Cheap (But Functional) Threads[†]

William L. Harrison and Adam M. Procter
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 straightforward to understand, implement, and 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.

Keywords: Monads, Resumptions, Concurrency, Functional Programming

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, underappreciated) computational model of concurrency is used to construct verifiable concurrent applications. The approach taken here is informal and should be accessible to anyone familiar with monads in Haskell[1]. And while the constructions described here occur in Haskell 98, they

[†] This article is an expanded version of the first author's paper *The Essence of Multitasking* published in the proceedings of AMAST 2006.

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

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 a 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. Resumption monads are both an expressive programming tool for concurrent applications and a foundation for their verification. The resumption-monadic structures explicated here were critical in the specification and verification of a design for a secure OS kernel (Harrison and Hook, 2009).

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 High Assurance Security Kernel (HASK) project[2] at the University of Missouri is working to develop and formally verify kernels 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 a functional languages 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.

[2] The HASK project at the University of Missouri explores the application of monadic semantics in formal methods. For more information, please consult the project webpage `http://hask.cs.missouri.edu`.

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 presents another example: how implicit concurrency in the form of a garbage collector may be encapsulated with resumption monads as well. Section 8 presents conclusions and outlines future work.

2. Related Work

2.1. Modeling Concurrency in Functional Settings

The concurrency models underlying previous applications of functional languages to concurrent system software fall broadly into four camps.

Concurrency via Non-deterministic Operators
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; see the appendix for further details.

Demand-driven Concurrency
The second camp 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 and 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.

Concurrency via First-class Continuations
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 and Plasmeijer, 2002; Li and Zdancewic, 2007) 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: all known effects may be expressed via CPS (Filinski, 1996; Filinski, 1994). Resumptions can be viewed as a disciplined form of continuations that support a stream-like reasoning principle akin to the well-known "take lemma" for streams (Bird and Wadler, 1988; Gibbons and Hutton, 2005). Readers interested in resumption-monadic verification should consult (Harrison and Hook, 2009).

Program-structuring Paradigms for Multithreading
The last camp uses program-structuring paradigms for multi-threading, the first of which is 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 second structuring technique is known as *delimited continuations*. Continuations are the meanings of whole evaluation contexts and are frequently motivated with the intuition that they denote the "rest of the program." A delimited continuation is the meaning of a "delimited" evaluation context, denoting the "first part" of the rest of the program. Delimited continuations bear a close resemblance to what we call reactive resumption monads and have recently been applied in the construction of system software (Kiselyov and Shan, 2007). Both

reactive resumptions and delimited continuations support a request and response notion of computation.

2.2. Concurrent Functional Languages

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; Biagioni et al., 2001; 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 and van Eekelen, 1998; Peyton Jones et al., 1996; Reppy, 1999; Cooper and Morrisett, 1990; Armstrong et al., 1996), but their concurrency models are built in to their run-time systems and provide no basis for 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.

2.3. Resumptions in Denotational Semantics

Resumptions are a denotational model of concurrency first introduced by Plotkin (Plotkin, 1976; Schmidt, 1986; Bakker and 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 and 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 and 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 and Maćoš, 2000; Papaspyrou, 2001; Krstic et al., 2001; Jacobs and 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 an expressive abstraction for concurrent programming and verification. 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 branching-time version of the kernel, closely resembles a published semantics (Papaspyrou, 2001). The constructions underlying the branching-time kernel are also the basis of a recent monadic semantics of asynchronous exceptions.

2.4. The "Awkward Squad"

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; printing out a character is performed by $putChar :: Char \to IO()$ for example. 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: $forkIO :: IO() \to IO()$ 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

[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`.

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). These impurities have been handled individually via various monadic constructions (consider the manifestly incomplete list (Moggi, 1990; Peyton Jones and Wadler, 1993; Papaspyrou, 1998)). The current approach combines these individual constructions into a single *layered* monad—i.e., a monad created 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. Indeed, recent work of Swierstra and Altenkirch (Swierstra and Altenkirch, 2007) applies a resumption-monadic framework (with a different, though similar, formulation) to the semantics of the *IO* monad. Furthermore, recent work of the authors (Harrison et al., 2008) describes how asynchronous exceptions (in both the most general and Haskell senses (Marlow et al., 2001)) can be modeled with the current framework extended with the monad of non-determinism.

2.5. Linearity vs. Branching Concurrency

Temporal logics (Pnueli, 1977; Emerson, 1990; Manna and 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 and Pnueli, 1991). Two notions of time—*linear* and *branching*—distinguish temporal logics (Emerson, 1990). *Linear* temporal logics 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 A 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[4]. 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:

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

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.

These operators must satisfy the monad laws:

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

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: $\mathbf{do}\ \{\ v\ \leftarrow\ x\ ;\ k\ \}\ =\ (x\ \texttt{>>=}\ \lambda\ v.\ k)$.

[4] For readers needing a more thorough introduction, we recommend Liang, Hudak, and Jones (Liang et al., 1995).

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:

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

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 *Int* type:

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

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:

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

The unit 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[5].

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 creates a monad adding single-threaded *sto*-passing to the computational raw material

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

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:

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

The bind and return of the input monad m are distinguished from those being defined by attaching a subscript (e.g., $return_m$). The monad $St\ =\ StateT\ Int\ Id$ is equivalent 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, too:

$$
\begin{array}{l}
g\ ::\ Monad\ m \Rightarrow StateT\ s\ m\ s \\
g\ =\ ST\ (\lambda\ s.\ return_m\ (s, s)) \\
u\ ::\ Monad\ m \Rightarrow (s\ \rightarrow\ s)\ \rightarrow\ StateT\ s\ m\ a \\
u\ \delta\ =\ ST\ (\lambda\ s.\ return_m\ ((), \delta\ s))
\end{array}
$$

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 to 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 $Sto\ =\ Name\ \rightarrow\ Int$. For the monad transformer, $StateT\ Sto$, two useful morphisms, $getloc$ and $setloc$, may be defined in terms of g and u as:

$$
\begin{array}{ll}
getloc & ::\ (Monad\ m)\ \Rightarrow\ Name\ \rightarrow\ StateT\ Sto\ m\ Int \\
getloc\ x & = g\ \texttt{>>=}\ \lambda\ \sigma.\ return\ (\sigma\ x) \\
setloc & ::\ (Monad\ m)\ \Rightarrow\ Name\ \rightarrow\ Int\ \rightarrow\ StateT\ Sto\ m\ () \\
setloc\ x\ v & = u\ [x \mapsto v] \\
[- \mapsto -] & ::\ (Eq\ a)\ \Rightarrow\ a\ \rightarrow\ b\ \rightarrow\ (a\ \rightarrow\ b)\ \rightarrow\ a\ \rightarrow\ b \\
[i \mapsto v]\ \sigma & = \lambda\ n.\ \textbf{if}\ i{==}n\ \textbf{then}\ v\ \textbf{else}\ (\sigma\ n)
\end{array}
$$

An application $getloc\ x$ returns the contents of location x. An application $setloc\ x\ v$ stores value v in location x. The type signatures for $getloc$ and $setloc$ mean that they may be defined for any monad m by applying the aforementioned state monad transformer. The polymor-

phic operation $[-\mapsto-]$ is used throughout to construct *Sto* transformers (i.e., $[x\mapsto v] :: \mathit{Sto} \to \mathit{Sto}$). In Section 7, it is also used to update environments.

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 and 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[6]; 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

[6] 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.

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.

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 (see below). An R-computation is either a completed computation, $(Done\ v)$, a finite sequence of $Pause$'s ending in a $Done$, $(Pause(\ldots Pause(Done\ v)\ldots))$, or an infinite sequence of $Pause$'s, $(Pause(\ldots))$. Computations in this version of R resemble streams without stream elements. The return operation for R is $Done$, while the bind operation (`>>=`), in effect, appends two R-computations. Note that the bind operator for R, `>>=`, is defined recursively.

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

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

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

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 $return_{St}$, 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:

$$\begin{aligned} &step \quad :: St\ a \to R\ a \\ &step\ x = Pause\ (x \texttt{ >>=}_{St} (return_{St} \circ Done)) \end{aligned}$$

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 following computations of type $R\ ()$ are the set of possible interleavings:

$$\begin{aligned} &step\ a_0 \texttt{ >> } step\ a_1 \texttt{ >> } step\ b_0 \\ &step\ a_0 \texttt{ >> } step\ b_0 \texttt{ >> } step\ a_1 \\ &step\ b_0 \texttt{ >> } step\ a_0 \texttt{ >> } step\ a_1 \end{aligned}$$

where >> is the null bind operation of the R monad. While the stream version implicitly uses a lazy cons operation $(h : t)$, the monadic version uses something analogous: $(step\ h) \texttt{ >> } t$. 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 (see below). The particular formulation of the basic resumption monad and monad transformer we use are due to Papaspyrou (Papaspyrou, 1998), although others exist as well (Moggi, 1990; Espinosa, 1995; Filinski, 1999).

$$\begin{aligned} &\textbf{data}\ ResT\ m\ a\ =\ Done\ a \mid Pause\ (m\ (ResT\ m\ a)) \\ &\textbf{instance}\ (Monad\ m)\ \Rightarrow\ Monad\ (ResT\ m)\ \textbf{where} \\ &\quad return\ \ \ \ \ \ \ \ \ \ \ \ = Done \\ &\quad (Done\ v) \texttt{ >>= } f\ \ = f\ v \\ &\quad (Pause\ r) \texttt{ >>= } f = Pause\ (r \texttt{ >>=}_m \lambda\, k\ .\ return_m\ (k \texttt{ >>= } f)) \\ &step\ ::\ (Monad\ m)\ \Rightarrow\ m\ a\ \to\ ResT\ m\ a \\ &step\ x\ =\ Pause\ (x \texttt{ >>=}_m (return_m\ \circ\ Done)) \end{aligned}$$

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[7] 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 ready 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.

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

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

Both *Req* and *Rsp* must have certain minimal structure; the continue request, *Cont*, signifies 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:

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

[7] A *reactive* program (Manna and 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.

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:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathit{Re}\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{return}\ v & = D\ v \\
D\ v\ \texttt{>>=}\ f & = f\ v \\
P\ (q, r)\ \texttt{>>=}\ f & = P\ (q,\ \lambda\ s.\ (r\ s)\ \texttt{>>=}_{St}\ \lambda\ k\ .\ \mathit{return}_{St}\ (k\ \texttt{>>=}\ f))
\end{array}
\end{array}
$$

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 and Wells, 1990). At this point, it may be helpful for some readers to compare such notation to the Haskell monad declarations:

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

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.

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):

$$
\begin{array}{ll}
\textbf{type}\ \mathit{Message} & = \mathit{Int} \\
\textbf{type}\ \mathit{PID} & = \mathit{Int} \\
\textbf{data}\ \mathit{Req} & = \mathit{Cont} \mid \mathit{Sleep}_q \mid \mathit{Fork}_q\ \mathit{Comm} \mid \mathit{Bcst}_q\ \mathit{Message} \\
 & \quad \mid \mathit{Rcv}_q \mid V_q \mid P_q \mid \mathit{Prnt}_q\ \mathit{String} \\
 & \quad \mid \mathit{PID}_q \mid \mathit{Kill}_q\ \mathit{PID} \\
\textbf{data}\ \mathit{Rsp} & = \mathit{Ack} \mid \mathit{Rcv}_r\ \mathit{Message} \mid \mathit{PID}_r\ \mathit{PID}
\end{array}
$$

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:

$$
\textbf{data}\ \mathit{ReactT}\ q\ r\ m\ a\ =\ D\ a \mid P\ (q,\ r \to (m\ (\mathit{ReactT}\ q\ r\ m\ a)))
$$

The corresponding instance declaration is as follows.

$$\begin{aligned}&\textbf{instance}\ (\mathit{Monad}\ m)\ \Rightarrow\ \mathit{Monad}\ (\mathit{ReactT}\ q\ r\ m)\ \textbf{where}\\&\quad \mathit{return}\ v\ \ \ \ \ \ \ \ = D\ v\\&\quad (D\ v)\ \texttt{>>=}\ f\ \ \ \ = f\ v\\&\quad P\ (q, r)\ \texttt{>>=}\ f = P\ (q,\ \lambda\ c\ .\ (r\ c)\ \texttt{>>=}_m\ \lambda\ k\ .\ \mathit{return}_m\ (k\ \texttt{>>=}\ f))\end{aligned}$$

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

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

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:

$$\begin{aligned}&\mathit{signal}\ \ \ :: \mathit{Req}\ \rightarrow\ \mathit{Re}\ \mathit{Rsp}\\&\mathit{signal}\ q = P(q,\ \mathit{return}_{St} \circ \mathit{return})\end{aligned}$$

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

$$\begin{aligned}&\mathit{signalI}\ \ \ :: \mathit{Req}\ \rightarrow\ \mathit{Re}\ ()\\&\mathit{signalI}\ q = P\ (q,\ \lambda\ \_\ .\ \mathit{return}_{St}\ (\mathit{return}_{Re}\ ()))\end{aligned}$$

4.3. Time Complexity of $\texttt{>>=}_R$ and $\texttt{>>=}_{Re}$

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

5. What is a "thread" anyway?

Operating systems texts (for example (Deitel, 1982)) define threads as lightweight processes[8] executed in the same address space. Some concurrent programming languages (notably CSP (Hoare, 1978) and SR (Andrews and Olsson, 1993)) also contain a notion of threads. But

[8] Throughout, we do not distinguish threads from processes.

```
type Name    = String
data Prog    = PL [Comm]
data Exp     = Plus Exp Exp | Var Name | Lit Int | 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/value
             | Psem                    — acquire semaphore
             | Vsem                    — release semaphore
             | Sleep                   — suspend execution
             | Fork Comm               — creates thread
             | Broadcast Name          — broadcasts variable
             | Receive Name            — receives avail. msg
             | Kill Exp                — terminates arg.
```

Figure 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.

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 :: St Sto                 g :: St Sto          g :: St Sto
u :: (Sto→Sto)→St a         u :: (Sto→Sto)→St a  u :: (Sto→Sto)→St a
                            step :: St a→R a     step :: St a→Re a
                                                 signal :: Req→Re Rsp
```

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

```
mexp :: Exp → Re Int
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 PIDq >>=
    λ(PIDr 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
```

Figure 2. Semantics for the Language of Threads: Expressions

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. To guarantee termination of atomic actions, we restrict the notion of atomic action to Haskell terms of type $St\ a$ constructed from *getloc*, $setloc\ x\ v$, $>>=_{St}$, and $return_{St}$ 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 $return_{Re}$ without *undefined* or *error*. A typical infinite thread starts with:

```
signal Rcvq >>= λ(Rcvr m).
      step (setloc x m) >>
           step (getloc x) >>= ···
```

Note that the restricted use of the store (i.e., using only combinations of *getloc* and $setloc\ x\ v$) guarantees the single-threadedness (in the sense of (Flanagan and 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.

```
store              :: Name→Int→Re a
store loc v        =  step (setloc loc v)

prog               :: Prog → [Re a]
prog (PL cs)       =  map cmd cs

cmd                :: Comm → Re a
cmd Skip           =  return ()
cmd (Assign x e)   =  (mexp e) >>= store x
cmd (Seq c1 c2)    =  cmd c1 >> cmd c2
cmd (If b c1 c2)   =  mbexp b >>= λ v. if v then cmd c1 else cmd c2
cmd (While b c)    =  mwhile (mbexp b) (cmd c)
  where
    mwhile β φ = do  v ← β
                     if v then φ >> (mwhile β φ) else return ()
```

Figure 3. Semantics for the Language of Threads: Programs & Commands (Part 1)

The programming language for threads is similar to familiar examples from the literature of concurrency such as CSP (Hoare, 1978), Dijkstra's guarded command language (Dijkstra, 1975), and the language of temporal logic (Manna and 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 "‖") 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 eight 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* PID_q and the response to this request will be of the form (PID_r *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

$$
\begin{array}{lcl}
cmd\ (Print\ m\ e) & = & (mexp\ e)\ \texttt{>>=}\ \lambda\ v.\ signalI\ (Prnt_q\ (output\ m\ v)) \\
\quad \textbf{where} & & \\
\quad\quad output\ m\ v & = & m +\!\!+ \text{“ : ”} +\!\!+ show\ v \\
cmd\ Sleep & = & signalI\ Sleep_q \\
cmd\ (Fork\ c) & = & signalI\ (Fork_q\ c) \\
cmd\ (Broadcast\ x) & = & mexp\ (Var\ x)\ \texttt{>>=}\ (signalI\ \circ\ Bcst_q) \\
cmd\ (Receive\ x) & = & signal\ Rcv_q\ \texttt{>>=}\ \lambda\ (Rcv_r\ m).\ (store\ x\ m) \\
cmd\ Psem & = & signalI\ P_q \\
cmd\ Vsem & = & signalI\ V_q \\
cmd\ (Kill\ e) & = & (mexp\ e)\ \texttt{>>=}\ \lambda\ pid.\ signalI\ (Kill_q\ pid)
\end{array}
$$

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

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 "loop unrolling":

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

Note that this "loop unrolling" describes an infinite computation (in the sense of Section 4) 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:

$$
\begin{array}{lcl}
cmd\ (Receive\ x) & = & signal\ Rcv_q\ \texttt{>>=}\ \lambda\ (Rcv_r\ m).\ (store\ x\ m) \\
cmd\ Psem & = & signalI\ P_q
\end{array}
$$

To *Receive* a message into the program variable x, first the command *signal*s a receive request, expecting a response of the form $(Rcv_r\ m)$ where m is the message contents. Having received the message, it is then *store*d in x. To acquire the semaphore, a thread sends the signal P_q. 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 deliberately 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 list of threads ready to execute, a message buffer, a single semaphore, an output channel, and a counter for generating new process identifiers. The system configuration is captured as the following Haskell type declaration:

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

The first component is the ready 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 blocked processes waiting on the semaphore. The fourth component is an output channel (merely a *String*) and the fifth is a counter for generating process identifiers.

In general, the types of a scheduler and service handler will be:

$$
\begin{array}{l}
\mathit{sched} \quad :: \mathit{System} \to R\ () \\
\mathit{handler} :: \mathit{System} \to (\mathit{PID}, \mathit{Re}\ ()) \to R\ ()
\end{array}
$$

A *sched* morphism takes the system configuration (which includes the ready 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 ready list—into a single, interwoven scheduling typed in the basic R monad. The range in the typings of *sched* and *handler* is $(R\ ())$ precisely because the requested thread interactions have been mediated by *handler* along the lines illustrated in Figure 5.

Figure 5 illustrates a high-level overview of the structure and operation of an arbitrary resumption-monadic kernel and, in particular,

$$sched \begin{Bmatrix} (req_1,t_1) \\ \vdots \\ (req_i,t_i) \\ \vdots \\ (req_n,t_n) \end{Bmatrix} = handler\ (req_i,t_i)\ ;\ sched \begin{Bmatrix} (req_1,t_1) \\ \vdots \\ (req_i',rest(t_i)) \\ \vdots \\ (req_n,t_n) \end{Bmatrix}$$

Figure 5. Intuitive Structure and Operation of a Resumption-monadic Kernel. Following to its scheduling policy, scheduler *sched* picks thread t_i to execute from a collection of ready threads. The request handler, *handler*, services the selected thread's request, req_i and executes a slice of t_i. The handler returns control to the scheduler, throwing a new system configuration including the remainder of t_i, $(req_i', rest(t_i))$, back to *sched*.

the interaction between a scheduler, *sched*, and the service handler, *handler*. From the ready 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 to reflect 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: multiple calls to the co-recursive *sched* function will weave together a single basic schedule computation from slices of the *handler*-serviced reactive thread computations in the ready list.

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

$$\begin{array}{ll} rr :: System \rightarrow R\,() & \\ rr\ ([], \_, \_, \_, \_) & = Done\ () \quad \text{— stop when no threads} \\ rr\ ((t : ts), q, s, o, g) & = handler\ (ts, q, s, o, g)\ t \end{array}$$

A typical call to *handler* has the form $(handler\ sys\ (pid, P(req, r)))$ 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

We make use of two auxiliary functions throughout. The *tick* function inserts a "no-op" step onto the beginning of its argument. The *ins* function inserts a thread into the ready list component of a system configuration.

$$
\begin{array}{lll}
\mathit{tick} & :: & \mathit{Monad}\ m \Rightarrow \mathit{ResT}\ m\ a \rightarrow \mathit{ResT}\ m\ a \\
\mathit{tick} & = & \mathit{Pause} \circ \mathit{return}_m \\
\mathit{ins} & :: & \mathit{System} \rightarrow \mathit{PID} \rightarrow \mathit{Re}\ () \rightarrow \mathit{System} \\
\mathit{ins}\ (w, q, s, o, g)\ i\ r & = & (w +\!\!+ [(i, r)], q, s, o, g)
\end{array}
$$

When *handler* encounters a thread which has completed (i.e., the thread is a computation of the form $D\_$), it simply calls the scheduler with the system configuration unchanged. If the thread wishes to continue (i.e., it is of the form $P(\mathit{Cont}, r)$), then *handler* acknowledges the request by passing Ack to r:

$$
\begin{array}{lll}
\mathit{handler} & :: & \mathit{System} \rightarrow (\mathit{PID}, \mathit{Re}\ ()) \rightarrow R\ () \\
\mathit{handler}\ \mathit{sys}\ (i, D\ v) & = & \mathit{tick}\ (\mathit{rr}\ \mathit{sys}) \\
\mathit{handler}\ \mathit{sys}\ (i, P(\mathit{Cont}, r)) & = & \mathit{step}\ (r\ \mathit{Ack})\ \texttt{>>=}\ (\mathit{rr} \circ (\mathit{ins}\ \mathit{sys}\ i))
\end{array}
$$

As a result, the first atom in r is scheduled, and the rest the thread is passed to the scheduler.

6.2. Printing

When a print request ($\mathit{Prnt}_q\ \mathit{msg}$) is signaled, then the string *msg* is appended to the output channel o and the rest of the thread is passed to the scheduler.

$$
\begin{array}{l}
\mathit{handler}\ \mathit{sys}\ (i, P(\mathit{Prnt}_q\ \mathit{msg},\ r)) = \mathit{step}\ (r\ \mathit{Ack})\ \texttt{>>=}\ \mathit{nxt} \\
\quad \textbf{where} \\
\quad\quad (w, q, s, o, g) = \mathit{sys} \\
\quad\quad \mathit{nxt}\ r' \quad\quad = \mathit{rr}\ (w +\!\!+ [(i, r')], q, s, o +\!\!+ \mathit{msg}, g)
\end{array}
$$

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 o.

6.3. Dynamic Scheduling

A thread may request suspension with the Sleep_q signal; the handler acknowledges the Sleep_q request and reschedules the thread. The effect of this is to delay the thread by one scheduling cycle.

$$
\mathit{handler}\ \mathit{sys}\ (i, P(\mathit{Sleep}_q, r)) = \mathit{step}\ (r\ \mathit{Ack})\ \texttt{>>=}\ (\mathit{rr} \circ (\mathit{ins}\ \mathit{sys}\ i))
$$

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

A request to spawn a thread executing command c is handled by the clause below. A thread for c is created by applying *cmd* and the

result is given a new identifier g. Then, both parent and child thread are added back to the ready list:

$$
\begin{array}{l}
\mathit{handler}\ \mathit{sys}\ (i, P(\mathit{Fork}_q\ c, r)) = \ \mathit{step}\ (r\ \mathit{Ack})\ \texttt{>>=}\ \mathit{nxt} \\
\quad \textbf{where} \\
\qquad (w, q, s, o, g) = \mathit{sys} \\
\qquad \mathit{nxt}\ r' \qquad\quad = \mathit{rr}\ (w\ +\!\!+\ [(i, r'),\ \mathit{child}], q, s, o, g + 1) \\
\qquad \mathit{child} \qquad\quad = (g, \mathit{cmd}\ c)
\end{array}
$$

6.4. Asynchronous Message Passing

When a thread broadcasts a message m, it is appended to the message queue:

$$
\begin{array}{l}
\mathit{handler}\ \mathit{sys}\ (i, P(\mathit{Bcst}_q\ m, r)) = \ \mathit{step}\ (r\ \mathit{Ack})\ \texttt{>>=}\ \mathit{nxt} \\
\quad \textbf{where} \\
\qquad (w, q, s, o, g) = \mathit{sys} \\
\qquad \mathit{nxt}\ r' \qquad\quad = \mathit{rr}\ (w +\!\!+ [(i, r')], q +\!\!+ [m], s, o, g)
\end{array}
$$

When a Rcv_q signal occurs and the message queue is empty, then the thread waits:

$$
\begin{array}{l}
\mathit{handler}\ (w, [], s, o, g)\ (i, P(\mathit{Rcv}_q,\ r)) = \ \mathit{tick}\ \mathit{wait} \\
\quad \textbf{where} \\
\qquad \mathit{wait} = \mathit{rr}\ (w\ +\!\!+\ [(i, P(\mathit{Rcv}_q,\ r))], [], s, o, g)
\end{array}
$$

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:

$$
\begin{array}{l}
\mathit{handler}\ \mathit{sys}\ (i, P(\mathit{Rcv}_q,\ r)) = \ \mathit{step}\ (r\ (\mathit{Rcv}_r\ m))\ \texttt{>>=}\ \mathit{nxt} \\
\quad \textbf{where} \\
\qquad (w, q, s, o, g) = \mathit{sys} \\
\qquad \mathit{nxt}\ r' \qquad\quad = \mathit{rr}\ (w\ +\!\!+\ [(i, r')], ms, s, o, g) \\
\qquad m \qquad\quad = \mathit{head}\ q \\
\qquad \mathit{ms} \qquad\quad = \mathit{tail}\ q
\end{array}
$$

6.5. Process-level Preemption

Processes in a cooperative multitasking system only return control to the system voluntarily. In a preemptive multitasking system (e.g., the kernel described in this section), the system may interrupt or preempt

a running process to recover control. Control over processes can also occur at the process level as well and the current subsection describes one such operation.

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 $Kill_q\ j$, removes the thread with process identifier j from the ready list:

$$
\begin{array}{l}
\mathit{handler\ sys}\ (i, P(\mathit{Kill}_q\ j,\ r)) = \ \mathit{step}\ (r\ \mathit{Ack})\ \texttt{>>=}\ \mathit{nxt} \\
\quad \textbf{where} \\
\quad\quad (w, q, s, o, g) = \mathit{sys} \\
\quad\quad \mathit{nxt}\ r' \qquad\quad = \textbf{let} \\
\quad\quad\quad\quad\quad\quad \mathit{exit}\ i = \lambda(\mathit{pid}, t).\ \mathit{pid} \neq i \\
\quad\quad\quad\quad\quad\quad wl' \quad\ = \mathit{filter}\ (\mathit{exit}\ j)\ (w +\!\!+ [(i, r')]) \\
\quad\quad\quad\quad\quad\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \mathit{rr}\ (wl', q, s, o, g)
\end{array}
$$

Thread j is removed from the ready list using the Haskell standard prelude function:

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

In a call ($\mathit{filter}\ b\ l$), *filter* returns those elements of list l on which b is true (in order of their occurrence in l).

The $Kill_q$ 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 ($\mathit{throwTo}\ p\ e$) 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 an approach to a resumption-monadic model for Haskell's asynchronous exceptions.

6.6. Synchronization Primitives

Requesting the system semaphore s will succeed if $s > 0$, in which case the requesting thread will continue with the semaphore decremented; if $s \not> 0$, the requesting thread will suspend. These possible outcomes

are bound to *goahead* and *tryagain* in the following *handler* clause, and *handler* chooses between them based on the current value of s (see below). This implementation uses busy waiting for the sake of simplicity. One could easily implement more efficient strategies by including a queue of waiting threads with the semaphore.

$$
\begin{array}{l}
\mathit{handler}\ \mathit{sys}\ (i, P(P_q,\ r)) = \ \textbf{if}\ s\ >\ 0\ \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{goahead} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{tryagain} \\
\quad \textbf{where} \\
\qquad (w, q, s, o, g) = \mathit{sys} \\
\qquad \mathit{goahead} \qquad = \mathit{step}\ (r\ \mathit{Ack})\ \texttt{>>=}\ \mathit{go} \\
\qquad \mathit{go}\ r' \qquad\quad = \mathit{rr}\ (w +\!\!+ [(i, r')], q, s - 1, o, g) \\
\qquad \mathit{tryagain} \qquad = \mathit{tick}\ (\mathit{rr}\ (w +\!\!+ [(i, P(P_q, r))], q, s, o, g))
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{handler}\ (w, q, s, o, g)\ (i, P(V_q,\ r)) = \ \mathit{step}\ (r\ \mathit{Ack})\ \texttt{>>=}\ \mathit{nxt} \\
\quad \textbf{where} \\
\qquad \mathit{nxt}\ r' = \mathit{rr}\ (w +\!\!+ [(i, r')], q, s + 1, o, g)
\end{array}
$$

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

7. Garbage Collection

This section presents another application of the resumption-monadic framework developed in the preceding sections. The example is that of adding a model for garbage collection to a definitional interpreter for a simple λ-calculus. It follows the familiar pattern in monadic specification set by Wadler in his now classic introduction to monads (Wadler, 1992). Wadler shows how new behaviors can be manifested in an interpreter via changes to the monad underlying the interpreter. Here, a λ-calculus interpreter written in terms of the monad of environments is reinterpreted in a reactive resumption monad. This reinterpretation, in turn, exposes the "breaks" in program evaluation when a memory manager might decide to perform garbage collection. This interaction between a program and the garbage collector illustrates a form of implicit concurrency standing in contrast to the explicit concurrency of Section 6.

We begin with an interpreter for a language with abstraction, application, variables, and arithmetic. From there, we refactor the design by transferring responsibility for managing the environment to a heap-based memory management discipline. The memory management system is implemented as a signal handler in the sense of Section 6. It is then a simple matter to extend the memory manager with garbage collection.

For the sake of simplicity, we use certain algebraic data types (such as lists) that, by their very nature, require the host language itself to implement garbage collection. This means that the garbage collector presented here should be viewed as an abstract model or specification, rather than a useful implementation. If we run the garbage collecting interpreter presented below in Haskell, the result is doubly garbage collected—once by the explicit garbage collector implemented in the resumption monad, and once by the implicit garbage collection of the Haskell runtime. Previous work has demonstrated, however, that specifications of this sort can be compiled directly to efficient machine code without incurring the double-garbage collection problem (Harrison et al., 2009).

7.1. The Standard Interpreter

First, the monad underlying the λ-calculus interpreter is defined, then the definition of the domain of values is given, followed finally by the text of the standard interpreter. The monad underlying the standard interpreter is an environment monad (Liang et al., 1995):

$$\begin{aligned}
&\textbf{data}\ E\ a &&= E\ (Env \to a)\\
&\textbf{type}\ Env &&= [(Name, V)]
\end{aligned}$$

$$\begin{aligned}
&rdenv :: E\ Env\\
&rdenv &&= E\ (\lambda\rho.\ \rho)\\
&inenv :: Env \to E\ a \to E\ a\\
&inenv\ \rho\ (E\varphi) &&= E\ (\lambda d.\varphi\rho)
\end{aligned}$$

This monad has two attendant non-proper morphisms, *rdenv* and *inenv*. The first, *rdenv*, is an E computation that returns the current environment, while a call to the second, $inenv\ \rho\ \varphi$, resets the current environment within φ to ρ. The bind and unit for E are defined in Liang et al. (1995, 1996, 1998).

The data type V, defined below, is the type of values in the interpreter. The standard interpreter is given in terms of the functions *mkfun*, *appenv*, and *apply*:

$$\textbf{data}\ V \qquad = Wrong \mid Num\ Int \mid Fun\ (V \to E\ V)$$

```
mkfun :: Name → E V → E V
mkfun  n  φ      =  rdenv >>= λρ.
                    return (Fun (λv. inenv ρ[n↦v] φ))
appenv :: Name → E V
appenv n         = rdenv >>= lookup n

apply :: V → V → EV
apply (Fun f) v = f v
```

The source language *Term* and interpreter $[\![-]\!]$ are defined as follows:

```
data Term  =  Var  Name              — Variables
           |  Con  Int               — Constants
           |  Add  Term  Term        — Addition
           |  Lam  Name  Term        — Abstraction
           |  App  Term  Term        — Application
```

```
[[−]] :: Term → E V
[[Var x]]    = appenv x
[[Con i]]    = return (N i)
[[Add u v]]  = [[u]] >>= λa. [[v]] >>= λb. add a b
[[Lam x v]]  = mkfun x [[v]]
[[App t u]]  = [[t]] >>= λf. [[u]] >>= λa. apply f a
```

7.2. Reinterpreting the Standard Interpreter in *Re*

We now reinterpret the interpreter with a reactive resumption monad where operations in environments in E are replaced by signals and a signal handler is given to service these requests. The requests and responses data types specifying the signals are:

```
data Req  =  MkCl Name (Re V)      — Make closure
          |  Ap V V                — Apply
          |  LkUp  Name            — Look up name
          |  Cont                  — Continue
data Rsp  =  Val  V                — Return value
          |  Ack                   — Acknowledge
```

And, the new monad hierarchy is:

```
type R  = ResT Id
type Re = ReactT Req Rsp Id
```

All operations that deal with environments (*mkfun*, *apply*, and *appenv*) are now implemented as signals to the handler. The helper functions are then redefined in terms of the reactive operator, *signal*:

$$
\begin{aligned}
&\mathit{mkfun} :: \mathit{Name} \to \mathit{Re}\ V \to \mathit{Re}\ V \\
&\mathit{mkfun}\ n\ \varphi = \mathit{signal}_V\ (\mathit{MkCl}\ n\ \varphi) \\
&\mathit{apply} :: V \to V \to \mathit{Re}\ V \\
&\mathit{apply}\ f\ v\ \ = \mathit{signal}_V\ (\mathit{Ap}\ f\ v) \\
&\mathit{appenv} :: \mathit{Name} \to \mathit{Re}\ V \\
&\mathit{appenv}\ n\ \ = \mathit{signal}_V\ (\mathit{LkUp}\ n)
\end{aligned}
$$

We are using a slightly altered form of the *signal* operator here; it is to be used when the response received from the handler is known to be a value constructed with *Val*:

$$
\begin{aligned}
&\mathit{signal}_V\ \ :: \mathit{Req} \to \mathit{Re}\ V \\
&\mathit{signal}_V\ q = P\ (q,\ \mathit{return}_K \circ \mathit{return}_{Re} \circ (\lambda\ (\mathit{Val}\ v).\ v))
\end{aligned}
$$

The function values in V are replaced with closures, because the memory manager presented later must inspect closures to implement garbage collection:

$$\textbf{data}\ V = \mathit{Wrong} \mid \mathit{Num}\ \mathit{Int} \mid \mathit{Cl}\ \mathit{Name}\ \mathit{Env}\ (\mathit{Re}\ V)$$

The code for the new handler is:

$$
\begin{aligned}
&\mathit{handler} :: \mathit{Env} \to \mathit{Re}\ V \to R(\mathit{Re}\ V) \\
&\mathit{handler}\ \rho\ (D\ v) &&= \mathit{return}\ (D\ v) \\
&\mathit{handler}\ \rho\ (P(\mathit{Cont}, r)) &&= \mathit{step}_R\ (r\ \mathit{Ack}) \\
&\mathit{handler}\ \rho\ (P(\mathit{MkCl}\ n\ \varphi, r)) &&= \mathit{step}_R\ (r\ (\mathit{Val}\ (\mathit{Cl}\ n\ \rho\ \varphi))) \\
&\mathit{handler}\ \rho\ (P(\mathit{Ap}(\mathit{Cl}\ n\ \rho'\ \varphi)\ v, r)) \\
&\qquad\qquad &&= \mathit{go}\ (\rho'[n \mapsto v])\ \varphi\ \texttt{>>=}\ \mathit{step}_R \circ r \circ \mathit{Val} \\
&\mathit{handler}\ \rho\ (P(\mathit{Ap}\ \_\ v, r)) &&= \mathit{step}_R\ (r\ (\mathit{Val}\ \mathit{Wrong})) \\
&\mathit{handler}\ \rho\ (P(\mathit{LkUp}\ n, r)) &&= \mathit{step}_R\ (r\ (\mathit{Val}\ (\mathit{lookup}\ n\ \rho)))
\end{aligned}
$$

The helper function *loop* iterates a handler h over its second argument. There is a clear analogy between resumption computations and list or streams; *loop* may be viewed as a resumption-monadic analogue of the *map* function on lists.

$$
\begin{aligned}
&\mathit{loop} :: (\mathit{Re}\ V \to R(\mathit{Re}\ V)) \to \mathit{Re}\ V \to R\ V \\
&\mathit{loop}\ h\ (D\ v) = \mathit{Done}\ v \\
&\mathit{loop}\ h\ \varphi\ \ \ \ = (h\ \varphi)\ \texttt{>>=}\ \mathit{loop}\ h
\end{aligned}
$$

Finally, the function *go* is the top-level function which supplies the initial environment to the handler.

$$
\begin{aligned}
&\mathit{go} :: \mathit{Env} \to \mathit{Re}\ V \to R\ V \\
&\mathit{go} = \mathit{loop} \circ \mathit{handler}
\end{aligned}
$$

Placing environments under the control of the handler paves the way for garbage-collected memory management. Notice that, while the *text* of the interpreter function $[\![-]\!]$ is unchanged, the notion of computation underlying it is radically different from the environment-monad interpreter described above.

7.3. Memory Management

We now extend the handler with a simple memory manager. Instead of storing values for live variables directly in the environment as in the previous interpreter, we store them in a map from locations to values; in other words, variables are now heap allocated. We extend the monad hierarchy:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Env} = [(\mathit{Name}, \mathit{Loc})] \\
&\textbf{type}\ \mathit{Sto}\ = [(\mathit{Loc}, V)] \\
&\textbf{type}\ K\ \ = \mathit{StateT}\ \mathit{Sto}\ \mathit{Id} \\
&\textbf{type}\ R\ \ = \mathit{ResT}\ K \\
&\textbf{type}\ \mathit{Re}\ = \mathit{ReactT}\ \mathit{Req}\ \mathit{Rsp}\ K
\end{aligned}
$$

The monad K, taken together with the non-proper morphisms *read* and *store* defined below, models the memory of a Von Neumann machine. One can envision compiling *read* and *store* operations almost directly to machine code on any standard instruction set architecture, and more will be said about this in Section 8. The operators *read* and *store* are implemented in Haskell as follows:

$$
\begin{aligned}
&\mathit{read} :: \mathit{Loc} \to KV \\
&\mathit{read}\ l\ \ \ = g\ \texttt{>>=}\ \lambda s.\ \mathit{return}\ (\mathit{lookup}\ l\ s) \\
&\mathit{store} :: V \to \mathit{Loc} \to K() \\
&\mathit{store}\ v\ l = u[l \mapsto v]
\end{aligned}
$$

Here, g and u are the "get" and "update" state monad operations.

Two changes to *handler* complete the new implementation. When the handler receives an *Ap* request, it stores the value at a free heap location using *storeNew*, and binds a pointer to that location to n in the closure's environment. *LkUp* requests are handled by retrieving the pointer from the environment, and following that pointer via *read* to retrieve the value from the heap.

$$
\begin{aligned}
&\mathit{handler} :: \mathit{Env} \to \mathit{Re}\ V \to R(\mathit{Re}\ V) \\
&\mathit{handler}\ \rho\ (P(\mathit{Ap}(\mathit{Cl}\ n\ \rho'\ \varphi)\ v, r))
\end{aligned}
$$

$$
\begin{aligned}
&= step_R(storeNew\ v)\ \texttt{>>=}\ \lambda l.\\
&\quad go\ (\rho'[n{\mapsto}l])\ \varphi\ \texttt{>>=}\ step_R \circ r \circ Val\\
handler\ \rho\ (P(LkUp\ n, r)) &= step_R\ (read\ (lookup\ n\ \rho))\\
&\quad \texttt{>>=}\ step_R \circ r \circ Val
\end{aligned}
$$

The function *storeNew* is defined:

$$
\begin{aligned}
&storeNew :: V \to Re\ Loc\\
&storeNew\ v = findFreeLoc\ \texttt{>>=}\ \lambda l.\ store\ v\ l\ \texttt{>>}\ return\ l
\end{aligned}
$$

Here, the function *findFreeLoc* inspects the heap to find an unallocated memory location; its straightforward definition is not included here.

7.4. Implementing Garbage Collection

The interpreter in Section 7.3 is equivalent to that of Section 7.2 in the sense that they ultimately produce the same answers, but it does not yet recover dead heap locations. This section presents an extension of the handler with garbage collection.

When the new handler, defined below, needs to store a new value on the heap, it first invokes the non-proper morphism *gc*, which implements a garbage collection strategy such as mark-and-sweep. We must therefore supply *gc* not just with the current environment, but also with all environments that are on the stack. This change is reflected in the new type of *handler*, which now takes a stack argument instead of an environment. By convention, the current environment is always at the top of the stack. With this modification, *gc* may inspect all bound variables at a point in program evaluation to distinguish live locations from dead ones. The modified *handler* is given below; the only change from the previous version is the addition of "$step_R\ (gc\ (\rho'\ :\ s))$ >>":

$$
\begin{aligned}
&\textbf{type}\ Stack\ =\ [Env]\\
&handler :: Stack \to ReV \to R(Re\ V)\\
&handler\ s\ (P(Ap\ (Cl\ n\ \rho'\ \varphi)\ v, r)) = \quad step_R\ (gc\ (\rho'\ :\ s))\ \texttt{>>}\\
&\qquad\qquad\qquad\qquad\qquad\quad step_R\ (storeNew\ v)\ \texttt{>>=}\ \lambda l.\\
&\qquad\qquad\qquad\qquad\qquad\quad go\ (\rho'[n{\mapsto}l]\ :\ s)\ \varphi\\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{>>=}\ step_R\ \circ\ r\ \circ\ Val
\end{aligned}
$$

Finally, we define the garbage collection function itself. The definition uses two helper functions whose definitions are not elaborated here: $findLive :: Stack \to [Loc]$ takes a stack of environments and returns a list of locations accessible from environments on the stack (essentially the "mark" in mark-and-sweep garbage collection), and *freeDead* is a helper function that takes a list of live locations and updates the store by marking as free all allocated locations not present in the live list. The

constant *gcThreshold* :: *Int* defines the minimum heap size required for a garbage collection pass to be triggered.

```
gc :: Stack → K ()
gc stk = u (gcHelper stk)
  where
    gcHelper :: Stack → Sto → Sto
    gcHelper sta sto = if length sto > gcThreshold then
                         freeDead (findLive sta) sto
                       else
                         sto
```

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 and Hook, 2009) and bioinformatics (Harrison and 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 and 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:

$$\textbf{data}\ \mathit{Req} = \mathit{Cont} \mid \mathit{Divide} \mid \mathit{Die} \mid \mathit{Sleep} \mid \mathit{Grow} \mid \mathit{LightConcentration}$$
$$\textbf{data}\ \mathit{Rsp} = \mathit{Ack} \mid \mathit{LightConcRsp}\ \mathit{Float}$$

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.

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 is 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. 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 (Harrison et al., 2008). In fact, Swierstra and Altenkirch have recently proposed an approach to the semantics of the Awkward Squad (Swierstra and Altenkirch, 2007) that models *IO* with reactive resumption monads (although the reactive resumption-monadic structure of these *IO* models is not explicitly identified as such in their paper).

An alternative formulation of the reactive resumption monad that differs in the treatment of responses and requests is explored by Swierstra and Altenkirch (Swierstra and Altenkirch, 2007). While the pre-

sentation in Section 4.2 specifies requests and responses so that, in principle, any response could be given to any request, Swierstra and Altenkirch's formulation supports a per-request notion of response. For example, the following is the syntax of the "teletype" monad TT:

$$\begin{aligned}\textbf{data}\ TT\ a = &\ GetChar\ (Char\ \rightarrow\ TT\ a)\\ \mid &\ PutChar\ Char\ (TT\ a)\\ \mid &\ Return\ a\end{aligned}$$

In our setting, separate requests are collected in the *Req* sum type and the system calls are handled by pattern matches (e.g., $P(PutChar\ c, k)$). Swierstra and Altenkirch's formulation, in effect, merges the pause "P" with the request, "$PutChar\ c$", in the definition of TT.

Monadic specifications can typically be rendered executable by embedding them in a higher-order functional programming language like Haskell, but alternative approaches to implementing monadic specifications have been explored in the past. Filinski (1999) introduced a general approach to translating layered monads into a λ-calculus with first-class continuations. One motivation for finding implementation alternatives is to avoid some of the overhead that can come with functional languages. Take, for example, the garbage collection building block presented in Section 7. As a means for relating a language semantics to an implementation, this specification has its virtues. But, a Haskell implementation of it will not be of much practical use because it will be doubly garbage collected—i.e., it has the explicit garbage collector from the building block as well as the Haskell run-time system's collector. Fortunately, kernels written in the style of Sections 6 and 7 are tail recursive and tail recursive functions can always be transformed into loop code via tail call elimination (Muchnick, 1997).

The High Assurance Security Kernel Laboratory[9] at the University of Missouri is building compilers to translate security kernels (Harrison and Hook, 2009) written within the resumption-monadic framework described here directly into machine language. These kernels are built from particular combinations of monad transformers (especially, resumption monad transformers for concurrency). This monad compilation strategy is not intended to be as general as Filinski's. Rather, they will take advantage of the restriction to concurrency monads to produce verifiable object code with predictable space and time behavior. For more information about monad compilation strategies, please consult the references (Harrison et al., 2009).

Layered resumption monads can play an essential rôle in the formal development of concurrent applications. While we have emphasized

[9] HASK home page: `http://hask.cs.missouri.edu`.

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 the verification of a security kernel built with the monadic structures explicated here (Harrison and Hook, 2009), 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 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 and Pnueli, 1991) for concurrent programming; that is, a specification language combined with a repertoire of related proof techniques.

References

Alexander, D., W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunder, S. Nettles, and J. Smith: 1998, 'The SwitchWare active network architecture'. *IEEE Network*.

Andrews, G. R. and R. A. Olsson: 1993, *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings.

Armstrong, J., R. Virding, C. Wikström, and M. Williams: 1996, *Concurrent Programming in Erlang*. Prentice-Hall, second edition.

Bakker, J. d.: 1980, *Mathematical Theory of Program Correctness*, International Series in Computer Science. Prentice-Hall.

Bakker, J. d. and E. d. Vink: 1996, *Control Flow Semantics*, Foundations of Computing Series. The MIT Press.

Barr, M. and C. Wells: 1990, *Category Theory for Computing Science*. Prentice Hall.

Biagioni, E., R. Harper, and P. Lee: 2001, 'A Network Protocol Stack in Standard ML'. *Higher-Order and Symbolic Computation* **14**(4), 309–356.

Bird, R. and P. Wadler: 1988, *Introduction to Functional Programming*. Prentice Hall.

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

Carter, D.: 1994, 'Deterministic Concurrency'. Ph.D. thesis, Department of Computer Science, University of Bristol.

Claessen, K.: 1999, 'A Poor Man's Concurrency Monad'. *Journal of Functional Programming* **9**(3), 313–323.

Cooper, E. C. and J. G. Morrisett: 1990, 'Adding Threads to Standard ML'. Technical Report CMU-CS-90-186, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Cupitt, J.: 1992, '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, E. W.: 1975, 'Guarded commands, nondeterminacy and formal derivation of programs'. *Communications of the ACM* **18**(8), 453–457.

Emerson, E. A.: 1990, 'Temporal and Modal Logics'. In: J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science*, Vol. B. Elsevier Science Publishers B.V., Chapt. 16, pp. 995–1072.

Espinosa, D.: 1995, 'Semantic Lego'. Ph.D. thesis, Columbia University.

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

Filinski, A.: 1996, 'Controlling Effects'. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

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

Flanagan, C. and S. Qadeer: 2003, 'Types for atomicity'. In: *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI '03)*. pp. 1–12, ACM Press.

Flatt, M., R. B. Findler, S. Krishnamurthi, and M. Felleisen: 1999, 'Programming Languages as Operating Systems (or Revenge of the Son of the Lisp Machine)'. In: *Proceedings of the 4th ACM International Conference on Functional Programming (ICFP)*. pp. 138–147, ACM Press.

Ganz, S. E., D. P. Friedman, and M. Wand: 1999, 'Trampolined style'. In: *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*. pp. 18–27, ACM Press.

Gibbons, J. and G. Hutton: 2005, 'Proof Methods for Corecursive Programs'. *Fundamenta Informaticae Special Issue on Program Transformation* **66**(4), 353–366.

Harper, R., P. Lee, and F. Pfenning: 1998, 'The Fox Project: Advanced Language Technology for Extensible Systems'. Technical Report CMU-CS-98-107, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. (Also published as Fox Memorandum CMU-CS-FOX-98-02).

Harrison, W. L., G. Allwein, A. Gill, and A. Procter: 2008, 'Asynchronous Exceptions as an Effect'. In: *Proceedings of the 9th International Conference on the Mathematics of Program Construction (MPC08)*, Vol. 5133 of *LNCS*. pp. 153–176.

Harrison, W. L. and R. W. Harrison: 2004, 'Domain Specific Languages for Cellular Interactions'. In: *Proceedings of the 26th Annual IEEE International Conference on Engineering in Medicine and Biology*.

Harrison, W. L. and J. Hook: 2009, 'Achieving Information Flow Security Through Monadic Control of Effects.'. *Journal of Computer Security* **17**(5), 599–653.

Harrison, W. L., A. Procter, J. Agron, G. Kimmel, and G. Allwein: 2009, 'Model-Driven Engineering from Modular Monadic Semantics: Implementation Techniques Targeting Hardware and Software'. In: *DSL '09: Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*. Berlin, Heidelberg, pp. 20–44, Springer-Verlag.

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

Hoare, C. A. R.: 1978, 'Communicating Sequential Processes'. *Communications of the ACM* **21**(8), 666–677. See corrigendum (**?**).

Jacobs, B. and E. Poll: 2003, 'Coalgebras and Monads in the Semantics of Java'. *Theoretical Computer Science* **291**(3), 329–349.

Kiselyov, O. and C. Shan: 2007, 'Delimited Continuations in Operating Systems'. In: B. N. Kokinov, D. C. Richardson, T. Roth-Berghofer, and L. Vieu (eds.): *CONTEXT*, Vol. 4635 of *Lecture Notes in Computer Science*. pp. 291–302, Springer.

Krstic, S., J. Launchbury, and D. Pavlovic: 2001, 'Categories of Processes Enriched in Final Coalgebras'. In: *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*. pp. 303–317, Springer-Verlag.

Launchbury, J. and S. L. Peyton Jones: 1994, 'Lazy functional state threads'. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. pp. 24–35, ACM Press.

Li, P. and S. Zdancewic: 2007, 'Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives'. In: *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA, pp. 189–199, ACM Press.

Liang, S.: 1998, 'Modular Monadic Semantics and Compilation'. Ph.D. thesis, Yale University.

Liang, S., P. Hudak, and M. Jones: 1995, 'Monad transformers and modular interpreters'. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. pp. 333–343, ACM Press.

Lin, A. C.: 1998, 'Implementing Concurrency for an ML-based Operating System'. Ph.D. thesis, Massachusetts Institute of Technology.

Loeckx, J., H.-D. Ehrich, and M. Wolf: 1996, *Specification of Abstract Data Types*. New York, USA: Wiley & Teubner.

Manna, Z. and A. Pnueli: 1991, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag.

Marlow, S., S. L. P. Jones, A. Moran, and J. H. Reppy: 2001, 'Asynchronous Exceptions in Haskell'. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. pp. 274–285.

Moggi, E.: 1990, 'An Abstract View of Programming Languages'. Technical Report ECS-LFCS-90-113, Department of Computer Science, Edinburgh University.

Muchnick, S. S.: 1997, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Papaspyrou, N. S.: 1998, '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, N. S.: 2001, 'A Resumption Monad Transformer and its Applications in the Semantics of Concurrency'. In: *Proceedings of the 3rd Panhellenic Logic Symposium*. An expanded version is available as a technical report from the author by request.

Papaspyrou, N. S. and D. Maćoš: 2000, 'A Study of Evaluation Order Semantics in Expressions with Side Effects'. *Journal of Functional Programming* **10**(3), 227–244.

Peyton Jones, S.: 2000, 'Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell'. In: *Engineering Theories of Software Construction*, Vol. III 180 of *NATO Science Series*. IOS Press, pp. 47–96.

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

Peyton Jones, S., A. Gordon, and S. Finne: 1996, 'Concurrent Haskell'. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. pp. 295–308, ACM Press.

Peyton Jones, S. and P. Wadler: 1993, 'Imperative Functional Programming'. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. pp. 71–84, ACM Press.

Plasmeijer, R. and M. van Eekelen: 1998, 'The Concurrent Clean Language Report'. Technical Report CSI-R9816, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands.

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

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

Reppy, J. H.: 1999, *Concurrent programming in ML*. New York, NY, USA: Cambridge University Press.

Roscoe, W. A.: 1998, *Theory and Practice of Concurrency*. Prentice-Hall.

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

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

Spiliopoulou, E.: 1999, 'Concurrent and Distributed Functional Systems'. Technical Report CS-EXT-1999-240, University of Bristol.

Stoye, W.: 1984, 'A New Scheme for Writing Functional Operating Systems'. Technical Report 56, Computing Laboratory, Cambridge University.

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

Swierstra, W. and T. Altenkirch: 2007, 'Beauty in the beast'. In: *Proceedings of the ACM SIGPLAN Haskell Workshop (Haskell07)*. pp. 25–36.

Tolmach, A. and S. Antoy: 2003, 'A Monadic Semantics for Core Curry'. In: *Proceedings of the 12th International Workshop on Functional and (Constraint) Logic Programming*.

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

Turner, D.: 1990, 'An Approach to Functional Operating Systems'. In: D. Turner (ed.): *Research Topics in Functional Programming*. Addison-Wesley Publishing Company, pp. 199–217.

van Weelden, A. and R. Plasmeijer: 2002, 'Towards a Strongly Typed Functional Operating System'. In: *Proceedings of the 14th International Workshop on the Implementation of Functional Languages (IFL)*, Vol. 2670 of *Lecture Notes in Computer Science*. Springer-Verlag.

Wadler, P.: 1992, 'The Essence of Functional Programming'. In: *Proceedings of the 19th Symposium on Principles of Programming Languages (POPL)*. pp. 1–14, ACM Press.

Wadler, P.: 1995, 'Monads for functional programming'. In: *Proceedings of the 1992 Marktoberdorf International Summer School on Logic of Computation*, Vol. 925 of *Lecture Notes in Computer Science*. pp. 24–52.

Wand, M.: 1980, 'Continuation-based multiprocessing'. In: *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*. pp. 19–28. Reprinted in Higher-Order and Symbolic Computation 12(3):285–299, 1999, with a foreword (**?**).

Appendix

A. The Branching Kernel

This appendix presents a generalization of the kernel from Section 6 to one with a *branching* notion of time (Manna and 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 and it is also the foundation for a recent model of asynchronous exceptions (Harrison et al., 2008). 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, $c \,||\, c'$, 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 A.1 describes how non-determinism is modeled semantically and how such models are represented in functional languages via the list monad. Section A.2 shows how the non-determinism effect is combined with the state and concurrency effects. Section A.3 describes the non-deterministic scheduler.

A.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; for example, the value of 1 *amb* 2 is either 1 or 2. The *amb* operator is *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. 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 "**let** $x = (1\ amb\ 2)$ **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:

$$return\ x = \{\ x\ \} \qquad S \gg\!= f = \bigcup(f\ S)$$

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'.

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:

```
instance Monad [] where
  return x       = [x]
  (x : xs) >>= f = f x ++ (xs >>= f)
  []       >>= f = []
```

This straightforward implementation suffices for our purposes, but it is known to contain an inaccuracy when the lists involved are infinite (Tolmach and 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:

$$merge_{[]} :: [[a]] \rightarrow [a]$$
$$merge_{[]} = concat$$

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

A.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:

$$\textbf{type } St_n = StateT\ Sto\ []$$
$$\textbf{type } R_n = ResT\ St_n$$
$$\textbf{type } Re_n = ReactT\ Req\ Rsp\ St_n$$

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:

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

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:

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

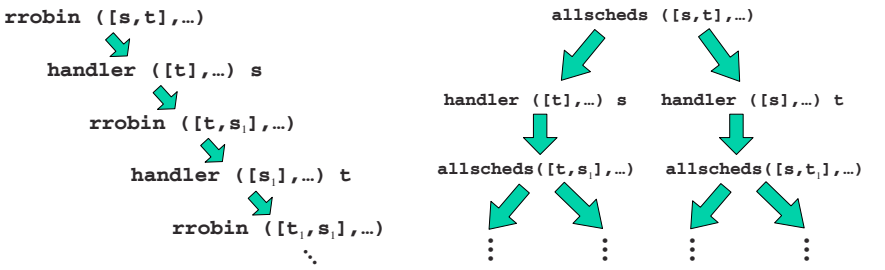

Figure 6. Round-robin Schedulings (left): Reflects the call graph between the *rr* scheduler and the *handler* routine. *Finite-branching Scheduler* (right): Reflects the call graph between the *allscheds* scheduler and the *handler* routine.

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

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

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.

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

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.

A.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 A.2. The second and final step defines a non-deterministic scheduler that picks every possible ready 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 all schedules in a tree-like fashion. For the sake of comparison, Figure 6 portrays (on the left) the actions of the deterministic scheduler, rr, and (on the right) the non-deterministic scheduler, *allscheds*.

The auxiliary function, *schedulings*, computes all possible scheduling choices from a ready list. For ready 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 $(t_i, [t_1, \ldots, t_{i-1}, t_{i+1}, t_n])$. Note that the ordering within the second component may not be identical to that of the input list. The Haskell code for *schedulings* is:

```
schedulings     :: [a] → [(a, [a])]
schedulings []  = []
schedulings wl = hts [] wl
  where
    hts front []       = []
    hts front (t : ts) = (t, front++ts) : (hts (t : front) ts)
```

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 6. The Haskell code for *allscheds* is:

```
allscheds :: System a → R_n a
allscheds (w, q, s, o, g) =merge_R (map dispatch scheds)
  where
    scheds   = schedulings wl
    dispatch = λ (t, ts). handler (ts, q, s, o, g) t
```

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