Cheap (But Functional) Threads

WILLIAM L. HARRISON
Department of Computer Science
University of Missouri
Columbia, Missouri

Abstract

This article demonstrates how a powerful and expressive abstraction from concurrency theory plays a dual rôle as a programming tool for concurrent applications and as a foundation for their verification. This abstraction—monads of resumptions expressed using monad transformers—is *cheap*: it is easy to understand, easy to implement, and easy to reason about. We illustrate the expressiveness of the resumption monad with the construction of an exemplary multitasking operating system kernel with process forking, preemption, message passing, and synchronization constructs in the pure functional programming language Haskell. Because resumption computations are stream-like structures, reasoning about this kernel may be posed as reasoning about streams, a problem which is well-understood. And, as an example, this article illustrates how a liveness property—fairness—is proven and especially how the resumption-monadic structure promotes such verifications.

1 "Cheap" Concurrency

This paper presents a low overhead approach to multi-threaded concurrent computation. We demonstrate how a rich variety of concurrent behaviors—including synchronization, message passing, the forking and suspension of threads among others—may be expressed succinctly in a non-strict functional language with no changes to the host language implementation. The necessary machinery—monads of resumptions—is so expressive, in fact, that the kernel described here requires fewer than fifty lines of Haskell 98 code. And, because this machinery may be generalized as monad transformers, the functionality described here may be reused and refined easily.

Many techniques and structures have emigrated from programming language theory to programming practice (e.g., types, CPS, etc.), and this article advocates that resumption monads make the journey as well. This work is not intended to be theoretical; on the contrary, its purpose is to demonstrate how a natural (but, perhaps, under-appreciated) computational model of concurrency is used to construct and verify concurrent applications. The approach taken here is informal and should be accessible to anyone familiar with monads in Haskell[1]. And while the constructions

[1] All the code presented in this paper is available from the author.

described here occur in Haskell 98, they may be transcribed in a straightforward manner into any functional programming language—even strict ones like ML or Scheme (Filinski, 1999; Espinosa, 1995).

A *resumption* (Plotkin, 1976) 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 (read: *cheaper*) than continuations—the *only* thing one may model with resumptions is interleaved computation. This conceptual economy makes concurrent applications structured with resumption monads easier to comprehend, modify, extend, and reason about. While techniques for concurrency based on continuation-passing style (CPS) are well-known (Wand, 1980; Haynes & Friedman, 1984; Claessen, 1999; Flatt *et al.*, 1999), programs in CPS-style are notoriously difficult to verify precisely because of the immense expressive power of continuations. Resumption monads are both an expressive programming tool for concurrent applications and a foundation for their verification.

The genesis of this work is the design, implementation, and verification of a secure operating system kernel in the pure, lazy functional language Haskell. The Programatica project[2] at OGI is working to develop and formally verify *OSKer* (Oregon Separation Kernel), a kernel for multi-level secure applications. Functional languages, particularly of the non-strict variety, are well-known for promoting mathematical reasoning about programs, and, perhaps because of this, there has been considerable research in the use of functional languages for operating systems and systems software. 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 interleaving concurrency.

The structure of this article is as follows. After reviewing the related work and necessary background in Sections 2 and 3, Section 4 describes in detail how resumption monads may be used to model interleaving concurrency. Two varieties of resumption computation are required here; one for thread scheduling (referred to here as *basic* resumptions) and another form that gives a precise account of our notion of thread (called *reactive* resumptions). Section 4 describes both forms in detail. Section 5 presents a resumption-monadic semantics for a CSP-like concurrent language extended with "signals"; a thread may signal a request to fork, suspend, print, send or receive a message, or acquire or release a semaphore. This language is a convenient abstract syntax for threads, and also serves to connect the previous research on resumption-based language semantics to the research reported here. Specifically, with a small change to the kernel (namely, incorporation of the non-determinism monad), a definitional interpreter can be given that recovers a typical resumption-based denotational semantics for the language. The appendix elaborates on this interpreter and its connection to previous applications of the resumption monad to language semantics. Section 6 describes the kernel itself, which consists of mutually recursive scheduling and service handler functions. Section 7

[2] The Programatica project at OGI/PacSoft explores the use of Haskell 98 in formal methods. For more information, please consult the project webpage `www.cse.ogi.edu/PacSoft/projects/programatica`.

presents the specification and verification of a liveness property (fairness) for the kernel from Section 6. Here the resumption-monadic structuring of the kernel pays dividends: because resumption computations most resemble streams or traces of events, the fairness verification remains elementary in the sense that it retains the flavor of stream-based reasoning.

2 Related Work

The concurrency models underlying previous applications of functional languages to concurrent system software fall broadly into four camps. The first camp (Henderson, 1982; Stoye, 1984; Stoye, 1986; Turner, 1987; Turner, 1990; Cupitt, 1992) 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; please see the appendix for further details.

The second model uses "demand-driven concurrency" (Carter, 1994; Spiliopoulou, 1999) in which threads are mutually recursive bindings whose lazy evaluation simulates interleaving 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 (Launchbury & Peyton Jones, 1994). 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; Section 6 describes such a deterministic scheduler in detail.

The third camp uses CPS to implement thread interleaving. Concurrent behavior may be modeled with first-class continuations (Claessen, 1999; Wand, 1980; Lin, 1998; Flatt *et al.*, 1999; van Weelden & Plasmeijer, 2002) because the explicit control over evaluation order in CPS allows multiple threads to be "interwoven" to produce any possible execution order. Claessen presents an elegant formulation of this style using the CPS monad transformer (Claessen, 1999), although apparently 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. Continuations are, as one wag put it, the "atom bomb" of programming language semantics as most known effects may be expressed via CPS (Filinski, 1996; Filinski, 1994). Programs in CPS are notoriously difficult to reason about perhaps *because* of this expressiveness, and this fact renders CPS less attractive as a foun-

dation 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 (Ganz *et al.*, 1999). 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 (Ganz *et al.*, 1999), the authors motivate trampolining with a type constructor equivalent to the functor part of the basic resumption monad (described in Section 4.1 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 (Harper *et al.*, 1998; Birman *et al.*, 2000; Alexander *et al.*, 1998); as the modeling of concurrency is not their primary concern, no further comparison is made. Similarly, there are many concurrent functional languages (Plasmeijer & van Eekelen, 1998; Peyton Jones *et al.*, 1996; Cooper & Morrisett, 1990; Armstrong *et al.*, 1996), 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 (Plotkin, 1976; Schmidt, 1986; Bakker & Vink, 1996), although this formulation of resumptions did not involve monads. 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 (Moggi, 1990). In his initial development, Moggi showed that most known semantic effects could be naturally expressed monadically. He also showed how a sequential theory of concurrency could be expressed in the resumption monad.

Both the basic and reactive formulations of the resumption monad and monad transformer first occur in the work of Moggi (Moggi, 1990). Espinosa presents a version of the basic resumption monad transformer in his thesis (Espinosa, 1995). The basic resumption monad transformer used here is due to Papaspyrou, et al. (Papaspyrou, 1998; Papaspyrou & Maćoš, 2000; Papaspyrou, 2001). Papaspyrou has made extensive use of this monad transformer in his research; this includes a monadic-denotational semantics of ANSI C (Papaspyrou, 1998), an elegant study of the relationship between side effects and evaluation order (Papaspyrou & Maćoš, 2000), and a monadic-denotational semantics of a non-deterministic CSP-like language (Papaspyrou, 2001). The formulation of reactive resumptions used here occurs (unsurprisingly) first in Moggi's work (Moggi, 1990). It is later mentioned in passing by Filinski (Filinski, 1999), although in a form appropriate for the strict language ML. In (Filinski, 1999), the author presents an example of shared-state concurrency implemented with basic resumptions.

Much of the literature involving resumption monads (Moggi, 1990; Papaspyrou, 1998; Papaspyrou & Maćoš, 2000; Papaspyrou, 2001; Krstic *et al.*, 2001; Jacobs & Poll, 2003) focuses on their use in elegant and abstract categorical semantics for programming languages. The current work advocates the use of resumption monads as a useful abstraction for concurrent programming and verification rather than as a basis for denotational semantics. The purpose of this account, in part, is to provide an exposition on resumption monads so that the interested reader may grasp the literature more readily. To this end, when resumption monads are introduced in Section 4, the constructions are presented both as Haskell data type declarations and in the categorical notation of Moggi's lecture notes (Moggi, 1990), so that the reader may compare the two formalisms side-by-side. And, although semantics is not the primary concern here, the kernel construction is not far off from resumption-monadic language semantics. A definitional interpreter for a small concurrent language of threads is given in Section 5, and the appendix outlines how this interpreter, when combined with a non-deterministic version of the kernel, closely resembles a published semantics (Papaspyrou, 2001).

Monads were introduced into functional languages as a means of dealing with "impurities" such as exceptions and state (Wadler, 1992). All Haskell implementations use the *IO* monad for impure actions; for example, printing out a character is performed by $putChar :: Char \rightarrow IO()$. There is also a version of Haskell with explicit concurrency—namely, Concurrent Haskell (Peyton Jones *et al.*, 1996)—with special constructs supporting shared-state concurrency; these are typed within the Haskell *IO* monad as well: $forkIO :: IO() \rightarrow IO()$ spawns a new thread executing for its argument.

But what is *IO*? In the memorably colorful words of Simon Peyton Jones, *IO* is a "giant sin-bin", into which all the impure aspects necessary for real world programming are swept[3]. But is *IO* truly a monad (i.e., does it obey the monad laws)? What is the precise behavior of combinators for the impurities such as concurrency or mutable state gathered together in *IO*? In short, how does one prove properties of programs of type *IO*? The fact is the structure of *IO* is opaque, and so without closely examining the source code of existing Haskell implementations, one simply could not answer such questions. And even with the source code, the situation does not improve much as there are multiple Haskell implementations, designed for efficiency rather than for the needs of formal analysis.

The kernel design presented here confronts many "impurities" considered difficult to accommodate within a pure, functional setting—concurrency, state, and input/output—which are all members of the so-called "Awkward Squad" (Peyton Jones, 2000). All of these impurities have been handled individually via various monadic constructions (consider the manifestly incomplete list (Moggi, 1990; Peyton Jones & Wadler, 1993; Papaspyrou, 1998)). The current approach combines these individual constructions into a single *layered* monad—i.e., a monad created

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

from monadic building blocks known as monad transformers (Liang, 1998; Espinosa, 1995). While it is not the intention of the current work to model either the Haskell *IO* monad or Concurrent Haskell, the techniques and structures presented here point the way towards such models.

Temporal logics (Pnueli, 1977; Emerson, 1990; Manna & Pnueli, 1991) are modal logics typically applied to the specification and verification of concurrent systems and algorithms. They contain modalities such as "eventually" that allow straightforward specifications of desirable system properties; for example, the fairness of process scheduling (i.e., "eventually all runnable processes execute") may be directly formulated in temporal logic (Manna & Pnueli, 1991). Two notions of time—*linear* and *branching*—distinguish temporal logics (Emerson, 1990). *Linear* temporal logics[4] allow that, at any moment in time, there is only one "next" moment. The kernel described in Section 6 is linear in this sense; because the kernel has deterministic scheduling, there is only one possible next step. The model of time underlying *branching* temporal logics allow for more than one "next" moment; time may split or "branch" into distinct possible futures. The kernel described in Appendix B supports a finitely branching notion of time.

3 Monads and Monad Transformers

This section serves as a review of monads and monad transformers (Moggi, 1990; Liang, 1998) with emphasis on how they are represented in Haskell[5]. Section 3.1 reviews the Haskell syntax for monads as well as the so-called *monad laws*. Readers familiar with this material may choose to skip this section. Monads can encapsulate program effects such as state, exceptions, multi-threading, environments, and CPS; combining such effects into a single monad may be achieved using *monad transformers* (Moggi, 1990; Liang, 1998), where each effect corresponds to an individual monad transformer. Section 3.1 also presents two basic monads relevant to this work—the identity and state monads. Section 3.2 reviews the basic ideas behind monad transformers, introducing the state monad transformer as an example. Monads constructed with monad transformers are easily combined and extended, and monadic programs inherit this modularity, as is also described in Section 3.2.

3.1 Monads in Haskell

A monad consists of a type constructor M and two functions:

$$\begin{array}{lll} \textbf{return} :: a \to M\ a & & \text{— the "unit" of } M \\ (\texttt{>>=}) \quad :: M\ a \to (a \to M\ b) \to M\ b & & \text{— the "bind" of } M \end{array}$$

The **return** operator is the monadic analogue of the identity function, injecting a value into the monad. The `>>=` operator gives a form of sequential application.

[4] These are not to be confused with *linear logic* (Girard, 1987).
[5] For readers needing a more thorough introduction, we recommend Liang, Hudak, and Jones (Liang *et al.*, 1995).

These operators must satisfy the monad laws:

$$\begin{array}{lll}
(\mathbf{return}\ v)\ \texttt{>>=}\ k & = k\ v & \text{— left unit} \\
x\ \texttt{>>=}\ \mathbf{return} & = x & \text{— right unit} \\
x\ \texttt{>>=}\ (\lambda\ a.(k\ a)\ \texttt{>>=}\ h) & = (x\texttt{>>=}\ k)\ \texttt{>>=}\ h & \text{— associativity}
\end{array}$$

Haskell syntax (Peyton Jones, 2003) has several alternative forms for the bind operator. The "null bind" operator, >>, is defined as $x\ \texttt{>>}\ k\ =\ x\ \texttt{>>=}\ \lambda\_.\ k$; it ignores the value produced by x. The "do" notation is sometimes easier to read than a sequence of nested binds: $\mathbf{do}\ \{\ v\ \leftarrow\ x\ ;\ k\ \}\ =\ (x\ \texttt{>>=}\ \lambda\ v.\ k)$.

3.1.1 The Identity Monad

Defining a monad in Haskell typically consists of declaring a data type and an instance of the built-in *Monad* class (Peyton Jones, 2003). For example, the identity monad is declared as:

$$\begin{array}{ll}
\mathbf{data}\ Id\ a\ =\ Id\ a & \mathbf{instance}\ Monad\ Id\ \mathbf{where} \\
deId\ (Id\ x)\ =\ x & \quad return\ v\ \ \ \ = Id\ v \\
& \quad (Id\ x)\ \texttt{>>=}\ f = f\ x
\end{array}$$

The data type declaration defines the computational "raw materials" encapsulated by the monad (the *Id* monad provides no such materials). The variables **return** and >>= are overloaded in Haskell and the instance declaration *Monad Id* gives their meaning for the type constructor *Id*. Note, however, that Haskell does not guarantee that such declarations obey the monad laws.

3.1.2 The State Monad

The following defines the well-known state monad in Haskell, assuming for the sake of this exposition, that the state is simply the Haskell type *Int*:

$$\begin{array}{l}
\mathbf{data}\ St\ a\ \ \ \ =\ ST\ (Int\ \rightarrow\ (a, Int)) \\
deST\ (ST\ \varphi) = \varphi \\
\mathbf{instance}\ Monad\ St\ \mathbf{where} \\
\quad \mathbf{return}\ v\ \ \ \ = ST\ (\lambda\ s.\ (v, s)) \\
\quad (ST\ \varphi)\ \texttt{>>=}\ f = ST\ (\lambda\ s_0.\ \mathbf{let}\ (v_1, s_1)\ =\ \varphi\ s_0\ \mathbf{in}\ deST\ (f\ v_1)\ s_1)
\end{array}$$

Monads typically have *non-proper* morphisms to manipulate their extra computational "raw material"; they are so-called because they are not the monad's "proper" morphisms (namely, >>= and **return**). The state monad possesses a store, passed in a single-threaded manner throughout a computation. Operators are defined to make use of this store; they are the "get store" operator, g, and the "update store" operator, u:

$$\begin{array}{ll}
g\ ::\ St\ Int & u\ \ \ ::\ (Int\ \rightarrow\ Int)\ \rightarrow\ St\ a \\
g = ST\ (\lambda\ s.\ (s, s)) & u\ \delta = ST\ (\lambda\ s.\ (nil, \delta\ s))
\end{array}$$

where $nil :: a$ is defined as $nil = undefined$, where $undefined\ ::\ a$ is a special Haskell

constant playing the same rôle as $\perp$ in denotational semantics. The *undefined* constant is returned by $(u\ \delta)$ to indicate that its return value is content-free. The g computation returns the current store and $(u\ \delta)$ applies store transformer δ to the current store.

3.2 Layered Monads & Monad Transformers

Monad transformers are generalizations of monads, each isolating a particular effect. There are several formulations of monad transformers, and we follow that given in (Liang, 1998; Liang *et al.*, 1995). A monad transformer consists of two data: a type constructor T, and a definition of the monad $(T\ M)$ in terms of any existing monad M. Section 3.2.1 presents the state monad transformer, and later in Section 4 introduces monad transformers for resumptions.

Monad transformers allow monads to be constructed in a "layered" fashion; if T_i are monad transformers and M is a monad, then $M' = T_1\ (\ldots\ (T_n\ M)\ldots)$ is also a monad. Such monads are called henceforth *layered* monads. A useful property of the layered approach is that the monad laws need not be checked for M'—it is known by construction to be a monad (Liang, 1998). Note that any non-proper morphisms arising in the intermediate layers of M' must be redefined for M'. That is, if M (or $T_n\ M$, etc.) has non-proper morphisms, they must be redefined with each transformer application. The general issue of whether a non-proper morphism can be redefined in a behavior-preserving manner—called *lifting*—involves a slight restriction on the order of application of monad transformers when the CPS monad transformer is included. None of these issues arise in the monads constructed in this article, so we elaborate this point no further, but the interested reader should consult the references[6].

Layered monads implement a particular signature, where a signature is a collection of typed, uninterpreted function symbols (Loeckx *et al.*, 1996). The *signature* of any layered monad includes its unit and bind as well as its non-proper morphisms. Considering the state monad for example, its signature consists of the bind and unit and the u and g operators. The essence of the modularity of the layered monadic approach is that signatures may be re-interpreted at different monads and that monad transformers give a canonical means of constructing such re-interpretations in a manner which preserves the behavior of the signature functions.

3.2.1 The State Monad Transformer

The state monad transformer generalizes the state monad. It takes two type parameters as input—the type constructor m representing an existing monad and a store type *sto*—and from these createsa monad adding single-threaded *sto*-passing to the

[6] Especially, consider the dissertations of Espinosa and Liang (Espinosa, 1995; Liang, 1998).

computational raw material of m. Using the bind and return of m, the bind and return of the new monad, $(StateT\ sto\ m)$, are defined in an instance declaration:

$$\begin{array}{l}
\textbf{data}\ (Monad\ m)\ \Rightarrow\ StateT\ sto\ m\ a\ =\ ST\ (sto\ \rightarrow\ m\ (a, sto)) \\
deST\ (ST\ x)\ =\ x \\
\textbf{instance}\ Monad\ m\ \Rightarrow\ Monad\ (StateT\ sto\ m)\ \textbf{where} \\
\quad \textbf{return}\ v\ \ \ \ \ = ST\ (\lambda\ s.\ \textbf{return}_m\ (v, s)) \\
\quad (ST\ x)\ \texttt{>>=}\ f = ST\ (\lambda\ s_0.\ (x\ s_0)\ \texttt{>>=}_m\ \lambda\ (y, s_1).\ deST\ (f\ y)\ s_1)
\end{array}$$

The bind and return of the input monad m are distinguished from those being defined by attaching a subscript (e.g., $\textbf{return}_m$). The monad $St\ =\ StateT\ Int\ Id$ is isomorphic to the previous, unlayered state monad of Section 3.1.2.

Notational Convention. In Haskell 98, monadic operators are overloaded, and, although the Haskell type class system would resolve the apparent ambiguity in the text of the definitions, such overloading may make the Haskell code confusing to read. As a notational convention adopted throughout this article, we eliminate such ambiguities by subscripting the bind and return operators when it seems helpful.

The state monad's non-proper morphisms may be generalized as well:

$$\begin{array}{ll}
g\ ::\ Monad\ m \Rightarrow StateT\ s\ m\ s & u\ ::\ Monad\ m \Rightarrow (s \rightarrow s) \rightarrow StateT\ s\ m\ a \\
g\ =\ ST\ (\lambda\ s.\ \textbf{return}_m\ (s, s)) & u\ \delta\ =\ ST\ (\lambda\ s.\ \textbf{return}_m\ (nil, \delta\ s))
\end{array}$$

In store-passing semantics of imperative programs (Schmidt, 1986), the store is typically modeled by a type of the form $Name \rightarrow Value$, where $Value$ refers the a type of storable values. Later in Section 5, we give a store-passing semantics in monadic form for an imperative language of threads in which the store is of the form $Name \rightarrow Integer$. For the monad transformer $(StateT\ (Name \rightarrow Integer))$, a useful morphism $getloc$ may be defined in terms of g as:

$$\begin{array}{l}
getloc\ ::\ (Monad\ m)\ \Rightarrow\ Name\ \rightarrow\ StateT\ (Name \rightarrow Integer)\ m\ Integer \\
getloc\ x\ =\ g\ \texttt{>>=}\ \lambda\ \sigma.\ \textbf{return}\ (\sigma\ x)
\end{array}$$

An application $getloc\ x$ returns the contents of location x. The above type signature means that $getloc$ may be defined for any monad m by applying the aforementioned state monad transformer. We may define a morphism which projects a stateful computation to a value:

$$\begin{array}{l}
run\ ::\ StateT\ sto\ Id\ a\ \rightarrow\ sto\ \rightarrow\ (a, sto) \\
run\ (ST\ \varphi)\ \sigma\ =\ deId\ (\varphi\ \sigma)
\end{array}$$

The effect of $(run\ (ST\ \varphi)\ \sigma)$ threads the initial store σ through stateful computation φ, producing a result value and store (v, σ'). The interaction of run with the operations of St is given by the following theorem:

Theorem 1 (*run-rules*)

(i) $run\ (\textbf{return}_{St}\ v)\ \sigma\ =\ (v, \sigma)$

(ii) $run\ (x\ \texttt{>>=}_{St}\ f)\ \sigma\ =\ run\ (f\ v)\ \sigma'$ where $(v, \sigma')\ =\ run\ x\ \sigma$

(iii) $run\ (u\ \delta)\ \sigma\ =\ (nil, \delta\ \sigma)$

(iv) $run\ g\ \sigma\ =\ (\sigma, \sigma)$

This theorem is proved equationally in a straightforward manner from the definition of $StateT$ and so we omit the proof here.

4 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 (Moggi, 1990; Papaspyrou, 1998; Papaspyrou & Maćoš, 2000; Papaspyrou, 2001; Espinosa, 1995; Filinski, 1999). The *basic* resumption monad (Section 4.1) encapsulates a notion of interleaving 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 4.2) encapsulates interleaving 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.

A natural model of concurrency is the "trace model" (Roscoe, 1998). 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. To illustrate this model, consider the following example. 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. Then, according to the trace model, the concurrent execution of threads a and b, $a \| b$, is denoted by the set of all their possible interleavings[7]; these are:

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

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 succinctly and is therefore well-suited to formal characterizations of properties of concurrent systems (e.g., liveness properties of schedulers such as fairness). However, a wide gap exists between this formal model and an executable system: traces are simply lists of events, and each event is itself merely 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 a formal, trace-based concurrency model that may be directly realized as executable Haskell code.

As we shall see in this section, the notion of computation provided by resumption monads is that of sequenced computation. A resumption computation has a list-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)—in this way, it is very much like the execution traces in (‡). We now describe the two forms of resumption monads in detail.

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

4.1 Sequenced Computation & Basic Resumptions

This section introduces sequenced computation in monadic style. First, the most elementary resumption monad is constructed as a Haskell data type declaration. Unlike other well-known monads such as state, resumptions as a "stand alone" monad are not interesting at all, so we then discuss the monad that combines resumptions with state. Finally, we describe the generalization of the monad of basic resumptions as a monad transformer.

The most basic resumption monad contains only a notion of sequencing and nothing else:

$$\textbf{data}\ R\ a\ =\ \mathit{Done}\ a \mid \mathit{Pause}\ (R\ a)$$

An R-computation is either a completed computation, $(\mathit{Done}\ v)$, a finite sequence of *Pause*'s ending in a *Done*, $(\mathit{Pause}(\ldots\mathit{Pause}(\mathit{Done}\ v)\ldots))$, or an infinite sequence of *Pause*'s, $(\mathit{Pause}(\ldots))$. So, we can see that this version of the resumption monad is not terribly interesting; computations in R resemble streams without stream elements. The return operation for R is *Done*, while the bind operation (`>>=`), in effect, appends two R-computations; the Haskell declaration for this monad is:

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ R\ \textbf{where}\\ \quad\begin{array}{ll}\textbf{return} & = \mathit{Done}\\ (\mathit{Done}\ v)\ \texttt{>>=}\ f & = f\ v\\ (\mathit{Pause}\ r)\ \texttt{>>=}\ f & = \mathit{Pause}\ (r\ \texttt{>>=}\ f)\end{array}\end{array}$$

Note that the bind operator for R, `>>=`, is defined recursively.

Combining the monad of resumptions with the monad of state, the resulting monad is much more expressive and useful:

$$\begin{array}{l}\textbf{data}\ R\ a\ =\ \mathit{Done}\ a \mid \mathit{Pause}\ (\mathit{St}\ (R\ a))\\ \textbf{instance}\ \mathit{Monad}\ R\ \textbf{where}\\ \quad\begin{array}{lcll}\textbf{return} & = & \mathit{Done} & (*)\\ (\mathit{Done}\ v)\ \texttt{>>=}\ f & = & f\ v & \\ (\mathit{Pause}\ r)\ \texttt{>>=}\ f & = & \mathit{Pause}\,(r\ \texttt{>>=}_{St}\ \lambda k.\,\textbf{return}_{St}\,(k\ \texttt{>>=}\ f)) & \end{array}\end{array}$$

Here, the bind operator for R is defined recursively in terms of the bind and unit for the state monad (written above as $\texttt{>>=}_{St}$ and $\textbf{return}_{St}$, respectively). The difference with the previous monad definition is that some stateful computation—within "$r\ \texttt{>>=}_{St}\ldots$" in the last line of the instance declaration—takes place.

A useful non-proper morphism, *step*, may be defined to recast a stateful computation as a resumption computation:

$$\begin{array}{l}\mathit{step}\ \ ::\ \mathit{St}\ a \to R\ a\\ \mathit{step}\ x = \mathit{Pause}\ (x\ \texttt{>>=}_{St}\ (\textbf{return}_{St} \circ \mathit{Done}))\end{array}$$

The *step* morphism is used later in Section 5 to define the atomic actions of the thread model.

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 $St\ ()$, then the computations of type $R\ ()$ are the set of possible interleavings:

$$\begin{array}{l} Pause\ (a_0 \texttt{>>} \mathbf{return}\ (Pause\ (a_1 \texttt{>>} \mathbf{return}\ (Pause\ (b_0 \texttt{>>} \mathbf{return}\ (Done\ ()))))))) \\ Pause\ (a_0 \texttt{>>} \mathbf{return}\ (Pause\ (b_0 \texttt{>>} \mathbf{return}\ (Pause\ (a_1 \texttt{>>} \mathbf{return}\ (Done\ ()))))))) \\ Pause\ (b_0 \texttt{>>} \mathbf{return}\ (Pause\ (a_0 \texttt{>>} \mathbf{return}\ (Pause\ (a_1 \texttt{>>} \mathbf{return}\ (Done\ ())))))))) \end{array}$$

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: $Pause\ (h \texttt{>>} \mathbf{return}\ t)$. 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.

With this discussion in mind, we may generalize the previous construction as a monad transformer:

$$\begin{array}{l} \mathbf{data}\ (Monad\ m) \Rightarrow ResT\ m\ a = Done\ a \mid Pause\ (m\ (ResT\ m\ a)) \\ \mathbf{instance}\ (Monad\ m) \Rightarrow Monad\ (ResT\ m)\ \mathbf{where} \\ \quad return \qquad\quad = Done \\ \quad (Done\ v) \texttt{>>=}\ f = f\ v \\ \quad (Pause\ r) \texttt{>>=}\ f = Pause\ (r \texttt{>>=}_m \lambda\ k\ .\ \mathbf{return}_m\ (k \texttt{>>=}\ f)) \\ step :: (Monad\ m) \Rightarrow m\ a \rightarrow ResT\ m\ a \\ step\ x = Pause\ (x \texttt{>>=}_m (\mathbf{return}_m \circ Done)) \end{array}$$

The particular formulation of the basic resumption monad and monad transformer we use are due to Papaspyrou (Papaspyrou, 1998; Papaspyrou & Maćoš, 2000; Papaspyrou, 2001), although others exist as well (Moggi, 1990; Espinosa, 1995; Filinski, 1999).

4.1.1 Approximation Lemma for Basic Resumptions

There are well-known techniques for proving equalities of infinite lists; these are the *take lemma* (Bird & Wadler, 1988) and the more general *approximation lemma* (Gibbons & Hutton, 2004). Both are based on the fact that, if all initial prefixes of two lists are equal, then the lists themselves are equal; it does not matter whether the lists are finite, infinite, or partial. The approximation lemma may be stated as: for any two lists xs and ys,

$$xs = ys \Leftrightarrow \text{for all } n \in \omega,\ approx\ n\ xs = approx\ n\ ys$$

where $approx :: Int \rightarrow [a] \rightarrow [a]$ is defined as:

$$\begin{array}{l} approx\ (n{+}1)\ [] \qquad = [] \\ approx\ (n{+}1)\ (x : xs) = x : (approx\ n\ xs) \end{array}$$

A similar result holds for basic resumption computations as well. The Haskell function *approx* approximates R computations:

$$\begin{aligned}
&\mathit{approx} :: \mathit{Int} \to R\,a \to R\,a\\
&\mathit{approx}\ (n{+}1)\ (\mathit{Done}\ v) = \mathit{Done}\ v\\
&\mathit{approx}\ (n{+}1)\ (\mathit{Pause}\ \varphi) = \mathit{Pause}\ (\varphi \texttt{ >>= } (\mathbf{return} \circ \mathit{approx}\ n))
\end{aligned}$$

where `>>=` and **return** are the bind and unit operations of the monad M. Note that, for any finite resumption-computation φ, $\mathit{approx}\ n\ \varphi = \varphi$ for any sufficiently large n—that is, $(\mathit{approx}\ n)$ approximates the identity function on resumption computations. We may now state the approximation lemma for R:

Theorem 2 (*Approximation Lemma for R*)
For any $\varphi, \gamma :: R\ a$, $\varphi = \gamma \Leftrightarrow$ *for all* $n \in \omega$, $\mathit{approx}\ n\ \varphi = \mathit{approx}\ n\ \gamma$.

Theorem 2 is proved in Appendix C.

4.2 Reactive Concurrency

We now consider a refinement to the concurrency model presented in the Section 4.1 which allows computations to signal requests and receive responses in a manner something like software interrupts; we coin the term *reactive* resumption[8] to distinguish this structure from the previous one. The concurrency associated with the reactive resumption monad resembles nothing so much as the interaction between an operating system and processes making system calls. Before presenting reactive concurrency in monadic form, we take a short detour to motivate this intuition.

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 r will be generated to indicate the status of the system call (e.g., its success or failure).
3. Using the information contained in the response code r, the process continues execution.

[8] A *reactive* program (Manna & Pnueli, 1991) is one which interacts continually with its environment and may be designed to not terminate (e.g., an operating system). Reactive programs may be modeled with reactive resumptions, hence the choice of name.

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.

We may add the computational raw material for interactivity to the "state + resumptions" monad (i.e., the monad defined at $(*)$ in Section 4.1) as follows:

$$\textbf{data}\ \mathit{Re}\ a = D\ a \mid P\ (\mathit{Req},\ \mathit{Rsp} \rightarrow (\mathit{St}(\mathit{Re}\ a)))$$

To distinguish the reactive variety of resumption monad from the basic, we use D and P instead of "*Done*" and "*Pause*", respectively. The notion of concurrency provided by this monad 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 from *Rsp*, is the rest of the computation. The instance declaration for this monad is:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ \mathit{Re}\ \textbf{where}\\
&\quad \mathit{return}\ v \qquad = D\ v\\
&\quad D\ v \mathbin{>\!\!>\!\!=} f \qquad = f\ v\\
&\quad P\ (q, r) \mathbin{>\!\!>\!\!=} f = P\ (q,\ \lambda\ \mathit{rsp}\ .\ (r\ \mathit{rsp}) \mathbin{>\!\!>\!\!=_{St}} \lambda\ k\ .\ \textbf{return}_{St}\ (k \mathbin{>\!\!>\!\!=} f))
\end{aligned}$$

The reactive variety of resumption monad has been known for some years now, having its origin in early work of Moggi (Moggi, 1990), where reactive monads are written in categorical style as functors (Barr & Wells, 1990). At this point, it may be helpful for some readers to compare such notation to the Haskell monad declarations:

$$\begin{aligned}
T\ A\ &=\ \mu X.\ (A\ +\ (\mathit{Rsp} \rightarrow X)) && \text{— interactive input}\\
T\ A\ &=\ \mu X.\ (A\ +\ (\mathit{Req} \times X)) && \text{— interactive output}\\
T\ A\ &=\ \mu X.\ (A\ +\ (\mathit{Req} \times (\mathit{Rsp} \rightarrow X))) && \text{— interactive input/output}
\end{aligned}$$

Here, the μ notation is used to make the recursion within a data type declaration explicit; $\mu X.\tau$ refers to the least solution of the recursive domain equation $X = \tau$. The third monad (a.k.a., interactive input/output) implements what we have called reactive concurrency.

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 6):

$$\begin{aligned}
&\textbf{type}\ \mathit{Message} = \mathit{Int}\\
&\textbf{type}\ \mathit{PID} \quad = \mathit{Int}\\
&\textbf{data}\ \mathit{Req} \quad = \mathit{Cont} \mid \mathit{SleepReq} \mid \mathit{ForkReq}\ \mathit{Comm} \mid \mathit{BcastReq}\ \mathit{Message}\\
&\qquad\qquad \mid \mathit{RecvReq} \mid \mathit{ReleaseSemReq} \mid \mathit{GetSemReq} \mid \mathit{PrintReq}\ \mathit{String}\\
&\qquad\qquad \mid \mathit{GetPIDReq} \mid \mathit{KillReq}\ \mathit{PID}\\
&\textbf{data}\ \mathit{Rsp} \quad = \mathit{Ack} \mid \mathit{ForkRsp} \mid \mathit{RecvRsp}\ \mathit{Message} \mid \mathit{GetPIDRsp}\ \mathit{PID}
\end{aligned}$$

As with basic resumptions, reactive resumption monads may also be generalized as a monad transformer. The following monad transformer abstracts over the request and response data types as well as over the input monad:

$$\begin{array}{l}\textbf{data}\ (\mathit{Monad}\ m)\ \Rightarrow\ \mathit{ReactT}\ \mathit{req}\ \mathit{rsp}\ m\ a\ =\ D\ a \mid P\ (\mathit{req},\ \mathit{rsp}\rightarrow(m\ (\mathit{ReactT}\ \mathit{req}\ \mathit{rsp}\ m\ a)))\\ \textbf{instance}\ (\mathit{Monad}\ m)\ \Rightarrow\ \mathit{Monad}\ (\mathit{ReactT}\ \mathit{req}\ \mathit{rsp}\ m)\ \textbf{where}\\ \quad \textbf{return}\ v\ \ \ \ \ \ = D\ v\\ \quad (D\ v)\ \texttt{>>=}\ f\ \ \ = f\ v\\ \quad P\ (q, r)\ \texttt{>>=}\ f = P\ (q,\ \lambda\ c\ .\ (r\ c)\ \texttt{>>=}_m\ \lambda\ k\ .\ \textbf{return}_m\ (k\ \texttt{>>=}\ f))\end{array}$$

Reactive resumption monads have two non-proper morphisms. The first of these, *step*, is defined along the same lines as with *ResT*:

$$\begin{array}{l}\mathit{step}\ \ \ :: \mathit{St}\ a\ \rightarrow\ \mathit{Re}\ a\\ \mathit{step}\ x = P\ (\mathit{Cont},\ \lambda\ \mathit{Ack}.\ x\ \texttt{>>=}_{St}\ (\textbf{return}_{St}\ \circ\ D))\end{array}$$

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 provided by *ReactT* allows a computation to raise a signal; its definition is given as:

$$\begin{array}{ll}\mathit{signal}\ \ \ :: \mathit{Req}\ \rightarrow\ \mathit{Re}\ \mathit{Rsp} & \mathit{signalI}\ \ \ :: \mathit{Req}\ \rightarrow\ \mathit{Re}\ a\\ \mathit{signal}\ q = P(q,\ \textbf{return}_{St}\circ\textbf{return}) & \mathit{signalI}\ q = P\ (q,\ \lambda\ \_\ .\ \textbf{return}_{St}\ (\textbf{return}_{Re}\ \mathit{nil}))\end{array}$$

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

5 What is a "thread" anyway?

Operating systems texts (for example (Deitel, 1982)) define threads as lightweight processes[9] executed in the same address space. Some concurrent programming languages (notably CSP (Hoare, 1978a) and SR(Andrews & Olsson, 1993)) also contain a notion of threads. But what is an ideal thread? The fundamental building block of a thread is the *atomic* action. An action is atomic when it is both non-interruptable and terminating—the archetypal example of such an action is a single machine instruction. A *thread* is, then, a (potentially infinite) sequence of such atomic actions: $(a_0; a_1; \ldots)$.

To precisely define "atom" and "thread" in the monadic setting, we begin by first summarizing the underlying monadic signatures created thus far with monad transformers; eliding the `>>=` and **return** operations, these are:

St=StateT Sto Id	*R=ResT St*	*Re=ReactT Req Rsp St*
$g\ ::\ \mathit{St}\ \mathit{Sto}$	$g\ ::\ \mathit{St}\ \mathit{Sto}$	$g\ ::\ \mathit{St}\ \mathit{Sto}$
$u\ ::\ (\mathit{Sto}\rightarrow\mathit{Sto})\rightarrow\mathit{St}\ a$	$u\ ::\ (\mathit{Sto}\rightarrow\mathit{Sto})\rightarrow\mathit{St}\ a$	$u\ ::\ (\mathit{Sto}\rightarrow\mathit{Sto})\rightarrow\mathit{St}\ a$
	$\mathit{step}\ ::\ \mathit{St}\ a\rightarrow\mathit{R}\ a$	$\mathit{step}\ ::\ \mathit{St}\ a\rightarrow\mathit{Re}\ a$
		$\mathit{signal}\ ::\ \mathit{Req}\rightarrow\mathit{Re}\ \mathit{Rsp}$

[9] Throughout, we do not distinguish threads from processes as operating systems texts typically do.

Here, *Req* and *Rsp* are as defined in Section 4.2. Each of the above monadic signatures allows useful non-proper morphisms to be defined; within each of the above monadic signatures, the morphism *getloc* :: *Name*→*St Integer* may be defined and *signalI* :: *Req*→*Re a* may be defined in the rightmost.

An atomic action will have type *St a*. However, not all Haskell terms of type *St a* may be considered atomic, as they may fail to terminate. Consider the term of type *St a* defined as $bomb = u\,(\lambda\,\sigma.\,\sigma)\;{>>}_{St}\;bomb$. Because *bomb* does not terminate, it can not be considered atomic. We restrict the notion of atomic action to Haskell terms of type *St a* constructed from *getloc*, $u[x{\mapsto}v]$, $>>=_{St}$, and $\mathbf{return}_{St}$ without the use of recursion or error-producing Haskell operations such as *undefined* and *error*. Threads are Haskell terms of type *Re a* defined with similar restrictions; namely, threads are formed from *P*, *D*, *step x* (for atomic *x* :: *St a*), $>>=_{Re}$, and $\mathbf{return}_{Re}$ without *undefined* or *error*. The general form of a thread is:

$$signal\ RecvReq\ >>=\ \lambda\,(RecvRsp\ m).\ step\ (u[x \mapsto\ m])\ >>\ step\ (getloc\ x)\ >>=\ \cdots$$

Note that the restricted use of the store (i.e., using *getloc* and $u[x{\mapsto}v]$ rather than *g* and $u\,\delta$ for arbitrary δ) guarantees the single-threadedness (in the sense of (Flanagan & Qadeer, 2003)) of such threads.

5.1 Language of Threads

We now consider a language of threads; its monadic semantics is convenient for thread creation. An abstract syntax for this language is shown in Figure 1. The language is much like that of (Papaspyrou, 2001) extended with signals; to accommodate signaling here, we must use reactive resumptions.

```
type Name    = String
data Prog    = PL [Comm]
data Exp     = Plus Exp Exp | Var Name | Lit Integer | GetPID
data BoolExp = Equal Exp Exp | Leq Exp Exp | TrueExp | FalseExp
data Comm    = Skip
             | Assign Name Exp
             | Seq Comm Comm
             | If BoolExp Comm Comm
             | While BoolExp Comm
             | Print String Exp        — prints string with expression value
             | Psem                    — acquire semaphore
             | Vsem                    — release semaphore
             | Sleep                   — suspend execution
             | Fork Comm               — creates thread for argument
             | Broadcast Name          — broadcasts value of variable
             | Receive Name            — rec'vs avail. msg; suspends otherwise
             | Kill Exp                — preempts its arg., causing termination
```

Fig. 1. *The Language of Threads.* This concurrent, imperative language allows threads to signal the operating system. Within the expression language, there is a special signal for returning a threads process identifier.

```
mexp :: Exp → Re Integer
mexp (Lit i) = return i
mexp (Plus e1 e2) =
  do v1 ← mexp e1
     v2 ← mexp e2
     return (v1+v2)
mexp (Var x) = step (getloc x)
mexp GetPID =
  signal GetPIDReq >>=
    λ(GetPIDRsp pid). return pid

mbexp :: BoolExp → Re Bool
mbexp (Equal e1 e2) =
  do v1 ← mexp e1
     v2 ← mexp e2
     return (v1==v2)
mbexp (Leq e1 e2) =
  do v1 ← mexp e1
     v2 ← mexp e2
     return (v1≤v2)
mbexp TrueExp = return True
mbexp FalseExp = return False
```

Fig. 2. Semantics for Language of Threads: Expressions

The programming language for threads is similar to familiar examples from the literature of concurrency such CSP (Hoare, 1978a), Dijkstra's guarded command language (Dijkstra, 1975), and the language of temporal logic (Manna & Pnueli, 1991). Programs in *Prog* are finite lists of commands: $c_1 \parallel \ldots \parallel c_n$, where $c_i \in Comm$, and collectively, these commands c_i are executed concurrently over a shared state. *Comm* includes the simple imperative language with loops, and the semantics of this language fragment (i.e., while programs with "$\parallel$") is the same as one would find elsewhere (Papaspyrou, 2001; Papaspyrou, 1998). Where the language and its semantics differ from previous work is in the inclusion of signals; commands may request some intervention on the part of the kernel. The abstract syntax for signal commands are in the last seven lines of the *Comm* grammar from Figure 1 and allow signals for output, synchronization, suspension and forking, and message passing.

The semantics of expressions, *Exp* and *BoolExp*, are shown in Figure 2. With the exception of *GetPID* expression, they are standard monadic definitions for arithmetic and boolean expressions. The *GetPID* expression requests the process identifier of the thread. First, the request is made using *signal GetPIDReq* and the response to this request will be of the form (*GetPIDRsp pid*); when such a response comes, the identifier *pid* is returned.

Figures 3 and 4 show the semantics of *Comm* and *Prog*. Just as the case for expressions, the *While* fragment of *Comm* has a typical definition for an imperative language, or rather, the *text* of its definition is typical. This semantics has the effect of "unrolling" the loop to create a (potentially) infinite computation in the *Re* monad. For example, the meaning of the program (`while (0==0) skip`)—written in concrete syntax—is equal to the following "unrolling":

$$(cmd\ \texttt{skip}) \gg\!=_{Re} (mexp\ \texttt{0==0}) \gg\!=_{Re} (cmd\ \texttt{skip}) \gg\!=_{Re} (mexp\ \texttt{0==0}) \gg\!=_{Re} \ldots$$

Note that this denotes an infinite computation in *Re* and not $\perp$ (as it would if the iteration occurred in the state monad *St*).

The interesting part is that of the signaling commands shown in Figure 4. Each of these commands is defined using a call to the *signal* morphism of the *Re* monad

(or the defined morphism *signalI* from Section 4.2); for example, the commands to receive a message and acquire the semaphore are defined as:

$$\begin{array}{lll} \mathit{cmd} & :: \mathit{Comm} \rightarrow \mathit{Re}\ a \\ \mathit{cmd}\ (\mathit{Receive}\ x) & = & \mathit{signal}\ \mathit{RecvReq} \gg\!= \lambda\,(\mathit{RecvRsp}\ m).\ (\mathit{store}\ x\ m) \\ \mathit{cmd}\ \mathit{Psem} & = & \mathit{signalI}\ \mathit{GetSemReq} \end{array}$$

To *Receive* a message into the program variable x, first the command *signal*s a receive request, expecting a response of the form $(\mathit{RecvRsp}\ m)$ where m is the message contents. Having received the message, it is then *store*d in x. To acquire the semaphore, simply send the signal *GetSemReq*. The response code to such a request, if it comes, is always *Ack* and, for this reason, *signalI* is used to ignore the response code.

6 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 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

$$\begin{array}{lll}
\mathit{tweek} & :: \mathit{Eq}\ a \Rightarrow a \rightarrow b \rightarrow (a \rightarrow b) \rightarrow a \rightarrow b \\
\mathit{tweek}\ i\ v\ \sigma & = & \lambda\, n.\ \mathbf{if}\ i == n\ \mathbf{then}\ v\ \mathbf{else}\ (\sigma\ n) \\
\mathit{store} & :: \mathit{Name} \rightarrow \mathit{Integer} \rightarrow \mathit{Re}\ a \\
\mathit{store}\ \mathit{loc}\ v & = & (\mathit{step} \circ u)\ (\mathit{tweek}\ \mathit{loc}\ v) \\
 & & \\
\mathit{prog} & :: \mathit{Prog} \rightarrow [\mathit{Re}\ a] \\
\mathit{prog}\ (\mathit{PL}\ cs) & = & \mathit{map}\ \mathit{cmd}\ cs \\
 & & \\
\mathit{cmd} & :: \mathit{Comm} \rightarrow \mathit{Re}\ a \\
\mathit{cmd}\ \mathit{Skip} & = & \mathbf{return}\ \mathit{nil} \\
\mathit{cmd}\ (\mathit{Assign}\ x\ e) & = & (\mathit{mexp}\ e) \gg\!= \mathit{store}\ x \\
\mathit{cmd}\ (\mathit{Seq}\ c_1\ c_2) & = & \mathit{cmd}\ c_1 \gg \mathit{cmd}\ c_2 \\
\mathit{cmd}\ (\mathit{If}\ b\ c_1\ c_2) & = & \mathit{mbexp}\ b \gg\!= \lambda\, v.\ \mathbf{if}\ v\ \mathbf{then}\ \mathit{cmd}\ c_1\ \mathbf{else}\ \mathit{cmd}\ c_2 \\
\mathit{cmd}\ (\mathit{While}\ b\ c) & = & \mathit{mwhile}\ (\mathit{mbexp}\ b)\ (\mathit{cmd}\ c) \\
\quad \mathbf{where} & & \\
\quad\quad \mathit{mwhile}\ \beta\ \varphi & = & \mathbf{do} \quad v \leftarrow \beta \\
 & & \quad\quad \mathbf{if}\ v\ \mathbf{then}\ \varphi \gg (\mathit{mwhile}\ \beta\ \varphi)\ \mathbf{else}\ \mathbf{return}\ \mathit{nil}
\end{array}$$

Fig. 3. Semantics for Language of Threads: Programs and Commands (Part 1)

```
cmd (Print m e)    =  (mexp e) >>= λ v. signalI (PrintReq (output m v))
  where
    output m v  =  m ++ ": " ++ show v
cmd Sleep          =  signalI SleepReq
cmd (Fork c)       =  signalI (ForkReq c)
cmd (Broadcast x)  =  mexp (Var x) >>= (signalI ∘ BcastReq)
cmd (Receive x)    =  signal RecvReq >>= λ (RecvRsp m). (store x m)
cmd Psem           =  signalI GetSemReq
cmd Vsem           =  signalI ReleaseSemReq
cmd (Kill e)       =  (mexp e) >>= λ pid. signalI (KillReq pid)
```

Fig. 4. Semantics for Language of Threads: Commands (Part 2)

channel, and a counter for generating new process identifiers. The system configuration is captured as the following Haskell type declaration:

```
type System a = ([(PID, Re a)],   — waiting list
                 [Message],       — message buffer
                 Semaphore,       — 1 initially
                 String,          — output channel
                 PID)             — identifier counter
```

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. We say that a system configuration is *acceptable* when the process identifiers in the waiting list are mutually distinct and smaller than the identifier counter.

The types of a scheduler and service handler are:

```
sched   :: System a → R a
handler :: System a → (PID, Re a) → R a
```

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 $R\ a$ precisely because the requested thread interactions have been mediated by *handler* along the lines illustrated in Figure 5.

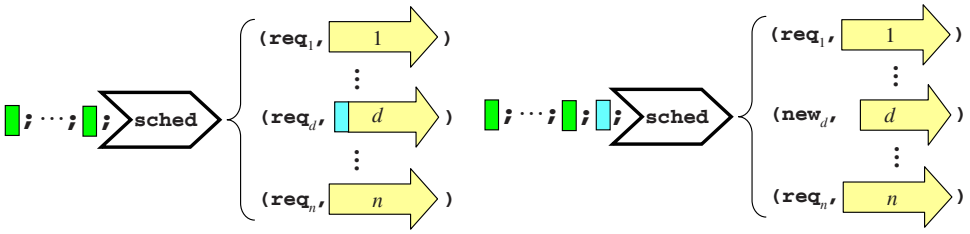

Fig. 5. *Structure and Operation of the Kernel.* (Left) The scheduler `sched` has a collection of threads 1 to n, each waiting on a service request $\texttt{req}_i$. To the left of `sched` are the previously dispatched slices of the threads, interleaved in order according to the scheduling policy. Thread d will be dispatched next, and the slice to be executed is highlighted. (Right) Thread d is passed to the service handler `handler` (not shown) which responds to $\texttt{req}_d$ and executes the d-slice. The handler then returns control to the scheduler.

Figure 5 illustrates the operation of the kernel and, in particular, the interaction between a scheduler, `sched`, and the service handler routines (not pictured). 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. This is represented pictorially in Figure 5: to the right of `sched` are the reactive thread computations that, when serviced, are woven into a single basic schedule computation.

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

$$
\begin{aligned}
&\mathit{rrobin} :: \mathit{System}\ a \rightarrow R\ a \\
&\mathit{rrobin}\ ([], \_, \_, \_, \_) &&= \mathit{Done}\ \mathit{nil} \quad \text{— stop when no threads} \\
&\mathit{rrobin}\ ((t : ts), mQ, \mathit{mutex}, \mathit{out}, \mathit{pgen}) &&= \mathit{handler}\ (ts, mQ, \mathit{mutex}, \mathit{out}, \mathit{pgen})\ t
\end{aligned}
$$

A typical call to *handler* has the form $(\mathit{handler}\ \mathit{sys}\ (\mathit{pid}, P(\mathit{req}, r)))$ and *handler* will choose the appropriate response for request *req*. Sections 6.1-6.6 describe the action of *handler* in detail.

6.1 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:

$$
\begin{aligned}
&\mathit{handler} :: \mathit{System}\ a \rightarrow (\mathit{PID}, \mathit{Re}\ a) \rightarrow R\ a \\
&\mathit{handler}\ (ts, mQ, \mathit{mutex}, \mathit{out}, \mathit{pgen})\ (\mathit{pid}, D\ \_) = \mathit{rrobin}\ (ts, mQ, \mathit{mutex}, \mathit{out}, \mathit{pgen})
\end{aligned}
$$

If the thread wishes to continue (i.e., it is of the form $P(Cont, r)$), then *handler* acknowledges the request by passing Ack to r:

$$\begin{aligned}
&handler\ (ts, mQ, mutex, out, pgen)\ (pid, P(Cont, r)) \\
&\quad = Pause\ ((r\ Ack)\ \texttt{>>=}_{St}\ \lambda\, k.\ \mathbf{return}_{St}\ (next\ (pid, k))) \\
&\quad \mathbf{where} \\
&\qquad next\ t\ =\ rrobin\ (ts{+\!\!+}[t], mQ, mutex, out, pgen)
\end{aligned}$$

As a result, the first atom in r is scheduled, and the rest the thread (written $next\ (pid, k)$ above) is passed to the scheduler.

6.2 Printing

When a print request ($PrintReq\ msg$) is signaled, 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.

$$\begin{aligned}
&handler\ (ts, mQ, mutex, out, pgen)\ (pid, P(PrintReq\ msg,\ r)) \\
&\quad = Pause\ (\mathbf{return}_{St}\ next) \\
&\quad \mathbf{where} \\
&\qquad next\ =\ rrobin\ (ts{+\!\!+}[(pid, P(Cont, r))], mQ, mutex, out{+\!\!+}msg, pgen)
\end{aligned}$$

An alternative implementation of output could use the "interactive output" monad formulation of Section 4.2 instead of encoding the output channel as the string out. An example of this formulation occurs in Section 7.

6.3 Dynamic Scheduling

A thread may request suspension with the *SleepReq* signal; the handler changes the *SleepReq* request to a *Cont* and reschedules the thread. The effect of this is to delay the thread by one scheduling cycle.

$$\begin{aligned}
&handler\ (ts, mQ, mutex, out, pgen)\ (pid, P(SleepReq, r)) = Pause\ (\mathbf{return}_{St}\ next) \\
&\quad \mathbf{where} \\
&\qquad next\ = rrobin\ (ts{+\!\!+}[(pid, P(Cont, r))], mQ, mutex, out, pgen)
\end{aligned}$$

An obvious refinement of this service would include a counter field within the *SleepReq* request and use this field to delay the thread through multiple cycles.

A request to spawn a new thread executing command c is handled by:

$$\begin{aligned}
&handler\ (ts, mQ, mutex, out, pgen)\ (pid, P(ForkReq\ c, r)) =\ Pause\ (\mathbf{return}_{St}\ next) \\
&\quad \mathbf{where} \\
&\qquad parent = (pid, cont\ r\ ForkRsp) \\
&\qquad child\ \ = (pgen, cmd\ c) \\
&\qquad next\ \ = rrobin\ (ts{+\!\!+}[parent,\ child], mQ, mutex, out, pgen{+}1)
\end{aligned}$$

A thread for c is created by applying cmd and the result is given a new identifier $pgen$. 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:

$$\begin{aligned}
&cont\ ::\ (Rsp\ \rightarrow\ St\ (Re\ a))\ \rightarrow\ (Rsp\ \rightarrow\ Re\ a) \\
&cont\ r\ rsp\ =\ P\ (Cont,\ \lambda\, Ack.\ r\ rsp)
\end{aligned}$$

6.4 Asynchronous Message Passing

For a thread to broadcast a message m, it is simply appended to the message queue:

$$\begin{array}{l} \mathit{handler}\ (\mathit{ts}, \mathit{mQ}, \mathit{mutex}, \mathit{out}, \mathit{pgen})\ (\mathit{pid}, P(\mathit{BcastReq}\ m, r)) = \mathit{Pause}\ (\mathbf{return}_{St}\ \mathit{next}) \\ \quad \mathbf{where} \\ \quad\quad \mathit{mQ}' = \mathit{mQ} +\!\!+ [m] \\ \quad\quad \mathit{next} = \mathit{rrobin}\ (\mathit{ts} +\!\!+ [(\mathit{pid}, \mathit{cont}\ r\ \mathit{Ack})], \mathit{mQ}', \mathit{mutex}, \mathit{out}, \mathit{pgen}) \end{array}$$

When a *RecvReq* signal occurs and the message queue is empty, then the thread must wait:

$$\begin{array}{l} \mathit{handler}\ (\mathit{ts}, [], \mathit{mutex}, \mathit{out}, \mathit{pgen})\ (\mathit{pid}, P(\mathit{RecvReq}, r)) = \mathit{Pause}\ (\mathbf{return}_{St}\ \mathit{next}) \\ \quad \mathbf{where} \\ \quad\quad \mathit{next} = \mathit{rrobin}\ (\mathit{ts} +\!\!+ [(\mathit{pid}, P(\mathit{RecvReq}, r))], [], \mathit{mutex}, \mathit{out}, \mathit{pgen}) \end{array}$$

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:

$$\begin{array}{l} \mathit{handler}\ (\mathit{ts}, (m : \mathit{ms}), \mathit{mutex}, \mathit{out}, \mathit{pgen})\ (\mathit{pid}, P(\mathit{RecvReq}, r)) = \mathit{Pause}\ (\mathbf{return}_{St}\ \mathit{next}) \\ \quad \mathbf{where} \\ \quad\quad \mathit{next} = \mathit{rrobin}\ (\mathit{ts} +\!\!+ [(\mathit{pid}, \mathit{cont}\ r\ (\mathit{RecvRsp}\ m))], \mathit{ms}, \mathit{mutex}, \mathit{out}, \mathit{pgen}) \end{array}$$

6.5 Preemption

One thread may preempt another by sending it a kill signal reminiscent of the Unix `(kill -9)` command; this is implemented by the following *handler* declaration that, upon receiving the signal *KillReq* i, removes the thread with process identifier i from the waiting list:

$$\begin{array}{l} \mathit{handler}\ (\mathit{ts}, \mathit{mQ}, \mathit{mutex}, \mathit{out}, \mathit{pgen})\ (\mathit{pid}, P(\mathit{KillReq}\ i,\ r))\ =\ \mathit{Pause}\ (\mathbf{return}_{St}\ \mathit{next}) \\ \quad \mathbf{where} \\ \quad\quad \mathit{next}\ = \mathit{rrobin}\ (\mathit{wl}', \mathit{mQ}, \mathit{mutex}, \mathit{out}, \mathit{pgen}) \\ \quad\quad \mathit{wl}'\ = \mathit{filter}\ (\mathit{exit}\ i)\ (\mathit{ts} +\!\!+ [(\mathit{pid}, \mathit{cont}\ r\ \mathit{Ack})]) \\ \quad\quad \mathit{exit}\ i = \lambda\,(\mathit{pid}, t).\ i\ /=\ \mathit{pid} \end{array}$$

Thread i is removed from the wait list using the Haskell built-in function:

$$\mathit{filter}\ ::\ (a\ \rightarrow\ \mathit{Bool})\ \rightarrow\ [a]\ \rightarrow\ [a]$$

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).

The *KillReq* signal is a primitive example of what is known in Concurrent Haskell as an asynchronous exception (Marlow *et al.*, 2001). In Concurrent Haskell, threads may interrupt one another using the asynchronous exception operator *throwTo*:

$$\mathit{throwTo} :: \mathit{ThreadID} \rightarrow \mathit{Exception} \rightarrow \mathit{IO}()$$

A call (*throwTo* p e) sends the exception e to the thread p, not returning until the exception has been raised. If the interrupted thread has no exception handler, it enters an error state; otherwise it processes the exception according to its handler. In one sense, this behaves much like sending a message asynchronously, where the

exception is the message itself. While this description is admittedly high-level and imprecise, it does suggest a likely approach to a resumption-monadic model for asynchronous exceptions.

6.6 Synchronization Primitives

Requesting the system semaphore *mutex* will succeed if $mutex > 0$, in which case the requesting thread will continue with the semaphore decremented; if $mutex \not> 0$, 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 *mutex*:

$$\begin{array}{l} handler\ (ts, mQ, mutex, out, pgen)\ (pid, P(GetSemReq,\ r)) \ =\ Pause\ (\mathbf{return}_{St}\ next) \\ \quad \mathbf{where} \\ \quad\quad next \quad\ = \mathbf{if}\ mutex > 0\ \mathbf{then}\ goahead\ \mathbf{else}\ tryagain \\ \quad\quad goahead\ = rrobin\ (ts{+\!\!+}[(pid, P(Cont, r))], mQ, mutex{-}1, out, pgen) \\ \quad\quad tryagain = rrobin\ (ts{+\!\!+}[(pid, P(GetSemReq, r))], mQ, mutex, out, pgen) \end{array}$$

Note that this implementation uses busy waiting, the reason being merely its simplicity. One could easily implement more efficient strategies by including a queue of waiting threads with the semaphore.

A thread may release the semaphore without blocking and this is encoded by the following *handler* routine:

$$\begin{array}{l} handler\ (ts, mQ, mutex, out, pgen)\ (pid, P(ReleaseSemReq, r)) = Pause\ (\mathbf{return}_{St}\ next) \\ \quad \mathbf{where} \\ \quad\quad next = rrobin\ (t{+\!\!+}[cont\ r\ Ack], mQ, mutex{+}1, out, pgen) \end{array}$$

Note that this semaphore is *general* rather than *binary* (Andrews & Olsson, 1993), meaning that the semaphore counter *mutex* may have as its value any non-negative integer rather than just 0 or 1.

7 A Stream-based Proof Technique

This section considers proving the fairness of the scheduler *rrobin* from Section 6 and, to accomplish this task, introduces a proof technique for the resumption-monadic framework developed in previous sections. That round robin scheduling is fair is certainly not earth-shaking news, but what makes this verification interesting is that it boils down to straightforward equational reasoning in a style reminiscent of Bird and Wadler's classic text on functional programming (Bird & Wadler, 1988) and makes use of techniques for reasoning about streams (Gibbons & Hutton, 2004). The verification is performed directly on the kernel code itself rather than on system abstractions as do some well-known techniques for proving liveness properties (Pnueli, 1977; Owicki & Lamport, 1982; Misra *et al.*, 1982).

The proof technique uses the stream-like nature of resumption computations as leverage and is akin to abstract interpretation (Abramsky & Hankin, 1987). With this method, a resumption computation (e.g., the scheduling of threads on the kernel) is projected into a stream of snapshots characterizing the aspects of the system state relevant to the system specification; this system specification may then be for-

mulated in terms of these streams. System verification then proceeds from algebraic properties of the monads and their non-proper morphisms, monad transformers, and streams. Monad transformers are well-known for providing software engineering benefits (e.g., modularity, etc.); one noteworthy aspect of the proof technique is how it relies on them for program verification. Specifically, this proof technique proceeds according to the following steps:

1. Defining the system snapshots. For the fairness of *rrobin*, the relevant snapshots are abstract "scheduling events" characterizing the contents of the wait list and the thread to be executed. The precise definition of the system snapshots appears below in Section 7.1, as does the definition of fair schedules in terms of these event streams.
2. Re-interpreting the kernel in a monad with interactive output. Recording system traces is supported by a refinement to the basic resumption monad transformer in Section 7.2. This refinement allows interactive output in the sense of Section 4.2. One advantage of the monad transformer-based approach is that it is clear the original kernel behaves identically when re-interpreted in the richer monad.
3. Instrumenting the kernel to observe snapshots. Making use of interactive output, the scheduler is now instrumented with "print snapshot" statements in Section 7.3. That this instrumented kernel behaves identically to the original with respect to thread execution is a consequence of the structuring by monad transformers (Theorem 3).
4. Generating the system traces. Section 7.4 describes the Haskell function *trace* which generates system traces for the instrumented kernel and specifies its interaction with *rrobin* and *handler*.
5. Proving the system traces meet the specification. Section 7.5 demonstrates in Theorem 4 the fairness of the instrumented kernel using straightforward equational reasoning.

7.1 Observations of the System State

A waiting process on an operating system is *live* if it will eventually run. A scheduler is *fair* if, in every possible system execution, every waiting process is live. Therefore, to specify the fairness of a scheduler, one needs a record of which processes are waiting and which process is running at any particular point in any system execution. So, the required system observations must describe each scheduling decision; that is, these snapshots describe which threads are waiting and which will be dispatched next. Events not corresponding to scheduler actions (e.g., the atomic actions within threads) are considered "null" events and are irrelevant to the fairness specification. With that in mind, such observations of system events may be encoded by the Haskell data type *Obs*:

Definition 1 (*Observations*)
The datatype of system observations is defined is Haskell as:

data *Obs* = *Sched* [*PID*] *PID* | *Null*

An *Obs* value, (*Sched waiting running*), is a snapshot of the scheduler where *waiting* is the thread waiting list (or, rather, the *PID*s of waiting threads) and *running* is the *PID* of the thread to be run next. A *Null* observation corresponds to any non-scheduler action. As a notational convention, if $s :: [Obs]$ is a stream of observations and $o :: Obs$ is a particular observation, then we use the set membership relation, $o \in s$, to signify that o occurs in the stream s.

Definition 1 specifies the events of interest in the specification and proof of fairness. However, arbitrary streams of *Obs* values may not correspond to any actual scheduler trace due to the possibility of undefined values (i.e., $\perp$) or other things inconsistent with operating system invariants (such as the finiteness of the waiting list, etc.). Definition 2 below presents necessary conditions for a stream of *Obs* values to correspond to an actual system trace and we refer to such traces as *schedulings*:

Definition 2 (*Scheduling*)
Let s :: $[Obs]$ be any stream of observations and (*Sched waiting running*) be any arbitrary observation in s, then s is a *scheduling*, if, and only if,

(i) $running \notin waiting$,
(ii) $Null \notin s$,
(iii) *waiting* is finite and contains no duplicates, and
(iv) $\perp$ "occurs nowhere" in s. I.e., $s \neq \perp$, $e \neq \perp$ for all $e \in s$, $(Sched\ wl\ r) \in s$ implies $wl \neq \perp$, $r \neq \perp$, and $\forall\, w \in wl.\ w \neq \perp$.

Condition (ii) is not strictly speaking necessary, but its inclusion removes irrelevant data from schedulings, thereby simplifying the system verification. Note that condition (iv) is merely a hygiene condition; because Haskell 98 is a lazy language, such seemingly unnatural side conditions must occasionally be given.

The following definition formulates liveness in terms of schedulings:

Definition 3 (*Liveness*)
Let s :: $[Obs]$ be a scheduling such that $o = (Sched\ wl\ r)$ and $s = o : os$. Then, p is *live in* s means precisely that either:

(i) $p = r$, or
(ii) $p \in wl$ and
  (a) there is an $o' = (Sched\ wl'\ r') \in os$ such that $p = r'$,
  (b) for all observations $(Sched\ wl^i\ r^i)$ between o and o' in s, $p \in wl^i$.

A scheduling is fair when any process is either running or, if in the waiting list, will eventually run. Definition 4 formalizes this notion:

Definition 4 (*Fair Scheduling*)
Let s :: $[Obs]$ be a scheduling. Then, s is *fair*, if, and only if, either:

(i) $s = []$, or
(ii) $s = (Sched\ wl\ r) : os$ where
  (a) w is live in s for all $w \in wl$, and
  (b) os is fair.

7.2 Reinterpreting the Kernel in a Richer Monad

Now that we know what we want to observe, how do we record these observations? The approach taken here is analogous to the familiar technique for debugging imperative programs where print statements are inserted into the source program to allow observation of the internal state of the executing program. How this is accomplished in the monadic setting may seem unclear at first, but it is another instance of where the use of monad transformers serves us well. Refining the resumption monad transformer used in defining *rrobin* (i.e., *ResT*) to include interactive output (in the sense of Section 4.2) allows each scheduler decision to be "printed out."

The steps to take are as follows. First, we refine the *ResT* transformer so that it allows computations to signal outputs. The monad transformer we define, which we call "*ObsT*", is:

```
data (Monad m) ⇒ ObsT obs m a = Dn a | Ps obs (m (ObsT obs m a))
instance Monad m ⇒ Monad (ObsT obs m) where
  return      = Dn
  (Dn v) >>= f = f v
  Ps o x >>= f = Ps o (x >>=_m λ r. return_m (r >>= f))
```

The *ObsT* transformer is quite similar to *ResT*; the difference being that the "pause" constructor of *ObsT*, *Ps*, has an additional field representing output, and such output is accomplished with a non-proper morphism called *observe*. The *observe* morphism, along with the resumption monad with observations, *Ro*, is defined as:

```
type Ro  = ObsT Obs St a
observe  :: Obs → Ro a
observe o = Ps o (return_St (return nil))
```

7.3 Instrumenting the Kernel

We now write the *rrobin* scheduler in terms of the *Ro* monad (and distinguish the instrumented scheduler and handler with a subscript "*i*"):

```
rrobin_i :: System a → Ro a
rrobin_i ([], _, _, _, _)                    = Dn nil
rrobin_i ((t : ts), mQ, mutex, out, pgen) =
  observe o >>_Ro handler_i (ts, mQ, mutex, out, pgen) t
    where
      o = Sched (map fst ts) (fst t)
```

Notice that the Haskell code for $rrobin_i$ is identical to the previous uninstrumented version, except that there is each scheduling choice is recorded by a call to *observe*. The instrumented $handler_i$ routines are identical to the original except that, instead of the *Pause* constructor of the *R* monad, the following morphism is used:

```
pause :: St (Ro a) → Ro a
pause = Ps Null
```

The complete text of the instrumented handler is included in Appendix A.

It is important that the behavior of the kernel does not change. We can formalize this idea by projecting the *Ro* computation to an R computation in a manner which "forgets" any output events of the form (*Sched* _ _), and to accomplish this, we define the following morphism:

$$\begin{aligned}
&\mathit{forget} :: \mathit{Ro}\ a \rightarrow R\ a \\
&\mathit{forget}\ (\mathit{Dn}\ v) &&= \mathit{Done}\ v \\
&\mathit{forget}\ (\mathit{Ps}\ \mathit{Null}\ \varphi) &&= \mathit{Pause}\ (\varphi \mathbin{>\!\!>\!\!=_{St}} (\mathbf{return}_{St} \circ \mathit{forget})) \\
&\mathit{forget}\ (\mathit{Ps}\ (\mathit{Sched}\ _\ _)\ \varphi) &&= \mathit{forget}\ \varphi' \\
&\quad \mathbf{where}\ (\varphi', _) = \mathit{run}\ \varphi\ \mathit{undefined}
\end{aligned}$$

Theorem 3 shows that the addition of interactive output has no effect on the operation of the system. In the theorem and for the rest of this article, system configurations are abbreviated by their waiting list components. Furthermore, we assume that each system configuration is acceptable in the sense of Section 6. It is clear by construction that, if the initial system configuration is acceptable, then each system configuration passed between rrobin_i and $\mathit{handler}_i$ is also acceptable.

Theorem 3
For threads $t_1, \ldots, t_n$:

$$\mathit{forget}\ (\mathit{rrobin}_i\ [(i_1, t_1), \ldots, (i_n, t_n)]) = \mathit{rrobin}\ [(i_1, t_1), \ldots, (i_n, t_n)]$$

Proof
Each call to the instrumented kernel, $\mathit{rrobin}_i\ \mathit{ts}$, "unwinds" into a sequence of *Ro*-atoms, $\hat{e}_i$, alternating with scheduling observations, o_i. This *Ro*-resumption computation[10] has the form: $o_1 \mathbin{>\!\!>} \hat{e}_1 \mathbin{>\!\!>} o_2 \mathbin{>\!\!>} \hat{e}_2 \mathbin{>\!\!>} \cdots$. Applying *forget* to $(\mathit{rrobin}_i\ \mathit{ts})$ "filters out" the scheduling observations, thereby producing the resumption computation in R, $e_1 \mathbin{>\!\!>} e_2 \mathbin{>\!\!>} \cdots$, which is simply the unwinding of the uninstrumented kernel $(\mathit{rrobin}\ \mathit{ts})$. The proof of Theorem 3 follows this pattern:

$$\begin{aligned}
(\mathit{forget} \circ \mathit{rrobin}_i)\ \mathit{ts} &=_{(1)} \mathit{forget}\ (o_1 \mathbin{>\!\!>} \hat{e}_1 \mathbin{>\!\!>} o_2 \mathbin{>\!\!>} \hat{e}_2 \mathbin{>\!\!>} \cdots) \\
&=_{(2)} e_1 \mathbin{>\!\!>} e_2 \mathbin{>\!\!>} \cdots \\
&=_{(3)} \mathit{rrobin}\ \ \mathit{ts}
\end{aligned}$$

This proof outline makes intuitive sense, although demonstrating step (3) requires more than induction. Both functions $\mathit{forget} \circ \mathit{rrobin}_i$ and *rrobin* (both of type $\mathit{System}\ a \rightarrow R\ a$) are *corecursive*, meaning that they are recursive in their codomain $R\ a$. Techniques abound for proving such programs (e.g., fixed point induction and coinduction among others) and they all formalize the above intuitive argument by showing that two programs are in some sense indistinguishable. An eminently readable overview of proof techniques for corecursion over lists is (Gibbons & Hutton, 2004). Our proof makes use of the approximation lemma for resumption computations (Theorem 2).

To show step (2), we demonstrate that *forget* "erases" the scheduling observations

[10] This is a slight simplification for the sake of the presentation; generally, the binding operation >>= would be used.

made by the instrumented kernel. Assume that o is of the form $(Sched\ \_\ \_)$ and that each >> below is in the Ro monad. Then, *forget* "erases" $(observe\ o)$:

$$
\begin{array}{ll}
& forget\ (observe\ o\ \texttt{>>}\ \varphi) \\
= & forget\ ((Ps\ o\ \mathbf{return}_{St}(\mathbf{return}\ nil))\ \texttt{>>}\ \varphi) \\
= & forget\ ((Ps\ o\ \mathbf{return}_{St}(\mathbf{return}\ nil\ \texttt{>>}\ \varphi)) \\
= & forget\ (Ps\ o\ \mathbf{return}_{St}(\varphi)) & \{\text{left unit for } Ro\} \\
= & forget\ (\pi_1\ (run\ (\mathbf{return}_{St}(\varphi))\ undefined)) & \{\text{defn. } forget,\ \pi_1(x,\_) = x\} \\
= & forget\ \varphi & \{\text{Theorem 1.(i)}\}
\end{array}
$$

This implies:

$$
\begin{array}{rl}
(forget\ \circ\ rrobin_i)\ (t : ts) = & forget\ (observe\ o\ \texttt{>>}\ handler_i\ ts\ t) \\
= & forget\ (handler_i\ ts\ t)
\end{array}
$$

It remains to show (3) that $(forget \circ rrobin_i)\ ws = rrobin\ ws$, for any system configuration ws. For this we use the approximation lemma for resumptions (Theorem 2) along with basic properties of monads. That is, for any natural number k, all we need show is:

$$approx\ k\ (forget \circ rrobin_i\ ws) = approx\ k\ (rrobin\ ws)$$

It is clear that the theorem holds both for any empty system configuration (i.e., one in which the thread wait is []) and if $k = 0$. Assume $(w : ws)$ is any system configuration and $k = n + 1$. We show the theorem for the case where $w = (pid, P(Cont, r))$; other cases are analogous.

Expanding the left-hand side, we obtain:

$$
\begin{array}{ll}
& approx\ (n{+}1)\ (forget \circ rrobin_i\ (w : ws)) \\
& \quad \{forget\ \text{"erases"}\ observe\} \\
= & approx\ (n{+}1)\ (forget\ (handler_i\ ws\ w)) \\
& \quad \{\text{letting } ws' = ws +\!\!+ [(pid, k)]\} \\
= & approx\ (n{+}1)\ (forget\ (pause\ (r\ Ack\ \texttt{>>=}_{St}\ \lambda\, k.\ \mathbf{return}_{St}\ (rrobin_i\ ws')))) \\
& \quad \{\text{defn. } pause, \text{ letting } \varphi = (r\ Ack\ \texttt{>>=}_{St}\ \lambda k.\ \mathbf{return}_{St}\ (rrobin_i\ ws')\} \\
= & approx\ (n{+}1)\ (forget\ (Ps\ Null\ \varphi)) \\
& \quad \{\text{defn. } forget\} \\
= & approx\ (n{+}1)\ (Pause\ (\varphi\ \texttt{>>=}_{St}\ (\mathbf{return}_{St}\ \circ\ forget))) \\
& \quad \{\text{assoc. and left unit monad laws}\} \\
= & approx\ (n{+}1)\ (Pause\ (r\ Ack\ \texttt{>>=}_{St}\ \lambda\, k.\ \mathbf{return}_{St}\ (forget\ (rrobin_i\ ws')))) \\
& \quad \{\text{defn. } \circ\} \\
= & approx\ (n{+}1)\ (Pause\ (r\ Ack\ \texttt{>>=}_{St}\ \lambda\, k.\ \mathbf{return}_{St}\ (forget \circ rrobin_i\ ws'))) \\
& \quad \{\text{similarly by the defn. of } approx\} \\
= & Pause\ (r\ Ack\ \texttt{>>=}_{St}\ \lambda\, k.\ \mathbf{return}_{St}\ (approx\ n \circ forget \circ rrobin_i\ ws'))
\end{array}
$$

By similar reasoning, it follows that:

$$
\begin{array}{ll}
& approx\ (n{+}1)\ (rrobin\ (w : ws)) \\
= & Pause\ (r\ Ack\ \texttt{>>=}_{St}\ \lambda\, k.\ \mathbf{return}_{St}\ (approx\ n \circ rrobin\ ws'))
\end{array}
$$

We know by the induction hypothesis that

$$(approx\ n \circ forget \circ rrobin_i)\ ws' \ =\ (approx\ n \circ rrobin)\ ws'$$

Therefore,

$$approx\ (n{+}1)\ (forget \circ rrobin_i\ (w : ws)) \ =\ approx\ (n{+}1)\ (rrobin\ (w : ws))$$

By Theorem 2, we are done. □

7.4 Generating System Traces

The next step is to generate the schedulings associated with the kernel. This is a simple matter with lazy lists, and the function *trace* (defined below) does so. Also, two technical lemmas characterizing the interaction between *trace* and the instrumented scheduler and handler are stated and proved; these are useful in the subsequent proof of fairness.

The function *trace* takes a store and an *Ro* computation and collects all observations within the computation into a stream in order of occurrence. Note also that the store σ is passed in the single-threaded manner:

$$\begin{array}{l}
trace \quad\quad\quad\quad :: Sto\ \rightarrow\ Ro\ a\ \rightarrow\ [Obs] \\
trace\ _\ (Dn\ v) \quad = [] \\
trace\ \sigma\ (Ps\ o\ \varphi) = \textbf{case}\ o\ \textbf{of} \\
\qquad\qquad\qquad\qquad (Sched\ ws\ r) \rightarrow\ o\ :\ os \\
\qquad\qquad\qquad\qquad Null \qquad\quad\ \rightarrow\ os \\
\quad \textbf{where} \\
\qquad os\ =\ let\ (r, \sigma')\ =\ (run\ \varphi\ \sigma)\ in\ trace\ \sigma'\ r
\end{array}$$

It is apparent that $trace\ \sigma\ (rrobin_i\ ws)$ is a scheduling for all σ and ws.

Lemma 7.1 captures the interaction between *trace* and $rrobin_i$. The lemma shows that the trace of a $rrobin_i$ call has an appropriate scheduling observation at its head and the trace of a $handler_i$ call as its tail.

Lemma 7.1 (*trace-rrobin*)
Let σ::*Sto* be any store and, for $1 \leq j \leq n$, let t_j be any threads and i_j be unique process identifiers. Then,

$$trace\ \sigma\ (rrobin_i\ [(i_1, t_1), \ldots, (i_n, t_n)]) \quad = \quad o : os$$

where $o \ = \ (Sched\ [i_2, \ldots, i_n]\ i_1)$
$os \ = \ trace\ \sigma\ (handler_i\ [(i_2, t_2), \ldots, (i_n, t_n)]\ (i_1, t_1))$

Proof
Let $f \ = \ \lambda\ \_.\ handler_i\ [t_2, \ldots, t_n]\ t_1$ and $o \ = \ (Sched\ [i_2, \ldots, i_n]\ i_1)$ below.

$$\begin{array}{l}
trace\ \sigma(rrobin_i\ [t_1, \ldots, t_n]) \\
\quad = trace\ \sigma\ (observe\ o\ \texttt{>>}_{Ro}\ handler_i\ [t_2, \ldots, t_n]\ t_1) \\
\{\text{defn.}\ observe\} \\
\quad = trace\ \sigma\ ((Ps\ o\ (\textbf{return}_{St}\ (\textbf{return}_{Ro}\ nil)))\ \texttt{>>=}_{Ro}\ f) \\
\{\text{defn.}\ \texttt{>>=}_{Ro}\}
\end{array}$$

$= \mathit{trace}\ \sigma\ (\mathit{Ps}\ o\ (\mathbf{return}_{St}\ (\mathbf{return}_{Ro}\ \mathit{nil})\ \texttt{>>=}_{St}\ \lambda\, r.\, \mathbf{return}_{St}(\ r\ \texttt{>>=}_{Ro}\ f))$
{left unit for St}
$= \mathit{trace}\ \sigma\ (\mathit{Ps}\ o\ (\mathbf{return}_{St}(\ \mathbf{return}_{Ro}\ \mathit{nil}\ \texttt{>>=}_{Ro}\ f)))$
{left unit for Ro}
$= \mathit{trace}\ \ \sigma\ \ (\mathit{Ps}\ o\ (\mathbf{return}_{St}(\mathit{handler}_i\ \ [t_2, \ldots, t_n]\ \ t_1))$
{defn. *trace*}
$= o\ :\ \mathit{os}$
where
$\mathit{os} =$ **let** $(r, \sigma_1) = (\mathit{run}\ (\mathbf{return}_{St}(\mathit{handler}_i\ [t_2, \ldots, t_n]\ t_1))\ \sigma)$
in $\mathit{trace}\ \ \sigma_1\ \ r$

Because $\mathit{run}\ (\mathbf{return}_{St}\ x)\ \sigma\ =\ (x, \sigma)$ by Theorem 1, the definition of *os* simplifies to: $\mathit{os}\ =\ \mathit{trace}\ \sigma\ (\mathit{handler}_i\ [t_2, \ldots, t_n]\ t_1)$. □

Lemma 7.2 captures the interaction between *trace* and $\mathit{handler}_i$. Simply put, the lemma shows that the trace of a $\mathit{handler}_i$ call reduces to the trace of a rrobin_i call.

Lemma 7.2 (*trace-handler*)

(i) $\mathit{trace}\ \sigma\ (\mathit{handler}_i\ \mathit{wl}\ (i, P(\mathit{Cont}, r))) \ =\ \mathit{trace}\ \hat{\sigma}\ (\mathit{rrobin}_i\ (\mathit{wl}{+\!\!+}[(i, \hat{r})]))$
for $(\hat{r}, \hat{\sigma})\ =\ \mathit{run}\ (r\ \mathit{Ack})\ \sigma$.

(ii) $\mathit{trace}\ \sigma\ (\mathit{handler}_i\ \mathit{wl}\ (i, D\ \_)) \ =\ \mathit{trace}\ \sigma\ (\mathit{rrobin}_i\ \mathit{wl})$

(iii) $\mathit{trace}\ \sigma\ (\mathit{handler}_i\ \mathit{wl}\ (i, P(\mathit{ForkReq}\ c, r)))$
$=\ \mathit{trace}\ \sigma\ (\mathit{rrobin}_i\ (\mathit{wl}{+\!\!+}[(i, \kappa), (i^*, \mathit{cmd}\ c)]))$
where $\kappa = \mathit{cont}\ r\ \mathit{ForkRsp}$ and i^* is a new process identifier.

(iv) $\mathit{trace}\ \sigma\ (\mathit{handler}_i\ \mathit{wl}\ (i, P(q, r))) \ =\ \mathit{trace}\ \sigma(\mathit{rrobin}_i\ \mathit{wl}{+\!\!+}[(i, \kappa)])$
where q is a *Req* not handled above and κ is one of $P(\mathit{Cont}, r)$,
$P(\mathit{GetSemReq},\ r)$, $P(\mathit{RecvReq},\ r)$, $\mathit{cont}\ r\ \mathit{Ack}$,
$\mathit{cont}\ r\ (\mathit{RecvRsp}\ m)$ for some $m :: \mathit{Message}$, or $\mathit{cont}\ r\ (\mathit{GetPIDRsp}\ i)$.

Proof

For part (i), assume *action* and *next* are defined as follows:

$$\begin{aligned}\mathit{action} &= (r\ \mathit{Ack})\ \texttt{>>=}_{St}\ \lambda\, k.\, \mathbf{return}_{St}\ (\mathit{next}\ (i, k))\\ \mathit{next}\ t\ &= \mathit{rrobin}_i\ (\mathit{ts}{+\!\!+}[t])\end{aligned}$$

Then, by the definitions of *pause* and *trace*:

$$\begin{aligned}\mathit{trace}\ \sigma\ (\mathit{handler}_i\ \mathit{wl}\ (i, P(\mathit{Cont}, r))) &= \mathit{trace}\ \sigma\ (\mathit{pause}\ \mathit{action})\\ &= \mathit{trace}\ \sigma\ (\mathit{Ps}\ \mathit{Null}\ \mathit{action})\\ &= \mathit{trace}\ \sigma'\ r'\end{aligned}$$

where $(r', \sigma')\ =\ \mathit{run}\ \mathit{action}\ \sigma$. Substituting the value of *action* on the r.h.s:

$$\begin{aligned}\mathit{run}\ \mathit{action}\ \sigma &= \mathit{run}\ ((r\ \mathit{Ack})\ \texttt{>>=}_{St}\ \lambda\, k.\, \mathbf{return}_{St}\ (\mathit{next}\ (i, k)))\ \sigma\\ &= \mathit{run}\ (\mathbf{return}_{St}\ (\mathit{next}\ (i, \hat{r})))\ \hat{\sigma}\\ &= (\mathit{next}\ (i, \hat{r}), \hat{\sigma})\end{aligned}$$

by Theorem 1 for $(\hat{r}, \hat{\sigma})\ =\ \mathit{run}\ (r\ \mathit{Ack})\ \sigma$.

Because $r' = next\ (i, \hat{r})$ and $\sigma' = \hat{\sigma}$:

$$\begin{aligned} trace\ \sigma\ (handler_i\ wl\ (i, P(Cont, r))) &= trace\ \sigma'\ r' \\ &= trace\ \hat{\sigma}\ (next\ (i, \hat{r})) \\ &= trace\ \hat{\sigma}\ (rrobin_i\ (ts{+\!\!+}[(i, \hat{r})])) \end{aligned}$$

Part (ii) follows immediately from the definition of *trace*. The proofs of parts (iii) and (iv) proceed along a similar, albeit simpler, line as part (i). □

7.5 Proving Fairness

Now we may prove the fairness of $rrobin_i$ in short order.

Theorem 4 ($rrobin_i$ *is fair*)
The scheduling $s = trace\ \sigma\ (rrobin_i\ [(i_1, t_1), ..., (i_n, t_n)])$ is fair for any store σ::*Sto* and any threads t_j.

Proof
It suffices to show that, for each $(Sched\ ws\ i) \in s$ and $w \in ws$, w is live; this follows from Lemmas 7.1 and 7.2 by induction on the length of ws. If $n = 0$, then $s = trace\ \sigma\ (rrobin_i\ []) = []$ and, hence is fair. If $n > 0$ then, by n applications of Lemmas 7.1 and 7.2:

$$\begin{aligned} &trace\ \sigma\ (rrobin_i\ [(i_1, t_1), ..., (i_n, t_n)]) \\ &= [Sched\ [i_2, \ldots, i_n]\ i_1, \ldots, Sched\ [i_1, \ldots, i_{n-1}]\ i_n]{+\!\!+}os \end{aligned}$$

for some *os* (and W.L.O.G. ignoring the effects of fork requests on the wait list). Therefore, each of $i_1, \ldots, i_n$ is live in s. □

8 Future Work and 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. Resumptions have remained, until now, primarily of interest to language theorists. This is a shame, because resumptions—especially in monadic form—are a natural and beautiful organizing principle for concurrency: they capture exactly what one needs to write and think about concurrent programs—and no more!

Resumption monads are both an expressive programming tool for concurrent applications and a foundation for their subsequent verification. To demonstrate the usefulness of resumption monads as a programming abstraction for concurrent, reactive systems, we have presented an exemplary operating system kernel supporting a broad range of behaviors. All of the behaviors typically provided by an operating system kernel may be easily and succinctly implemented using resumption monads and one may verify the resulting programs with straightforward, equational reasoning. Simplicity was a necessary design goal for the kernel as it is meant to show 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., the semaphore implementation relies on an inefficient busy-waiting strategy and the message broadcast is too simple to be of practical use). The kernel is not intended to be useful in and of itself, but rather to provide a starting point from which efficient concurrent applications may be designed, implemented, and verified.

The framework for reactive concurrency developed here has been applied to such seemingly diverse purposes as language-based security (Harrison *et al.*, 2003) and bioinformatics (Harrison & Harrison, 2004); each of these applications is an instance of this framework. The main difference lies in the request and response data types *Req* and *Rsp*. Consider the subject of (Harrison & Harrison, 2004), 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. One can easily imagine kernel instances for other concurrent system software; for example, a garbage collector could be written in the instance:

data *Req* = *Cont* | *AllocReq*
data *Rsp* = *Ack* | *AllocRsp* ⟨heap address⟩

Here, a thread may request allocation on the heap, and the kernel may grant the request by returning a pointer to the allocated memory. This kernel instance maintains a free list and collects garbage when necessary.

Instances of this kernel, being written in terms of the signature of a layered monad, inherit the software engineering benefits from 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 6, for example, one merely extends the *Req* and *Rsp* types with a new request and response and adds a corresponding *handler* definition.

One promising application for this work may be as a denotational foundation for the "Awkward Squad" (Peyton Jones, 2000) as it occurs in Haskell 98 and Concurrent Haskell/GHC: concurrency, shared state, and exceptions. That the shared-state concurrency of Concurrent Haskell could be specified within this framework must surely at this point be plausible. The exception mechanisms of Concurrent Haskell—especially asynchronous exceptions (Marlow *et al.*, 2001; Peyton Jones, 2000)—seem to fit well into the reactive concurrency paradigm.

The proof technique presented in Section 7 exploits the stream-like nature of resumption computations, closely resembling abstract interpretation (Abramsky & Hankin, 1987). With this method, the scheduling of threads by the kernel is projected into a stream of "snapshots" characterizing the aspects of the system state relevant to the system specification, which may then be verified using techniques for proving properties of streams, algebraic properties of layered monads and their non-proper morphisms, and techniques specific to resumption computations (such as Theorem 2). Monad transformers are well-known for promoting software engineering concerns like modularity and extensibility, but the layered monadic approach has benefits with respect to formal verification as well. Generating this snapshot stream is accomplished via a simple refinement to the basic resumption monad transformer: by introducing a new effect *observe*, the relevant aspects of the system state may be captured. Monadic encapsulation of effects guarantees that introducing *observe* had no impact on the kernel's operation.

The current work arose as part of a design for a kernel for multi-level security applications with a verified non-interference-style security property (Harrison *et al.*, 2003). Non-interference (Goguen & Meseguer, 1990) is a classic formal model of security in which system security is formulated in terms of streams of abstract system "events". The proof technique presented in Section 7 provides a direct link to non-interference-based security models via the abstract interpretation of resumption computations—this is part of the motivation for that technique. In this article, the verified property addressed liveness rather than security, but the principle is the same: by projecting system executions into streams of "snapshots," one may characterize the aspects of the system state relevant to the system specification.

This work demonstrates how layered resumption monads can play an essential rôle in the formal development of concurrent applications. While we have emphasized resumption monads as a programming tool, it should not be forgotten that layered monads are mathematical constructions as well; in fact, resumption monads *are* a mathematical theory of concurrency with a long pedigree. This firm foundation supports verification of programs through the mathematics underlying the programs; in our example, no other formalisms (e.g., temporal logic) were needed—powerful properties of the constructions themselves were sufficient and effective.

Layered monads play a dual rôle here as a mathematical structure and as a programming abstraction. This duality supports both executability and precise reasoning—as well as the software engineering benefits of modularity and extensibility. Taken as a whole, this approach constitutes what is sometimes called a *formal methodology* (Manna & Pnueli, 1991) for concurrent programming; that is, a specification language combined with a repertoire of related proof techniques.

References

Abramsky, Samson, & Hankin, Chris. (1987). *Abstract Interpretation of Declarative Languages*. Ellis Horwood.

Alexander, D., Arbaugh, W., Hicks, M., Kakkar, P., Keromytis, A., Moore, J., Gunder, C., Nettles, S., & Smith, J. (1998). The switchware active network architecture. *IEEE Network*, May/June.

Andrews, Gregory R., & Olsson, Ronald A. (1993). *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings.

Armstrong, Joe, Virding, Robert, Wikström, Claes, & Williams, Mike. (1996). *Concurrent Programming in Erlang*. Second edn. Prentice-Hall.

Bakker, J.W. de. (1980). *Mathematical Theory of Program Correctness*. International Series in Computer Science. Prentice-Hall.

Bakker, J.W. de, & Vink, E.P. de. (1996). *Control Flow Semantics*. Foundations of Computing Series. The MIT Press.

Barr, Michael, & Wells, Charles. (1990). *Category Theory for Computing Science*. Prentice Hall.

Bird, Richard, & Wadler, Phillip. (1988). *Introduction to Functional Programming*. Prentice Hall.

Birman, K., Constable, R., Hayden, M., Kreitz, C., Rodeh, O., van Renesse, R., & Vogels, W. 2000 (January). The Horus and Ensemble projects: Accomplishments and Limitations. *Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX '00)*.

Carter, David. 1994 (January). *Deterministic Concurrency*. Ph.D. thesis, Department of Computer Science, University of Bristol.

Claessen, Koen. (1999). A poor man's concurrency monad. *Journal of Functional Programming*, **9**(3), 313–323.

Cooper, Eric C., & Morrisett, J. Gregory. (1990). *Adding Threads to Standard ML*. Tech. rept. CMU-CS-90-186. Pittsburgh, PA.

Cupitt, John. 1992 (October). *The Design and Implementation of an Operating System in a Functional Language*. Ph.D. thesis, Computing Laboratory, University of Kent at Canterbury.

Deitel, H. M. (1982). *An Introduction to Operating Systems*. Addison-Wesley.

Dijkstra, Edsger W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, **18**(8), 453–457.

Emerson, E. Allen. (1990). Temporal and modal logics. *Chap. 16, pages 995–1072 of:* van Leeuwen, J. (ed), *Handbook of theoretical computer science*, vol. B. Elsevier Science Publishers B.V.

Espinosa, David. (1995). *Semantic Lego*. Ph.D. thesis, Columbia University.

Filinski, Andrzej. (1994). Representing monads. *Pages 446–457 of: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM Press.

Filinski, Andrzej. (1996). *Controlling effects*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

Filinski, Andrzej. (1999). Representing layered monads. *Pages 175–188 of: Proceedings of the 26st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM Press.

Flanagan, Cormac, & Qadeer, Shaz. (2003). Types for atomicity. *Pages 1–12 of: Proceed-*

ings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI '03). ACM Press.

Flatt, Matthew, Findler, Robert Bruce, Krishnamurthi, Shriram, & Felleisen, Matthias. (1999). Programming languages as operating systems (or revenge of the son of the lisp machine). *Pages 138–147 of: Proceedings of the 4th ACM International Conference on Functional Programming (ICFP)*. ACM Press.

Ganz, Steven E., Friedman, Daniel P., & Wand, Mitchell. (1999). Trampolined style. *Pages 18–27 of: Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*. ACM Press.

Gibbons, Jeremy, & Hutton, Graham. 2004 (Mar.). *Proof Methods for Corecursive Programs. Submitted to:* Fundamenta Informaticae Special Issue on Program Transformation.

Girard, Jean-Yves. (1987). Linear Logic. *Theoretical Computer Science*, **50**, 1–102.

Goguen, Joseph A., & Meseguer, José. (1990). Security policies and security models. *Pages 11–20 of: Proceedings of the 1982 Symposium on Security and Privacy (SSP '82)*. IEEE Computer Society Press.

Gunter, Carl A. (1992). *Semantics of programming languages: Programming techniques*. Cambridge, Massachusetts: The MIT Press.

Harper, Robert, Lee, Peter, & Pfenning, Frank. 1998 (January). *The Fox project: Advanced language technology for extensible systems*. Tech. rept. CMU-CS-98-107. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. (Also published as Fox Memorandum CMU-CS-FOX-98-02).

Harrison, William, Tullsen, Mark, & Hook, James. 2003 (June). Domain separation by construction. *LICS Satellite Workshop on Foundations of Computer Security (FCS03)*.

Harrison, William L., & Harrison, Robert W. 2004 (September). Domain specific languages for cellular interactions. *Proceedings of the 26th Annual IEEE International Conference on Engineering in Medicine and Biology*.

Haynes, Christopher T., & Friedman, Daniel P. (1984). Engines build process abstractions. *Pages 18–24 of: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. ACM Press.

Henderson, Peter. (1982). Purely functional operating systems. *Pages 177–191 of:* Darlington, J., Henderson, P., & Turner, D. (eds), *Functional Programming and Its Applications: an Advanced Course*. Cambridge University Press.

Hoare, C. A. R. (1978a). Communicating sequential processes. *Communications of the ACM*, **21**(8), 666–677. See corrigendum (Hoare, 1978b).

Hoare, C. A. R. (1978b). Corrigendum: "Communicating Sequential Processes". *Communications of the ACM*, **21**(11), 958–958.

Jacobs, Bart, & Poll, Erik. (2003). Coalgebras and Monads in the Semantics of Java. *Theoretical Computer Science*, **291**(3), 329–349.

Krstic, Sava, Launchbury, John, & Pavlovic, Dusko. (2001). Categories of processes enriched in final coalgebras. *Pages 303–317 of: Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*. Springer-Verlag.

Launchbury, John, & Peyton Jones, Simon L. (1994). Lazy functional state threads. *Pages 24–35 of: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press.

Liang, Sheng. (1998). *Modular Monadic Semantics and Compilation*. Ph.D. thesis, Yale University.

Liang, Sheng, Hudak, Paul, & Jones, Mark. (1995). Monad transformers and modu-

lar interpreters. *Pages 333–343 of: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press.

Lin, Albert C. 1998 (February). *Implementing Concurrency for an ML-based Operating System*. Ph.D. thesis, Massachusetts Institute of Technology.

Loeckx, Jacques, Ehrich, Hans-Dieter, & Wolf, Markus. (1996). *Specification of Abstract Data Types*. New York, USA: Wiley & Teubner.

Manna, Zohar, & Pnueli, Amir. (1991). *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag.

Marlow, Simon, Jones, Simon L. Peyton, Moran, Andrew, & Reppy, John H. (2001). Asynchronous exceptions in Haskell. *Pages 274–285 of: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Misra, Jayadev, Chandy, K. Mani, & Smith, Todd. (1982). Proving safety and liveness of communicating processes with examples. *Pages 201–208 of: Proceedings of the 1st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. ACM Press.

Moggi, Eugenio. (1990). *An Abstract View of Programming Languages*. Tech. rept. ECS-LFCS-90-113. Department of Computer Science, Edinburgh University.

Owicki, Susan, & Lamport, Leslie. (1982). Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, **4**(3), 455–495.

Papaspyrou, Nikos S. 1998 (February). *A Formal Semantics for the C Programming Language*. Ph.D. thesis, National Technical University of Athens, Department of Electrical and Computer Engineering, Software Engineering Laboratory.

Papaspyrou, Nikos S. (2001). A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. *Proceedings of the 3rd Panhellenic Logic Symposium*. An expanded version is available as a technical report from the author by request.

Papaspyrou, Nikos S., & Maćoš, Dragan. (2000). A Study of Evaluation Order Semantics in Expressions with Side Effects. *Journal of Functional Programming*, **10**(3), 227–244.

Peyton Jones, Simon. (2000). Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell. *Pages 47–96 of: Engineering Theories of Software Construction*. NATO Science Series, vol. III 180. IOS Press.

Peyton Jones, Simon (ed). (2003). *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press.

Peyton Jones, Simon, & Wadler, Phillip. (1993). Imperative functional programming. *Pages 71–84 of: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press.

Peyton Jones, Simon, Gordon, Andrew, & Finne, Sigbjorn. (1996). Concurrent haskell. *Pages 295–308 of: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press.

Plasmeijer, Rinus, & van Eekelen, Marko. 1998 (June). *The Concurrent Clean Language Report*. Technical Report CSI-R9816. Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands.

Plotkin, Gordon D. (1976). A Powerdomain Construction. *SIAM Journal of Computation*, **5**(3), 452–487.

Pnueli, Amir. (1977). The temporal logic of programs. *Pages 46–57 of: Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS)*.

Roscoe, William A. (1998). *Theory and Practice of Concurrency*. Prentice-Hall.

Schmidt, David A. (1986). *Denotational Semantics: A Methodology for Language Development*. Boston: Allyn and Bacon.

Smyth, Michael B. (1978). Powerdomains. *Journal of Computer and System Sciences*, **16**(1), 23–36.

Spiliopoulou, Eleni. 1999 (January). *Concurrent and Distributed Functional Systems.* Tech. rept. CS-EXT-1999-240. University of Bristol.

Stoye, William. (1984). *A New Scheme for Writing Functional Operating Systems.* Tech. rept. 56. Computing Laboratory, Cambridge University.

Stoye, William. (1986). Message-based Functional Operating Systems. *Science of Computer Programming*, **6**(3), 291–311.

Tolmach, Andrew, & Antoy, Sergio. 2003 (June). A monadic semantics for core curry. *Proceedings of the 12th International Workshop on Functional and (Constraint) Logic Programming.*

Turner, David. (1987). Functional programming and communicating processes. *Pages 54–74 of: Proceedings of Parallel Architectures and Languages Europe (PARLE).* Lecture Notes in Computer Science, vol. 259. Springer-Verlag.

Turner, David A. (1990). An Approach to Functional Operating Systems. *Pages 199–217 of:* Turner, David A. (ed), *Research Topics in Functional Programming.* Addison-Wesley Publishing Company.

van Weelden, Arjen, & Plasmeijer, Rinus. (2002). Towards a strongly typed functional operating system. *Proceedings of the 14th International Workshop on the Implementation of Functional Languages (IFL).* Lecture Notes in Computer Science, vol. 2670. Springer-Verlag.

Wadler, Philip. (1992). The essence of functional programming. *Pages 1–14 of: Proceedings of the 19th Symposium on Principles of Programming Languages (POPL).* ACM Press.

Wadler, Phillip. (1995). Monads for functional programming. *Pages 24–52 of: Proceedings of the 1992 Marktoberdorf International Summer School on Logic of Computation.* Lecture Notes in Computer Science, vol. 925.

Wand, Mitchell. (1980). Continuation-based multiprocessing. *Pages 19–28 of: Proceedings of the 1980 ACM Conference on LISP and Functional Programming.* ACM Press.

A Code Listing for Instrumented Handler

```
pause :: St (Ro a) → Ro a
pause = Ps Null

handler_i :: System a → Thread a → Ro a
handler_i (ts, mQ, mutex, out, pgen) (pid, D _)          = rrobin_i (ts, mQ, mutex, out, pgen)
handler_i (ts, mQ, mutex, out, pgen) (pid, P(Cont, r)) = pause action
  where
    action =(r Ack) >>=_St λ k. return_St (next (pid, k))
    next t =rrobin_i (ts++[t], mQ, mutex, out, pgen)
handler_i (ts, mQ, mutex, out, pgen) (pid, P(PrintReq msg, r))  = pause (return_St next)
  where
    next = rrobin_i (ts++[(pid, P(Cont, r))], mQ, mutex, out++msg, pgen)
handler_i (ts, mQ, mutex, out, pgen) (pid, P(SleepReq, r))      = pause (return_St next)
  where
    next =rrobin_i (ts++[(pid, P(Cont, r))], mQ, mutex, out, pgen)
handler_i (ts, mQ, mutex, out, pgen) (pid, P(ForkReq c, r))     = pause (return_St next)
  where
    parent = (pid, cont r ForkRsp)
    child  = (pgen, cmd c)
    next   = rrobin_i (ts++[parent, child], mQ, mutex, out, pgen+1)
handler_i (ts, mQ, mutex, out, pgen) (pid, P(BcastReq m, r))    = pause (return_St next)
  where
    next =rrobin_i (ts++[(pid, cont r Ack)], mQ++[m], mutex, out, pgen)
handler_i (ts, [], mutex, out, pgen) (pid, P(RecvReq, r))       = pause (return_St next)
  where
    next =rrobin_i (ts++[(pid, P(RecvReq, r))], [], mutex, out, pgen)
handler_i (ts, (m : ms), mutex, out, pgen) (pid, P(RecvReq, r)) = pause (return_St next)
  where
    next =rrobin_i (ts++[(pid, cont r (RecvRsp m))], ms, mutex, out, pgen)
handler_i (ts, mQ, mutex, out, pgen) (pid, P(GetSemReq, r))     =
  if mutex > 0 then
    pause ((r Ack) >>=_St λ k. return_St (rrobin_i (ts++[(pid, k)], mQ, mutex−1, out, pgen)))
  else
    pause (return_St (rrobin_i (ts++[(pid, P(GetSemReq, r))], mQ, mutex, out, pgen)))
handler_i (ts, mQ, mutex, out, pgen) (pid, P(ReleaseSemReq, r)) = pause (return_St next)
  where
    next =rrobin_i (t++[cont r Ack], mQ, mutex+1, out, pgen)
handler (ts, mQ, mutex, out, pgen) (pid, P(KillReq i, r))       = pause (return_St next)
  where
    next  = rrobin (wl', mQ, mutex, out, pgen)
    wl'   = filter (exit i) (ts++[(pid, cont r Ack)])
    exit i = λ (pid, t). i ≠ pid
```

B The Branching Kernel

This appendix presents a generalization of the kernel from Section 6 to one with a *branching* notion of time (Manna & Pnueli, 1991; Pnueli, 1977). The branching kernel elaborates all possible schedulings of a set of threads, and its realization requires little more than a tiny change to the monad definitions. Specifically, this change is the use of the non-determinism monad (defined below) instead of the identity monad in the definitions of the St, R and Re monads.

Interestingly, the branching kernel illuminates the connection between the definitional interpreter presented in Section 5 and previous applications of the resumption monad to language semantics. Papaspyrou (2001) presents a categorical semantics of a language much like that of Section 5 without the signaling commands. The monad of denotation encapsulates state and basic concurrency—just as the R monad from previous sections—combined with a computational theory of non-determinism based on the powerdomain construction (Plotkin, 1976; Smyth, 1978). In Papaspyrou's semantics, the meaning of a concurrent command, $c \mid\mid c'$, elaborates all interleavings of the meanings of c and c'. This is precisely what the definitional interpreter of Section 5 does when combined with the branching kernel.

This section proceeds as follows. Section B.1 describes how non-determinism is modeled semantically and how such models are represented in functional languages via the list monad. Section B.2 shows how the non-determinism effect is combined with the state and concurrency effects. Section B.3 describes the non-deterministic scheduler.

B.1 The List Monad & Non-determinism

When a program may return multiple values, one says that it is *non-deterministic*. Consider, for example, the *amb* operator of McCarthy (1963). Given two arguments, it returns either one or the other[11]; for example, the value of $1\ amb\ 2$ is either 1 or 2. In the presence of such non-determinism, many familiar equational reasoning principles fail—one cannot even say that $1\ amb\ 2 = 1\ amb\ 2$. Referential transparency—that one can substitute "equals for equals"—is destroyed by such a non-deterministic operations. Consider, for example, the (possibly) non-equal values of "$\mathbf{let}\ x = (1\ amb\ 2)\ \mathbf{in}\ x + x$" and "$(1\ amb\ 2) + (1\ amb\ 2)$".

Semantically, non-deterministic programs may be viewed as returning sets of values rather than just one value (Bakker, 1980; Schmidt, 1986). According to this view, the meaning of $(1\ amb\ 2)$ is simply the set $\{1, 2\}$. The encoding of non-determinism as sets of values may be expressed monadically via the set monad; this monad may be expressed in pseudo-Haskell notation as:

$$\mathbf{return}\ x = \{\ x\ \} \qquad S\ \texttt{>>=}\ f = \bigcup(f\ S)$$

where $f\ S = \{\ f\ x \mid x \in S\ \}$. In the set monad, the meaning of $(e\ amb\ e')$ is the union of the meanings of e and e'.

[11] It is also *angelic*: in the case of the non-termination of one of its arguments, *amb* returns the terminating argument. For the purposes of this exposition, however, we ignore this technicality.

That lists are similar structures to sets is familiar to any functional programmer; a classic exercise in introductory functional programming courses represents sets as lists and set operations as functions on lists (in particular, casting set union ($\cup$) as list append ($+\!\!+$)). Some authors (Wadler, 1992; Wadler, 1995; Espinosa, 1995; Liang, 1998) have made use of the "sets as lists" pun to implement non-deterministic programs within functional programming languages via the list monad. The list monad (written "[]" in Haskell) is defined by the instance declaration:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ [\,]\ \textbf{where} \\
\quad \mathit{return}\ x \qquad = [x] \\
\quad (x : xs)\ \texttt{>>=}\ f = f\ x +\!\!+ (xs\ \texttt{>>=}\ f) \\
\quad [\,]\ \texttt{>>=}\ f \qquad = [\,]
\end{array}
$$

This straightforward implementation suffices for our purposes, but it is known to contain an inaccuracy when the lists involved are infinite (Tolmach & Antoy, 2003). Specifically, because $l +\!\!+ k = l$ if the list l is infinite, append ($+\!\!+$) loses information that set union ($\cup$) would not.

The non-determinism monad has a non-proper morphism, *merge*, that combines a finite number of nondeterministic computations, each producing a set of values, into a single computation returning their union. For the set monad, it is union ($\cup$), while with the list implementation, *merge* is concatenation:

$$
\begin{array}{l}
\mathit{merge}_{[]} :: [[a]] \rightarrow [a] \\
\mathit{merge}_{[]} = \mathit{concat}
\end{array}
$$

Note that the finiteness of the argument of *merge* is assumed and is not reflected in its type.

B.2 Combining Non-determinism with Other Effects

This section considers the combination of non-determinism with the state and resumption effects. The monads constructed in this section are exactly the ones that would arise through the application of monad transformers using the list monad in place of the identity monad; that is, they are defined as:

$$
\begin{array}{l}
\textbf{type}\ \mathit{St}_n\ = \mathit{StateT}\ \mathit{Sto}\ [\,] \\
\textbf{type}\ \mathit{R}_n\ = \mathit{ResT}\ \mathit{St}_n \\
\textbf{type}\ \mathit{Re}_n = \mathit{ReactT}\ \mathit{Req}\ \mathit{Rsp}\ \mathit{St}_n
\end{array}
$$

The monads constructed here are distinguished from corresponding earlier ones with a subscript ("n" for non-determinism). This section constructs the above monads "by hand" with the intention of assisting the reader.

It is enlightening to consider what the monad St_n would look like if it were constructed "by hand" (i.e., without the application of the state monad transformer to the list monad); the Haskell data type declaration to do this is:

$$
\textbf{data}\ \mathit{St}_n\ a = \ \mathit{ST}\ (\mathit{Sto}\ \rightarrow\ [(a, \mathit{Sto})])
$$

A computation in St_n, when applied to a store, returns a collection of value and

output state pairs. In our setting, this collection will always be finite, although this is obviously not expressed in the type declaration itself. In equivalent categorical language (Moggi, 1990), this monad would be written:

$$St_n\ A\ =\ (P_{fin}(A{\times}Sto))^{Sto}$$

where $(-)^{Sto}$ are maps from *Sto* into its argument and $P_{fin}(-)$ is set of finite subsets drawn from its argument.

The monad St_n has the bind, unit, update and get operations defined by the state monad transformer (i.e., `>>=`, *return*, u, g, respectively) as well as the lifted *merge* morphism, defined as:

$$\begin{aligned}&merge_{St} \quad :: [St_n\ a]{\rightarrow}St_n\ a\\ &merge_{St}\ phis = ST\ (\lambda\,\sigma.\ merge_{[]}\ (map\ (\lambda\ (ST\ \varphi).\ \varphi\ \sigma)\ phis)\end{aligned}$$

More interesting still is when *ResT* is applied to St_n; this monad is constructed "by hand" as follows:

$$\mathbf{data}\ R_n\ a = Done\ a \mid Pause\ (Sto\ \rightarrow\ [(R_n\ a, Sto)])$$

While resumptions without non-determinism resemble streams, resumptions with non-determinism resemble trees in the following sense. Given an input store, a *Pause*d computation returns a collection of resumption and result store pairs, and these pairs may be viewed as the children of the original computation. An R_n-computation of the form, $Pause(\lambda\,\sigma.\,[(r_1,\sigma_1),\ldots,(r_n,\sigma_n)])$, may be viewed as a "root" with the resumption computations produced by r_i as its children.

The *merge* operation may be lifted to R_n by the following definition.

$$\begin{aligned}&merge_R \quad :: [R_n\ a] \rightarrow R_n\ a\\ &merge_R\ ts = Pause\ (merge_{St}\ (map\ \mathbf{return}_{St_n}\ ts))\end{aligned}$$

In effect, $merge_R$ creates a new "root node" with the arguments in *ts* as its children. In Moggi's categorical notation (Moggi, 1990), R_n would be defined as the functor:

$$R_n\ A\ =\ \mu X.\,(P_{fin}((A+X){\times}S))^S$$

The by-hand construction of Re_n is analogous to that of R_n, so no further comment is necessary.

B.3 Generalizing the Kernel with Non-deterministic Scheduling

The kernel described in Section 6 chooses one scheduling out of many possible schedulings; to change this kernel so that it elaborates *all* possible schedules, only a few simple and straightforward changes are necessary. First, add the raw material for non-determinism to the system by changing the "base monad" from *Id* to []; that is, define St_n, R_n, and Re_n as described in Section B.2. The second and final step defines a non-deterministic scheduler that picks every possible waiting thread to be run next; this is accomplished with one application of the $merge_R$ morphism. The resulting scheduler—called *allscheds* and defined in this section—elaborates

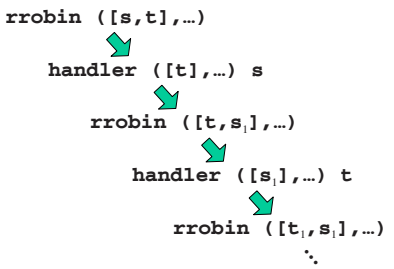

Fig. B 1. *Round-robin Schedulings*: Reflects the call graph between the *rrobin* scheduler and the *handler* routine.

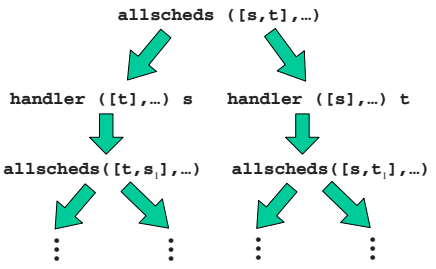

Fig. B 2. *Finite-branching Scheduler*: Reflects the call graph between the *allscheds* scheduler and the *handler* routine.

all schedules in a tree-like fashion. For the sake of comparison, Figures B 1 and B 2 portray the actions of the deterministic scheduler, *rrobin*, and the non-deterministic scheduler, *allscheds*.

The auxiliary function, *schedulings*, computes all possible scheduling choices from a waiting list. Given a waiting list $wl = [t_1, \ldots, t_n]$, (*schedulings wl*) computes each possible choice for the next thread to execute, where each choice has the form[12] $(t_i, [t_1, \ldots, t_{i-1}, t_{i+1}, t_n])$. The Haskell code for *schedulings* is:

$$
\begin{array}{l}
\mathit{schedulings} \quad :: [a] \rightarrow [(a, [a])] \\
\mathit{schedulings}\ [] \quad = [] \\
\mathit{schedulings}\ wl = \mathit{hts}\ []\ wl \\
\quad \textbf{where} \\
\quad\quad \mathit{hts}\ \mathit{front}\ [] \qquad = [] \\
\quad\quad \mathit{hts}\ \mathit{front}\ (t : ts) = (t, \mathit{front} +\!\!+ ts)\ :\ (\mathit{hts}\ (t : \mathit{front})\ ts)
\end{array}
$$

Using the merge_R operation arising from the above monad constructions and *schedulings*, it is a simple matter to define a scheduler elaborating all possible schedules. This scheduler, *allscheds*, applies *handler* to each scheduling choice computed by *schedulings* and merges the resulting executions together as portrayed in Figure B 2. The Haskell code for *allscheds* is:

$$
\begin{array}{l}
\mathit{allscheds} :: \mathit{System}\ a \rightarrow R_n\ a \\
\mathit{allscheds}\ (wl, mQ, \mathit{mutex}, \mathit{out}, \mathit{pgen}) = \mathit{merge}_R\ (\mathit{map}\ \mathit{dispatch}\ \mathit{scheds}) \\
\quad \textbf{where} \\
\quad\quad \mathit{scheds} \quad = \mathit{schedulings}\ wl \\
\quad\quad \mathit{dispatch} = \lambda\,(t, ts).\ \mathit{handler}\ (ts, mQ, \mathit{mutex}, \mathit{out}, \mathit{pgen})\ t
\end{array}
$$

Note that *handler* has been altered to call *allscheds* instead of *rrobin*, but, other than this trivial change, it is identical to the definition in Section 6.

[12] N.b., the ordering within the second component may not be identical to that of the input list.

C Proof of Approximation Lemma for Basic Resumption Computations

In this section, we prove an approximation lemma for resumption monads analogous to the approximation lemma for lists (Gibbons & Hutton, 2004). The statement and proof of the resumption approximation lemma are almost identical to those of the list case; this is perhaps not too surprising because of the analogy between lists and resumptions remarked upon earlier. Assume that, for a given monad M, that R is declared in Haskell as:

$$\textbf{data}\ R\ a = \mathit{Done}\ a \mid \mathit{Pause}\ (M\ (R\ a))$$

The Haskell function *approx* approximates R computations:

$$\begin{aligned}
&\mathit{approx}\ ::\ \mathit{Int}\ \to\ R\,a\ \to\ R\,a\\
&\mathit{approx}\ (n{+}1)\ (\mathit{Done}\ v)\ \ = \mathit{Done}\ v\\
&\mathit{approx}\ (n{+}1)\ (\mathit{Pause}\ \varphi) = \mathit{Pause}\ (\varphi\ \texttt{>>=}\ (\textbf{return}\ \circ\ \mathit{approx}\ n))
\end{aligned}$$

where `>>=` and **return** are the bind and unit operations of the monad M. Note that, for any finite resumption-computation φ, $\mathit{approx}\ n\ \varphi\ =\ \varphi$ for any sufficiently large n—that is, $(\mathit{approx}\ n)$ approximates the identity function on resumption computations. We may now state the approximation lemma for R:

Theorem 2 (*Approximation Lemma for R*)
For any $\varphi, \gamma :: Ra$, $\varphi = \gamma \Leftrightarrow$ *for all* $n \in \omega$, $\mathit{approx}\ n\ \varphi = \mathit{approx}\ n\ \gamma$.

To prove this theorem requires a denotational model of R—that is, we need to know precisely what φ, γ, *approx*, etc., are—and for this we turn to the denotational semantics of resumption computations as developed by Papaspyrou (Papaspyrou, 2001). His semantics for resumption monads applies a categorical technique for constructing denotational models of lazy data types by calculating fixed points of functors (Gunter, 1992; Barr & Wells, 1990). Using this semantics, we show that, for every simple type τ (i.e., a variable-free type such as *Int*), the approximation lemma holds for $R\,\tau$.

Assume that D and M are a fixed domain and monad[13], respectively. Assume for the sake of this proof that domain D is the model of the simple type τ. Let O denote the trivial domain $\{\bot_{\mathrm{O}}\}$. Then the following defines the functor F:

$$\begin{aligned}
&FX = D + MX\\
&Ff\ \ = [\mathit{inl}, \mathit{inr} \circ Mf]\ \text{ for }\ f\ \in\ \mathrm{Hom}(A, B)
\end{aligned}$$

F is indeed an endofunctor on the category of domains *Dom* (Papaspyrou, 2001).

Iterating functor F on O produces the following sequence of domains approximating resumption computations:

$$\begin{aligned}
F^0\mathrm{O} &= \mathrm{O} & F^3\mathrm{O} &= D\ +\ M(D\ +\ M(D\ +\ M\mathrm{O}))\\
F^1\mathrm{O} &= D\ +\ M\mathrm{O} & F^4\mathrm{O} &= D\ +\ M(D\ +\ M(D\ +\ M(D\ +\ M\mathrm{O})))\\
F^2\mathrm{O} &= D\ +\ M(D\ +\ M\mathrm{O}) & &\ \ \vdots
\end{aligned}$$

[13] The functor component of the monad M is required to be *locally continuous* (Papaspyrou, 2001), although we make no explicit use of this fact here.

These constructions approximate computations in R in the sense that the left and right injections, *inl* and *inr*, in each of the sums correspond to the *Done* and *Pause* constructors, respectively, in the declaration of R above. Each domain $F^i\mathrm{O}$ is a finite approximation of $R\,\tau$; for example, the finite R-computation *Pause* (**return** (*Done* 9)) closely resembles the element *inr* (*return* (*inl* 9)) in the domain $F^2\mathrm{O}$.

Between these approximations of $R\,\tau$, there are useful functions that extend an approximation in $F^i\mathrm{O}$ to an approximation in $F^{i+1}\mathrm{O}$ and truncate an approximation in $F^{i+1}\mathrm{O}$ to one in $F^i\mathrm{O}$; these are defined in terms of the *embedding-projection* functions ι^e and ι^p (Gunter, 1992; Schmidt, 1986):

$$\begin{array}{llll} \iota^e : \mathrm{O} \to F\mathrm{O} & & \iota^p : F\mathrm{O} \to \mathrm{O} \\ \iota^e \perp_{\mathrm{O}} & = & \perp_{F\mathrm{O}} & \iota^p x \quad = \quad \perp_{\mathrm{O}} \end{array}$$

The function $F^i(\iota^e) : F^i(\mathrm{O}) \to F^{i+1}(\mathrm{O})$ extends a length-i approximation to a length-$i{+}1$ approximation, while $F^i(\iota^p) : F^{i+1}(\mathrm{O}) \to F^i(\mathrm{O})$ truncates a length-$i{+}1$ approximation by "clipping off" the last step.

The domain for type $R\,\tau$ is formed from the collection of all infinite sequences $(\varphi_n)_{n\in\omega}$ such that $\varphi_i \in F^i\mathrm{O}$. Each component of the domain element $(\varphi_n)_{n\in\omega}$ is an approximation of its successor; this condition is expressed formally using truncation: $\varphi_i = F^i(\iota^p)\,\varphi_{i+1}$. This collection, when ordered pointwise, forms a domain (Papaspyrou, 2001).

The Haskell function *approx* is denoted by the continuous function approx defined on $(\varphi_m)_{m\in\omega}$ by:

$$\mathsf{approx}\ n\ (\varphi_m)_{m\in\omega} = (\gamma_m)_{m\in\omega} \ \text{ where } \gamma_m = \begin{cases} \varphi_m & (n < m) \\ \mathsf{ext}_{nm}\,\varphi_n & (m \geq n) \end{cases}$$

where the embedding $\mathsf{ext}_{ij} : F^i\mathrm{O} \to F^j\mathrm{O}$ is defined for naturals $i \leq j$:

$$\mathsf{ext}_{ij} = \begin{cases} id_{F^j\mathrm{O}} & (i = j) \\ \mathsf{ext}_{(i+1)j} \circ F^i(\iota^e) & (i < j) \end{cases}$$

Three things are clear from the definition of approx: for all $n \in \omega$,

$$\mathsf{approx}\ n \sqsubseteq \mathsf{approx}\ (n+1),\ \mathsf{approx}\ n \sqsubseteq id, \ \text{ and } \bigsqcup\{\mathsf{approx}\ i\} = id$$

Given these facts, the proof of the approximation lemma for resumption computations proceeds exactly as that of the list version in Gibbons and Hutton (Gibbons & Hutton, 2004); we repeat this proof for the convenience of the reader.

Proof of Theorem 2
The "$\Rightarrow$" direction follows immediately by extensionality. For the "$\Leftarrow$" direction, assume that $\mathsf{approx}\ n\ \varphi = \mathsf{approx}\ n\ \gamma$ for all $n \in \omega$.

$\therefore$ $\bigsqcup\{\mathsf{approx}\ n\ \varphi\} = \bigsqcup\{\mathsf{approx}\ n\ \gamma\}$ by extensionality.

$\therefore$ $(\bigsqcup\{\mathsf{approx}\ n\})\ \varphi = (\bigsqcup\{\mathsf{approx}\ n\})\ \gamma$ by the continuity of application.

$\therefore$ $id\ \varphi = id\ \gamma$ by the aforementioned observation.

$\therefore$ $\varphi = \gamma$ $\square$