.) will succeed if $s > 0$, in which case the requesting thread will continue with the semaphore decremented; if $\not> 0$, the requesting thread will suspend. These possible outcomes are bound to *goahead* and *tryagain* in the following *handler* clause, and *handler* chooses between them based on the current value of s: Note that this implementation uses busy waiting merely for simplicity's sake. One could easily implement more efficient strategies by including a queue of waiting threads with the semaphore. A thread may release the semaphore (j.) without blocking. Note this semaphore is *general* rather than *binary*, meaning that the counter s may have as its value any non-negative integer rather than just 0 or 1.

Process Id Request. A thread may request its identifier i (k.), which is simply passed to it in $cont\ r\ (PID_r\ i)$.

Preemption. One thread may preempt another by sending it a kill signal reminiscent of the Unix `(kill -9)` command; this is implemented by the *handler* declaration at line (l.). Upon receiving the signal $Kill_q\ j$, the thread with process identifier j (if one exists) is removed from the waiting list using the Haskell built-in function *filter* : $(a \to Bool) \to [a] \to [a]$. In a call $(filter\ b\ l)$, *filter* returns those elements of list l on which b is true (in order of their occurrence in l).

Time Behavior of `>>=`$_R$ and `>>=`$_{Re}$. Because the bind operations for R and Re are both $O(n)$ in the size of their first arguments, one can write programs that, through the careless use of the bind, end up with quadratic (or worse) time complexity. Note, however, the kernel avoids this entirely by relying on co-recursion in the definition of *handler*.

Executing the kernel. An R computation may be run by projecting it to St with $run\ :\ R\ a \rightarrow St\ a$ defined by

$$run\ (Done\ v)\ =\ \mathbf{return}_{St}\ v \qquad run\ (Pause\ \varphi)\ =\ \varphi\ \texttt{>>=}_{St}\ run$$

Running the kernel on initial processes $p_1, \ldots, p_n$ is accomplished with

$$run\ (rrobin\ ([\mathcal{P}[\![p_1]\!], \ldots, \mathcal{P}[\![p_n]\!]], [], 1, \texttt{""}, 0))$$

Sample executions are provided in the code base [13].

6 Conclusions

As of this writing, resumptions as a model of concurrency have been known for thirty years and, in monadic form, for almost twenty. Yet, unlike other techniques and structures from language theory (e.g., continuations, type systems, etc.), resumptions have evidently never found wide-spread acceptance in programming practice. This is a shame, because resumptions—especially in monadic form—are a natural and beautiful organizing principle for concurrent applications: they capture exactly what one needs to write and think about multitasking programs—and no more! Resumptions capture precisely the intuition that threads are potentially infinite sequences of atoms interacting according to some discipline. The framework presented here has been applied in a number of diverse settings and expresses a broad sampling of concurrent behaviors. This is solid evidence that resumptions express the true essence and structure of multitasking computation.

Although the kernel is, of necessity, simple, it does demonstrate both the wide scope of concurrent behaviors expressible with resumption monads and the ease with which such behaviors may be expressed. To be sure, more efficient implementations and realistic features may be devised (e.g., by eliminating busy-waiting). As each of the three monads may be generalized as monad transformers, instances of this kernel inherit the software engineering benefits of monad transformers that one would expect—namely, modularity, extensibility, and reusability. Such kernel instances may be extended by either application of additional monad transformers or through refinements to the resumption monad transformers themselves. Such refinements are typically straightforward; to add a new service to the kernel of Section 5, for example, one merely extends the *Req* and *Rsp* types with a new request and response and adds a corresponding *handler* definition. The kernel in Section 5 may, in fact, be viewed as the result of multiple extensions to a core "basic operation" kernel (i.e., one having only a *Cont* request).

The framework developed here has been applied to such seemingly diverse purposes as language-based security [15] and bioinformatics [14]; each of these applications is an instance of this framework. The difference is evident in the request and response data types *Req* and *Rsp*. Consider the subject of [14], which is the formal modeling of the life cycles of autonomous, intercommunicating cellular systems using domain-specific programming languages. Each cell has some collection of possible actions describing its behavior with respect to itself

and its environment. The actions of the photosynthetic bacterium *Rhodobacter Sphaeroides* are reflected in the request and response types:

data *Req* = *Cont* | *Divide* | *Die* | *Sleep* | *Grow* |*LightConcentration*
data *Rsp* = *Ack* | *LightConcRsp Float*

Each cell may undergo physiological change (e.g., cell division) or react to its immediate environment (e.g., to the concentration of light in its immediate vicinity). The kernel instance here also maintains the physical integrity of the model.

The kernel presented here confronts many "impurities" considered difficult to accommodate within a pure, functional setting—concurrency, state, and i/o—which are all members of the so-called "Awkward Squad" [24]. In Haskell, these real world impurities are swept, in the memorably colorful words of Simon Peyton Jones, into a "giant sin-bin" called the *IO* monad[5]. But is *IO* truly a monad (i.e., does it obey the monad laws)? All of these impurities have been handled individually via various monadic constructions (consider the manifestly incomplete list [21, 26]) and the current approach combines some of these constructions into a single monad. While it is not the intention of the current work to model the awkward squad as it occurs in Haskell, the techniques and structures presented here point the way towards such models.

References

1. D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunder, S. Nettles, and J. Smith. The switchware active network architecture. *IEEE Network*, May/June 1998.
2. J.W. de Bakker and E.P. de Vink. *Control Flow Semantics*. Foundations of Computing Series. The MIT Press, 1996.
3. D. Carter. *Deterministic Concurrency*. PhD thesis, Department of Computer Science, University of Bristol, January 1994.
4. K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.
5. J. Cupitt. *The Design and Implementation of an Operating System in a Functional Language*. PhD thesis, Computing Laboratory, Univ. Kent, October 1992.
6. D. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.
7. A. Filinski. Representing monads. In *Proceedings of POPL '94*, 1994.
8. A. Filinski. Representing layered monads. In *Proceedings of POPL '99*, 1999.
9. M. Flatt, R. Findler, S. Krishnamurthi, and M. Felleisen. Programming languages as operating systems (or revenge of the son of the lisp machine). In *Proceedings of ICFP'99*, pages 138–147, 1999.
10. S. Ganz, D. Friedman, and M. Wand. Trampolined style. In *Proceedings of ICFP'99*, pages 18–27, 1999.
11. Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*, pages 317–331, London, UK, 2000. Springer-Verlag.

[5] See his slides "*Lazy functional programming for real: Tackling the Awkward Squad*" available at `research.microsoft.com/Users/simonpj/papers/marktoberdorf/Marktoberdorf.ppt`.

12. R. Harper, P. Lee, and F. Pfenning. The Fox project: Advanced language technology for extensible systems. Technical Report CMU-CS-98-107, CMU, 1998.
13. W. Harrison. The Essence of Multithreading Codebase. Available from `www.cs.missouri.edu/~harrison/EssenceOfMultitasking`.
14. W. Harrison and R. Harrison. Domain specific languages for cellular interactions. In *Proceedings of the 26th Annual IEEE International Conference on Engineering in Medicine and Biology*, September 2004.
15. W. Harrison and J. Hook. Achieving information flow security through precise control of effects. In *18th IEEE Computer Security Foundations Workshop (CSFW05)*, June 2005.
16. P. Henderson. Purely functional operating systems. In *Functional Programming and Its Applications: an Advanced Course*, pages 177–191. Cambridge University Press, 1982.
17. B. Jacobs and E. Poll. Coalgebras and Monads in the Semantics of Java. *Theoretical Computer Science*, 291(3):329–349, 2003.
18. S. Krstic, J. Launchbury, and D. Pavlovic. Categories of processes enriched in final coalgebras. In *Proceedings of FOSSACS'01*, pages 303–317, 2001.
19. J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Proceedings of PLDI'94*, pages 24–35, 1994.
20. S. Liang. *Modular Monadic Semantics and Compilation*. PhD thesis, Yale University, 1998.
21. E. Moggi. An Abstract View of Programming Languages. Technical Report ECS-LFCS-90-113, Department of Computer Science, Edinburgh University, 1990.
22. N. Papaspyrou. A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. In *Proceedings of the 3rd Panhellenic Logic Symposium*, 2001. An expanded technical report is available from the author by request.
23. N. Papaspyrou and D. Maćoš. A Study of Evaluation Order Semantics in Expressions with Side Effects. *Journal of Functional Programming*, 10(3):227–244, May 2000.
24. S. Peyton Jones. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell. In *Engineering Theories of Software Construction*, pages 47–96. 2000.
25. S. Peyton Jones, editor. *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, 2003.
26. S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of POPL'93*, pages 71–84, 1993.
27. G. Plotkin. A Powerdomain Construction. *SIAM Journal of Computation*, 5(3):452–487, 1976.
28. W. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall, 1998.
29. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Boston, 1986.
30. E. Spiliopoulou. Concurrent and Distributed Functional Systems. Technical Report CS-EXT-1999-240, Univ. Bristol, 1999.
31. W. Stoye. Message-based Functional Operating Systems. *Science of Computer Programming*, 6(3):291–311, 1986.
32. D. Turner. An Approach to Functional Operating Systems. In *Research Topics in Functional Programming*, pages 199–217. 1990.
33. A. van Weelden and R. Plasmeijer. Towards a strongly typed functional operating system. In *Proceedings of IFL'02*, volume 2670 of *LNCS*, 2002.
34. M. Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, pages 19–28. ACM Press, 1980.