Asynchronous Exceptions As An Effect

William L. Harrison[1], Gerard Allwein[2], Andy Gill[3], and Adam Procter[1]

[1] Dept. of Computer Science, University of Missouri, Columbia, Missouri, USA.
[2] Naval Research Laboratory, Code 5543, Washington, DC 20375, U.S.A.
[3] Galois, Inc., Beaverton OR 97005, USA.

Abstract

Asynchronous interrupts abound in computing systems, yet they remain a thorny concept for both programming and verification practice. The ubiquity of interrupts underscores the importance of developing programming models to aid the development and verification of interrupt-driven programs. The research reported here recognizes asynchronous interrupts as a computational effect and encapsulates them as a building block in modular monadic semantics. The resulting integrated semantic model can serve as both a guide for functional programming with interrupts and as a formal basis for reasoning about interrupt-driven computation as well.

1 Introduction

The asynchronous interrupt, according to Dijkstra [4], was a great invention:

> *...but also a Box of Pandora. Because the exact moments of the interrupts were unpredictable and outside our control, the interrupt mechanism turned the computer into a nondeterministic machine with a non-reproducible behavior, and could we control such a beast?*

The construction and verification of programs in the presence of asynchronous exceptions is notoriously difficult. Certainly, one cause of this difficulty is that a setting with interrupts is necessarily concurrent, there being, at a minimum, distinct interrupting and interrupted entities. But, more fundamentally, the notion of computation for asynchronous interrupt is inherently non-deterministic and controlling the non-determinism "beast" remains a challenge.

Is there an integrated model that can serve as both a guide for programming with interrupts and a formal semantics for reasoning about them as well? In the past, language semanticists and functional programmers have both turned to monads and monadic semantics when confronted with non-functional effects (e.g., state, exceptions, etc.). This paper argues that one way to understand asynchronous interrupts is as a computational effect encapsulated by a monadic construction. This paper explores precisely this path by decomposing the asynchronous interrupt effect into monadic components. Surprisingly enough, although all of the monadic components applied here are well-known [18, 21], this work is apparently the first to put them together to model interrupts.

This paper argues that monadic semantics [18] (and, particularly, modular monadic semantics (MMS) [15]) is an appropriate foundation for an integrated model of interrupts. With the monadic approach, one may have one's cake and eat it, too: formal specifications in monadic style are easily rendered as executable functional programs. The contributions of this paper are:

- A semantic building block for asynchronous behaviors. This building block follows the same lines as other, well-known building blocks in MMS: by a straightforward extension of the underlying monad, we may define a set of operators that, in turn, may be used to define asynchronous interrupts. In the lingo of MMS, asynchronicity becomes a "building block" that may be added to other monadic specifications.
- A denotational framework for modeling a member of the infamous "Awkward Squad" [22]. These are behaviors considered difficult to accommodate within a pure, functional setting.
- Extend published formal models of OS kernels with asynchronous behavior. We apply the interrupts building block to monadic kernels [9], thereby extending the expressiveness of these models of concurrent systems.
- Supports functional style programming of interrupts. The rendering of this semantics in Haskell also allows deterministic replay of non-deterministic systems.

What do we mean by an asynchronous interrupt? It is a form of exception thrown by the environment external to a thread. A synchronous exception, by contrast, arises from an event internal to a thread's execution; for example, they can originate from run-time errors (e.g., division-by-zero or pattern match failure) and OS system calls. Asynchronous exceptions arise from events external to a thread and may occur at any point between the atomic actions that comprise the thread. The terms *interrupt* and *asynchronous exception* are used interchangeably throughout this article.

As we use it, the term *interrupt* should not be limited to the hardware mechanism by which a CPU communicates with the external world. Hardware interrupts are a special form of asynchronous exception; they can arise at any time and result in a temporary transfer of control to a interrupt service routine. Asynchronous exceptions do not, in general, involve such a temporary control transfer. The case study we present concerns a hardware interrupt mechanism, although the model we use applies equally well for asynchronous exceptions in general. In particular, the current model has been applied in another significant case study in the semantic specification of the typed interrupt calculus of Palsberg [2, 20]. This specification will be published in a sequel.

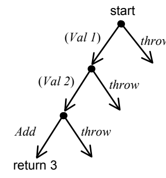
1.1 Summary of the MMAE

This section summarizes two applications of the monadic model of asynchronous exceptions (MMAE)—one in denotational semantics and the other in functional programming—and will provide the reader with helpful intuitions about the

MMAE. Asynchronous exceptions according to this model are really a composite effect, combining non-determinism, concurrency, and some notion of interactivity. Each of these effects corresponds to well-known monads, and the MMAE encapsulates them all within a single monad. Haskell renderings of all the examples in this paper are available online [8].

Non-determinism is inherent in asynchronous exceptions. The reason is simple: when (or if) a computation is interrupted is not determined by the computation itself and so exceptions are, therefore, by definition non-deterministic w.r.t. the computation. That a notion of concurrency is necessary is less obvious. Computation here is assumed to be "interruptible" only at certain identified points within its execution. These breakpoints (implicitly) divide the computation up into un-interruptible units or atoms. This is tantamount to a theory of concurrency. Finally, a notion of interactivity is required for obvious reasons: the "interruptee" and "interrupter" interact according to some regime.

In Section 4, we consider Hutton's language of arithmetic expressions with synchronous and asynchronous exceptions [12]. This language is defined in detail there, but, for the time being, consider the simple arithmetic expression, $(Add\,(Val\,1)\,(Val\,2))$. In Hutton's operational semantics, there are four possible evaluations of this expression, assuming that exceptions are "on". These evaluations may be described as a tree (see inset) in which each breakpoint "•" represents a place at which the arithmetic computation may be interrupted. The evaluation which returns 3 proceeds along the leftmost path. An asynchronous exception may occur at any of the breakpoints, corresponding to the three evaluations ending in *throw*. In the MMAE, the semantics of $(Add\,(Val\,1)\,(Val\,2))$ and its interaction with exceptions is represented in closed form as a single term in a monadic algebra of effects:



$$merge\,\{merge\,\{merge\,\{\eta\,3, throw\}, throw\}, throw\} \qquad (\dagger)$$

The intuitive meaning of this term is clear from it correspondence to the inset tree: the *merge* operator combines two "possible histories" together, rather like a branch constructor in a tree data type; the *throw* "leaf" is the result of an asynchronous exception; the monadic unit "$(\eta\,3)$" leaf computation returns the final value.

Figure 1 gives an example application of the MMAE. Recent research has demonstrated how kernels with a broad range of OS behaviors (e.g., message-passing, synchronization, forking, etc.) may be formulated in terms of resumption monads [9] and generalized with monad transformers. The application presented in Section 5 describes how such kernels may be extended to include asynchronous exception behaviors as well. This extension manifests itself in Figure 1 in the creation of the monads K and R underlying the kernel specification. For the kernel without the interrupt-driven port (Figure 1, middle), the monad transformers constructing K and R (i.e., StateT and ResT) are applied to the identity monad, Id; in the kernel with the interrupt-driven port (Figure 1, middle), they are

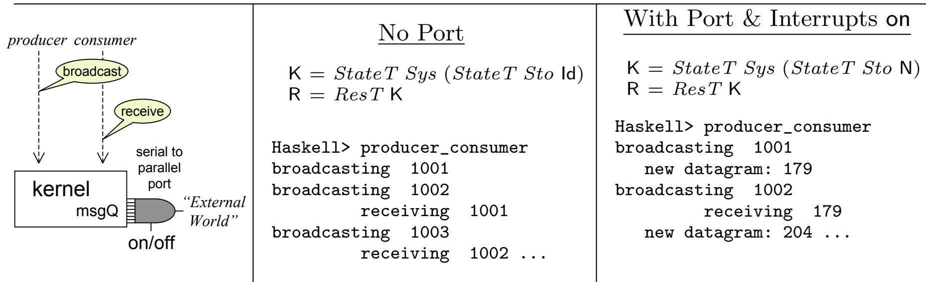


Fig. 1. Kernels with interrupt-driven input port. *The kernel design (left) supports an interrupt-driven serial-to-parallel input port and synchronous message-passing primitives. The FIFO message queue,* msgQ*, stores messages in flight between threads. The input port receives bits non-deterministically from the external world, buffers them, and enqueues a one byte message on* msgQ *when possible. The baseline kernel (middle) has no such input port; the modified asynchronous kernel (right) extends the baseline kernel with non-determinism, thereby supporting the interrupt-driven port. When a producer-consumer application is run on both kernels, the datagrams received through the port are manifest. See text for further description.*

applied instead to the non-determinism monad, N. The addition of asynchronous behaviors requires little more than this refinement of the monad underlying the kernel specification. What precisely all this means and why is the subject of this paper.

2 Background on Monads and Monad Transformers

This section outlines the background material necessary to understand the present work. We must assume of necessity that the reader is familiar with monads. Readers requiring more background should consult the related work (especially, Liang et al. [15]); other readers may skip this section. Section 2.1 contains an overview of the non-determinism monad.

A structure $(\mathsf{M}, \eta, \star)$ is a *monad* if, and only if, M is a type constructor (functor) with associated operations *bind* $(\star : \mathsf{M}\,a \rightarrow (a \rightarrow \mathsf{M}\,b) \rightarrow \mathsf{M}\,b)$ and *unit* $(\eta : a \rightarrow \mathsf{M}\,a)$ obeying the well-known "monad laws" [14].

$$
\begin{array}{lll}
\text{(left-unit)} & (\eta\ v) \star k & = k\ v \\
\text{(right-unit)} & x \star \eta & = x \\
\text{(assoc)} & x \star (\lambda\ v.(k\ v\ \star\ h)) & = (x \star k) \star h
\end{array}
$$

Two such monads are the identity and state monads:

$$
\begin{array}{llll}
\mathsf{Id}a = a & \mathsf{S}\,a = s \rightarrow a \times s & & \mathsf{g_S} : \mathsf{S}s \\
\eta_{\mathsf{Id}} v = v & \eta_{\mathsf{S}} v = \lambda\sigma.(v, \sigma) & & \mathsf{g_S} = \lambda\sigma.(\sigma, \sigma) \\
x \star_{\mathsf{Id}} f = fx & \varphi \star_{\mathsf{S}} f = \lambda\sigma_0.\text{let } (v, \sigma_1) = \varphi\sigma_0 \text{ in } fv\sigma_1 & & \mathsf{u_S} : (s \rightarrow s) \rightarrow \mathsf{S}() \\
 & & & \mathsf{u_S} f = \lambda\sigma.((), f\sigma)
\end{array}
$$

Here, s is a fixed type argument, which can be replaced by any type which is to be "threaded" through the computation. What makes monads interesting is not their bind and unit (these are merely computational glue) but, rather, the operations one can define in terms of the extra computational "stuff" they encapsulate. For example, one may define read and write operations to manipulate the underlying state: Given two monads, M and M', it is natural to ask if their composition, $\mathsf{M} \circ \mathsf{M}'$, is also a monad, but it is well-known that monads generally do not compose in this simple manner [6]. However, *monad transformers* do provide a form of monad composition [6, 15, 17]. When applied to a monad M, a monad transformer T creates a new monad M'. The monad $(\mathsf{StateT}\, s\, \mathsf{Id})$ is identical to the state monad S. The state monad transformer, $(\mathsf{StateT}\, s)$, is shown below.

$$
\begin{array}{llll}
\mathsf{S}a & = \mathsf{StateT}\, s\, \mathsf{M}\, a = s \to \mathsf{M}(a \times s) & \mathsf{u} f & = \lambda\sigma.\, \eta_{\mathsf{M}}((), f\,\sigma) \\
\eta_{\mathsf{S}}\, x & = \lambda\sigma.\, \eta_{\mathsf{M}}(x, \sigma) & \mathsf{g} & = \lambda\sigma.\, \eta_{\mathsf{M}}(\sigma, \sigma) \\
x \star_{\mathsf{S}} f & = \lambda\sigma_0.\, (x\,\sigma_0) \star_{\mathsf{M}} (\lambda(a, \sigma_1).f\,a\,\sigma_1) & \mathit{lift}_{\mathsf{S}}\, x & = \lambda\sigma.\, x \star_{\mathsf{M}} \lambda y.\, \eta_{\mathsf{M}}(y, \sigma)
\end{array}
$$

2.1 Non-determinism as a Monad

Semantically, non-deterministic programs (i.e., those with more than one possible value) may be viewed as returning sets of values rather than just one value [1, 25]. Consider, for example, the *amb* operator of McCarthy (1963). Given two arguments, it returns either one or the other; for example, the value of $1\ \mathit{amb}\ 2$ 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. According to this view, the meaning of $(1\ \mathit{amb}\ 2)$ is simply the set $\{1, 2\}$ and the meaning of $(\mathbf{let}\ x = (1\ \mathit{amb}\ 2)\ \mathbf{in}\ x + x)$ is $\{2, 4\}$. Encoding non-determinism as sets of values is expressed monadically via the finite set monad:

$$
\begin{array}{ll}
\eta\ :\ a \to \mathcal{P}_{\mathrm{fin}}(a) & \star\ :\ \mathcal{P}_{\mathrm{fin}}(a) \to (a \to \mathcal{P}_{\mathrm{fin}}(b)) \to \mathcal{P}_{\mathrm{fin}}(b) \\
\eta\, x = \{\, x\, \} & S \star f = \bigcup (f\ S)
\end{array}
$$

where $f\ S = \{\, f\ x \mid x \in S\, \}$ and $\mathcal{P}_{\mathrm{fin}}(-)$ is set of finite subsets drawn from its argument. In the finite set monad, the meaning of $(e\ \mathit{amb}\ e')$ 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 [28, 6, 14] 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 [27]. 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.

3 A Monadic Model for Asynchronous Exceptions

This section presents a monadic model for asynchronous exceptions (MMAE). The MMAE is an algebra of effects with monads and associated operators expressing notions of non-determinism, concurrency and interactivity. Section 3.1 first defines this algebra of effects and then, in Section 3.2, presents a number of theorems specifying the interactions between the algebraic operators. The convention that operators associated with a monad are referred to as *effects* is followed throughout this paper.

3.1 Monadic Algebra of Effects

This section presents the effect algebra underlying the semantics in Section 4. In Section 5, this algebra will be extended and generalized. The left hand column in Definition 1 below specifies the functors part of three monads using both a category-theoretic notation while the right hand column presents them as data type declarations in the Haskell language. The intention in doing so is to appeal to the widest audience. One might note, however, that the Haskell representations are really approximate (e.g., lists in Haskell may be infinite).

Definition 1 (Functors for monads N,E,R).

Non-deter.	$\mathsf{N}\,A = \mathcal{P}_{\mathit{fin}}(A)$	$\mathbf{type}\ \mathsf{N}\ a$	$= [a]$
		$\mathbf{data}\ \mathit{Err}\ a$	$= \mathit{Ok}\ a \mid \mathit{Error}$
Exceptions	$\mathsf{E}\,A = \mathsf{N}(A + \mathit{Error})$	$\mathbf{type}\ \mathsf{E}\ a$	$= \mathsf{N}(\mathit{Err}\ a)$
Concurrency	$\mathsf{R}\,A = \mu X.\,A + \mathsf{E}\,X$	$\mathbf{data}\ \mathsf{R}\ a$	$= \mathit{Done}\ a \mid \mathit{Pause}\,(\mathsf{E}\,(\mathsf{R}\,a))$

Technical note: In the categorical definition of R (left column, bottom), the binder μX can be taken to indicate the either the least or greatest fixed point solution to the corresponding recursive domain equation. For the semantics in

Section 4, either will suffice as there will be no need to represent infinite computations. In Section 5, it will represent the greatest fixed point.

There is a codebase with Haskell implementations of the monadic constructions presented in this paper [8]. Each of the operator definitions below is followed by Haskell examples demonstrating the operators. For consistency, Haskell concrete syntax is eschewed in favor of the mathematical syntax of this paper. So, for example, Haskell lists representing sets are written with set brackets "{" and "}" rather than with Haskell list constructors "[" and "]" and any other Haskell syntactic details inessential to the presentation are struck altogether.

Definition 2 specifies the unit (η) and bind ($\star$) operators for the N, E and R monads. Discussion motivating the definitions of $\star_{\mathsf{N}}$ and η_{N} can be found in Section 2.1.

Definition 2 (η,$\star$ for monads N,E,R). *The unit* (η) *and bind* ($\star$) *operations have type* $\eta_{\mathsf{m}} : a \rightarrow \mathsf{m}\, a$ *and* $(\star_{\mathsf{m}}) : \mathsf{m}\, a \rightarrow (a \rightarrow \mathsf{m}\, b) \rightarrow \mathsf{m}\, b$ *for monads* $\mathsf{m} = \mathsf{N}, \mathsf{E}, \mathsf{R}$ *and are defined by the following equations:*

$$\begin{aligned} &\eta_{\mathsf{N}} x = \{x\} \\ &\{\varphi_1, \ldots, \varphi_n\} \star_{\mathsf{N}} f = \bigcup (f\, \varphi_i) \end{aligned}$$

$$\begin{aligned} &\eta_{\mathsf{E}} = \eta_{\mathsf{N}} \circ Ok \\ &\varphi \star_{\mathsf{E}} f = \\ &\quad \varphi \star_{\mathsf{N}} \lambda v. \\ &\quad\quad \textbf{case}\ v\ \textbf{of} \\ &\quad\quad\quad (Ok\, x) \rightarrow f\, x \\ &\quad\quad\quad Error \rightarrow \eta_{\mathsf{N}} Error \end{aligned}$$

$$\begin{aligned} &\eta_{\mathsf{R}} = Done \\ &(Done\ v) \star_{\mathsf{R}} f = f\ v \\ &(Pause\ \varphi) \star_{\mathsf{R}} f = \\ &\quad Pause(\varphi \star_{\mathsf{E}} \lambda\kappa.\eta_{\mathsf{E}}(\kappa \star_{\mathsf{R}} f)) \end{aligned}$$

Below are Haskell examples demonstrating the unit and bind operators of the N and E monads. Binding N-computation $\{1, 2, 3\}$ to the function $(\lambda v.\eta_{\mathsf{N}}(v + 1))$ with $\star_{\mathsf{N}}$ has the effect of incrementing each element. Binding E-computation $\{Ok\ 1, Ok\ 2, Error\}$ to the function $(\lambda v.\eta_{\mathsf{E}}(v + 1))$ with $\star_{\mathsf{E}}$ performs a similar "mapping" to the previous case: each $(Ok\ x)$ element is incremented while the *Error* is unchanged.

Haskell> $\eta_{\mathsf{N}}\ 9$
  $\{ 9 \}$
Haskell> $\eta_{\mathsf{E}}\ 9$
  $\{ Ok\ 9 \}$
Haskell> $\{1, 2, 3\} \star_{\mathsf{N}} (\lambda v.\ \eta_{\mathsf{N}}(v + 1))$
  $\{2, 3, 4\}$
Haskell> $\{Ok\ 1, Ok\ 2, Error\} \star_{\mathsf{E}} (\lambda v.\ \eta_{\mathsf{E}}(v + 1))$
  $\{Ok\ 2, Ok\ 3, Error\}$

Notational Convention. A notational convention is borrowed from Haskell. The "null bind" of a monad m, $(\texttt{>>}_{\mathsf{m}})$, is defined as:

$$\begin{aligned} &(\texttt{>>}_{\mathsf{m}}) : \mathsf{m}a \rightarrow \mathsf{m}b \rightarrow \mathsf{m}b \\ &\varphi \texttt{>>}_{\mathsf{m}} \gamma = \varphi \star_{\mathsf{m}} \lambda d.\gamma \end{aligned}$$

where d is a dummy variable not occurring in γ. The effect of the computation, $\varphi \gg_{\mathsf{m}} \gamma$, is to evaluate φ, ignore the value it produces, and then evaluate γ.

The *step* operator takes an E-computation and produces an R-computation; it is an example of a *lifting* [15] from E to R. The lifted computation, *step* x, is atomic in the sense that there are no intermediate *Pause* breakpoints and is, in that sense, indivisible. The *run* operator projects computations from R to E. An R-computation may be thought of intuitively as an E-computation with a number (possibly infinite) of inserted *Pause* breakpoints; *run* removes each of those breakpoints.

Definition 3 (Operators relating E and R).

$$\begin{array}{ll} \mathit{step} : \mathsf{E}\,a \rightarrow \mathsf{R}\,a & \mathit{run} : \mathsf{R}\,a \rightarrow \mathsf{E}\,a \\ \mathit{step}\,x \;=\; \mathit{Pause}(x \star_{\mathsf{E}} (\eta_{\mathsf{E}} \circ \mathit{Done})) & \mathit{run}(\mathit{Pause}\,\varphi) = \varphi \star_{\mathsf{E}} \mathit{run} \\ & \mathit{run}\ (\mathit{Done}\ v) = \eta_{\mathsf{E}}\ v \end{array}$$

The Haskell session below shows that applying *run* to the lifting of an E-computation makes no change to that computation. This suggests that *run* is an inverse of *step*, which is, in fact, the case (see Theorem 5 below).

Haskell> $\mathit{run}\ (\mathit{step}\ \{\mathit{Ok}\ 1, \mathit{Ok}\ 2, \mathit{Error}\})$
  $\{\mathit{Ok}\ 1, \mathit{Ok}\ 2, \mathit{Error}\}$

The *merge* operators on N, E and R are given in Definition 4. As discussed in Section 2.1 above, $\mathit{merge}_{\mathsf{N}}$ is a non-proper morphism in the non-determinism monad. Its definition is lifted to the E and R monads below:

Definition 4 (Merge operators).

$$\begin{array}{l} \mathit{merge}_{\mathsf{N}} : \mathcal{P}_{\mathit{fin}}(\mathsf{N}\,a) \rightarrow \mathsf{N}\,a \\ \mathit{merge}_{\mathsf{N}}\ X = \cup_{(x \in X)}\ x \\ \mathit{merge}_{\mathsf{E}} : \mathcal{P}_{\mathit{fin}}(\mathsf{E}\,a) \rightarrow \mathsf{E}\,a \\ \mathit{merge}_{\mathsf{E}} \quad = \mathit{merge}_{\mathsf{N}} \\ \mathit{merge}_{\mathsf{R}} : \mathcal{P}_{\mathit{fin}}(\mathsf{R}\,a) \rightarrow \mathsf{R}\,a \\ \mathit{merge}_{\mathsf{R}}\{\varphi_1, \ldots, \varphi_n\} = \mathit{Pause}(\mathit{merge}_{\mathsf{E}}\{\eta_{\mathsf{E}}\,\varphi_1, \ldots, \eta_{\mathsf{E}}\,\varphi_n\}) \end{array}$$

The effect of merging some finite number of computations in N or E together is to collect the sets of their outcomes into a single outcome set. Merging in R has a similar effect, but, rather than collecting outcomes, $\mathit{merge}_{\mathsf{R}}\ \{\varphi_1, \ldots, \varphi_n\}$ creates a single R-computation that branches out with n "sub-computations".

Haskell> $\mathit{merge}_{\mathsf{N}}\ \{\{1, 2\}, \{4\}\}$
  $\{1, 2, 4\}$

Haskell> $merge_{\mathsf{E}}\ \{\{Ok\ 1, Ok\ 2, Error\},\ \{Ok\ 4,\ Error\}\}$
  $\{Ok\ 1, Ok\ 2, Ok\ 4, Error\}$ — dupl. *Error* not shown

An equation like " $\langle$raise exception$\rangle \star f\ =\ \langle$raise exception$\rangle$" will hold in any exception monad like E. That is, a raised exception trumps any effects that follow. The *status* operators for E and R catch an exception producing computation and "defuse" it. So, if $\varphi : \mathsf{E}\ a$ produces an exception, then the value returned by $status_{\mathsf{E}}(\varphi) : \mathsf{E}\,(a\ +\ Error)$ will be *Error* itself. The *status* operators may be used to discern when an exception has occurred in its argument while isolating the exception's effect.

Definition 5 (Status).

$$
\begin{array}{l}
status_{\mathsf{E}} : \mathsf{E}\,a \rightarrow \mathsf{E}(a + Error)\\
status_{\mathsf{E}}\varphi =\\
\quad \varphi \star_{\mathsf{N}} \lambda v.\\
\qquad \textbf{case}\ v\ \textbf{of}\\
\qquad\quad (Ok\ y)\ \rightarrow \eta_{\mathsf{E}}(Ok\ y)\\
\qquad\quad Error\ \ \rightarrow \eta_{\mathsf{E}} Error
\end{array}
\qquad
\begin{array}{l}
status_{\mathsf{R}} : \mathsf{R}\,a \rightarrow \mathsf{R}\,(a + Error)\\
status_{\mathsf{R}}\ (Pause\ \varphi) =\\
\quad Pause\,(status_{\mathsf{E}}\ \varphi \star_{\mathsf{E}} \lambda v.\\
\qquad \textbf{case}\ v\ \textbf{of}\\
\qquad\quad (Ok\ x) \rightarrow \eta_{\mathsf{E}}(status_{\mathsf{R}}\ x)\\
\qquad\quad Error\ \rightarrow \eta_{\mathsf{E}}(Done\ Error))\\
status_{\mathsf{R}}\ (Done\ v)\ = (Done\ (Ok\ v))
\end{array}
$$

The $throw_{\mathsf{E}} : \mathsf{E}\ a$ operator raises an exception (it is defined below in Definition 6). Note how (in the second example) it "trumps" the remaining computation. ($status_{\mathsf{E}}\ throw_{\mathsf{E}}$), in the third example shows how the effect of the $throw_{\mathsf{E}}$ exception is isolated. Instead of returning the exception $\{\ Error\ \}$, the ($status_{\mathsf{E}}\ throw_{\mathsf{E}}$) computation returns the *Error* token as its value. With a non-exception throwing computation (e.g., $(\eta_{\mathsf{E}}\ 9)$), $status_{\mathsf{E}}$ returns the value produced by its argument wrapped in an *Ok*.

Haskell> $throw_{\mathsf{E}}$
  $\{\ Error\ \}$
Haskell> $throw_{\mathsf{E}} \star_{\mathsf{E}} \lambda v.\ \eta_{\mathsf{E}}\ (v + 1)$
  $\{\ Error\ \}$
Haskell> $status_{\mathsf{E}}\ throw_{\mathsf{E}}$
  $\{Ok\ Error\}$
Haskell> $status_{\mathsf{E}}\ (\eta_{\mathsf{E}}\ 9)$
  $\{Ok\ (Ok\ 9)\}$

Definition 6 gives the specification for the exception-raising and -catching operations, *throw* and *catch*, in E and R and the branching operation, *fork*, in the R monad. The *fork* operation is particularly important in the semantic framework of the next section. If $\varphi : \mathsf{R}\ a$, then $(fork\ \varphi) : \mathsf{R}\ a$ is a computation that, roughly speaking, will do either whatever φ would do or be asynchronously interrupted by exception $throw_{\mathsf{R}}$.

Definition 6 (Control flow operators).

$$
\begin{array}{ll}
\mathit{throw}_{\mathsf{E}} : \mathsf{E}\,a & \text{For monad } \mathsf{m} = \mathsf{E}, \mathsf{R}, \\
\mathit{throw}_{\mathsf{E}} = \eta_{\mathsf{N}}\ \mathit{Error} & \mathit{catch}_{\mathsf{m}} : \mathsf{m}\,a \to \mathsf{m}\,a \to \mathsf{m}\,a \\
\mathit{throw}_{\mathsf{R}} : \mathsf{R}\,a & \mathit{catch}_{\mathsf{m}}\ \varphi\ \gamma = (\mathit{status}_{\mathsf{m}}\ \varphi) \star_{\mathsf{m}} \lambda s. \\
\mathit{throw}_{\mathsf{R}} = \mathit{step}\ \mathit{throw}_{\mathsf{E}} & \quad\quad \mathbf{case}\ \ s\ \ \mathbf{of} \\
\mathit{fork} : \mathsf{R}\,a \to \mathsf{R}\,a & \quad\quad\quad (\mathit{Ok}\,v) \ \to\ \eta_{\mathsf{m}}\ v \\
\mathit{fork}\ \varphi\ = \mathit{merge}_{\mathsf{R}}\ \{\varphi, \mathit{throw}_{\mathsf{R}}\} & \quad\quad\quad \mathit{Error}\ \to\ \gamma
\end{array}
$$

The first two examples in the following Haskell transcript illustrate the behavior of $\mathit{catch}_{\mathsf{E}}$ and $\mathit{throw}_{\mathsf{E}}$. If the first argument to $\mathit{catch}_{\mathsf{E}}$ raises an exception (as, obviously, $\mathit{throw}_{\mathsf{E}}$ does), then the second argument is returned. If the first argument to *catch* does not raise an exception, then its second argument is ignored. The third and fourth examples illustrate the *fork* effect. The effect of $\mathit{fork}\ (\eta_{\mathsf{R}}\ 9)$ is to create a single R-computation with two "branches" (both underlined below). The first branch is just the argument to *fork* (recall that $\eta_{\mathsf{R}} = \mathit{Done}$) and the second branch is equal to $(\mathit{step}\ \mathit{throw}_{\mathsf{E}})$. This simple example exposes an important construction within the MMAE: an asynchronous exception thrown to a computation is modeled as an alternative branch from the computation. The fourth example shows the application of *run* to the previous example. The result is to collect all possible outcomes from each branch. Following a tree data type analogy, *run* is used to calculate the *fringe*.

Haskell> $\mathit{catch}_{\mathsf{E}}\ \mathit{throw}_{\mathsf{E}}\ (\eta_{\mathsf{E}}\ 9)$
  $\{\mathit{Ok}\ 9\}$
Haskell> $\mathit{catch}_{\mathsf{E}}\ (\eta_{\mathsf{E}}\ 9)\ \mathit{throw}_{\mathsf{E}}$
  $\{\mathit{Ok}\ 9\}$
Haskell> $\mathit{fork}\ (\eta_{\mathsf{R}}\ 9)$
  $\mathit{Pause}\ (\{\ \mathit{Ok}\ \underline{(\mathit{Done}\ 9)},\ \mathit{Ok}\ (\underline{\mathit{Pause}\ (\{\ \mathit{Error}\ \}\ )})\})$
Haskell> $\mathit{run}\ (\mathit{fork}\ (\eta_{\mathsf{R}}\ 9))$
  $\{\mathit{Ok}\ 9, \mathit{Error}\}$

3.2 Interactions Between Effects

This section presents a number of theorems characterizing the algebraic effects developed in the previous section and their relationships to one another. These theorems are used in the next section to prove an equivalence between the MMAE semantics given there and recently published semantics for asynchronous exceptions [12].

Theorem 1 gives a distribution rule for $\star$ over *merge*. This distribution exposes the tree-like structure of computations in R.

Theorem 1. *For monads* $\mathsf{m} = \mathsf{N}, \mathsf{E}, \mathsf{R}$, $\star_{\mathsf{m}}$ *distributes over* $\mathit{merge}_{\mathsf{m}}$*:*

$$\mathit{merge}_{\mathsf{m}}\{\varphi_1, \ldots, \varphi_n\} \star_{\mathsf{m}} f = \mathit{merge}_{\mathsf{m}}\{\varphi_1 \star_{\mathsf{m}} f, \ldots, \varphi_n \star_{\mathsf{m}} f\}$$

Proof.

$$
\begin{array}{ll}
 & \mathit{merge}_{\mathsf{N}}\{\varphi_1,\ldots,\varphi_n\}\star_{\mathsf{N}} f \\
\{\text{def } \mathit{merge}_{\mathsf{N}}\} & = (\cup\,\varphi_i)\star_{\mathsf{N}} f \\
\{\text{def } \star_{\mathsf{N}}\} & = f\ (\cup\,\varphi_i) \\
 & = \cup\,(f\ \varphi_i) \\
\{\text{def } \mathit{merge}_{\mathsf{N}}\} & = \mathit{merge}_{\mathsf{N}}\ \{f\ \varphi_1,\ldots,f\ \varphi_n\} \\
\{\text{def } \star_{\mathsf{N}}\} & = \mathit{merge}_{\mathsf{N}}\ \{\varphi_1\star_{\mathsf{N}} f,\ldots,\varphi_n\star_{\mathsf{N}} f\}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{merge}_{\mathsf{E}}\{\varphi_1,\ldots,\varphi_n\}\star_{\mathsf{E}} f \\
 & = \mathit{merge}_{\mathsf{E}}\ \{\varphi_1,\ldots,\varphi_n\}\star_{\mathsf{N}}\hat{f} \\
 & \quad \text{where } \hat{f}\ v = \ \ \mathbf{case}\ v\ \mathbf{of} \\
 & \qquad\qquad (\mathit{Ok}\ x) \ \rightarrow\ \eta_{\mathsf{N}}\ (f\ x) \\
 & \qquad\qquad \mathit{Error}\ \rightarrow\ \eta_{\mathsf{N}}\ \mathit{Error} \\
\{\text{def } \mathit{merge}_{\mathsf{E}}\} & = \mathit{merge}_{\mathsf{N}}\ \{\varphi_1,\ldots,\varphi_n\}\star_{\mathsf{N}}\hat{f} \\
\{\text{prev. case}\} & = \mathit{merge}_{\mathsf{N}}\ \{\varphi_1\star_{\mathsf{N}}\hat{f},\ldots,\varphi_n\star_{\mathsf{N}}\hat{f}\} \\
\{\text{def } \star_{\mathsf{E}}\} & = \mathit{merge}_{\mathsf{N}}\ \{\varphi_1\star_{\mathsf{E}} f,\ldots,\varphi_n\star_{\mathsf{E}} f\} \\
\{\text{def } \mathit{merge}_{\mathsf{E}}\} & = \mathit{merge}_{\mathsf{E}}\ \{\varphi_1\star_{\mathsf{E}} f,\ldots,\varphi_n\star_{\mathsf{E}} f\}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{merge}_{\mathsf{R}}\ \{\varphi_1,\ldots,\varphi_n\}\star_{\mathsf{R}} f \\
\{\text{def } \mathit{merge}_{\mathsf{R}}\} & = \mathit{Pause}\,(\mathit{merge}_{\mathsf{E}}\ \{\eta_{\mathsf{E}}\ \varphi_1,\ldots,\eta_{\mathsf{E}}\ \varphi_n\})\star_{\mathsf{R}} f \\
\{\text{def } \star_{\mathsf{R}}\} & = \mathit{Pause}\,(\mathit{merge}_{\mathsf{E}}\ \{\eta_{\mathsf{E}}\ \varphi_1,\ldots,\eta_{\mathsf{E}}\ \varphi_n\}\star_{\mathsf{E}}\lambda\kappa.\ \eta_{\mathsf{E}}\ (\kappa\star_{\mathsf{R}} f)) \\
\{\text{prev. case}\} & = \mathit{Pause}\,(\mathit{merge}_{\mathsf{E}}\ \{\varphi'_1,\ldots,\varphi'_n\}) \\
 & \quad \text{where } \varphi'_i\ =\ (\eta_{\mathsf{E}}\ \varphi_i)\star_{\mathsf{E}}\lambda\kappa.\ \eta_{\mathsf{E}}\ (\kappa\star_{\mathsf{R}} f) \\
\{\text{left unit}\} & = \mathit{Pause}\,(\mathit{merge}_{\mathsf{E}}\ \{\ \eta_{\mathsf{E}}\ (\varphi_1\star_{\mathsf{R}} f),\ldots,\ \eta_{\mathsf{E}}\ (\varphi_n\star_{\mathsf{R}} f)\}) \\
\{\text{def } \mathit{merge}_{\mathsf{R}}\} & = \mathit{merge}_{\mathsf{R}}\ \{\varphi_1\star_{\mathsf{R}} f,\ldots,\ \varphi_n\star_{\mathsf{R}} f\}
\end{array}
$$

□

Theorem 2 gives a distribution law for *run* over $\mathit{merge}_{\mathsf{R}}$. This distribution law is something like an inverse "lifting" as it projects resumption-based merged computations into a single computation in E.

Theorem 2. $\mathit{run}(\mathit{merge}_{\mathsf{R}}\{\varphi_1,\ldots,\varphi_n\}) = \mathit{merge}_{\mathsf{E}}\{\mathit{run}\ \varphi_1,\ldots,\mathit{run}\ \varphi_n\}$

Proof.

$$
\begin{array}{ll}
 & \mathit{run}\,(\mathit{merge}_{\mathsf{R}}\ \{\varphi_1,\ldots,\varphi_n\}) \\
\{\text{def } \mathit{merge}_{\mathsf{R}}\ \} & = \mathit{run}\ (\mathit{Pause}\,(\mathit{merge}_{\mathsf{E}}\ \{\eta_{\mathsf{E}}\ \varphi_1,\ldots,\eta_{\mathsf{E}}\ \varphi_n\})) \\
\{\text{def } \mathit{run}\ \} & = (\mathit{merge}_{\mathsf{E}}\ \{\eta_{\mathsf{E}}\ \varphi_1,\ldots,\eta_{\mathsf{E}}\ \varphi_n\})\star_{\mathsf{E}}\mathit{run} \\
\{\text{thm 1}\} & = \mathit{merge}_{\mathsf{E}}\ \{(\eta_{\mathsf{E}}\ \varphi_1)\star_{\mathsf{E}}\mathit{run}\,,\ldots,(\eta_{\mathsf{E}}\ \varphi_n)\star_{\mathsf{E}}\mathit{run}\} \\
\{\text{left unit}\} & = \mathit{merge}_{\mathsf{E}}\ \{\mathit{run}\,\varphi_1,\ldots,\ \mathit{run}\,\varphi_n\}
\end{array}
$$

□

Theorem 3 shows how *run* distributes over $\star_{\mathsf{R}}$ to produce and E-computation. Theorem 3 may be proved easily by induction on the number of *Pause* constructors in its argument if that number is finite. If it is not, then the property is trivially true, because, in that case, both sides of Theorem 3 are $\bot$.

Theorem 3. $run(x \star_{\mathsf{R}} f) = (run\ x) \star_{\mathsf{E}} (run \circ f)$

Theorem 4 shows that the *throw* exception overrides any effects following it. A consequence of this theorem is that, for any $f, g : a \to \mathsf{E}\, b$

$$throw_{\mathsf{E}} \star_{\mathsf{E}} f \;=\; throw_{\mathsf{E}} \;=\; throw_{\mathsf{E}} \star_{\mathsf{E}} g$$

Theorem 4. *$throw_{\mathsf{m}} \star_{\mathsf{m}} f = throw_{\mathsf{m}}$, for monad $\mathsf{m} = \mathsf{E}, \mathsf{R}$.*

Proof.

$$\begin{array}{ll}
 & throw_{\mathsf{E}} \star_{\mathsf{E}} f \\
\{\text{def } throw_{\mathsf{E}}\} & = (\eta_{\mathsf{N}}\ Error) \star_{\mathsf{E}} f \\
\{\text{def } \star_{\mathsf{E}}\} & = (\eta_{\mathsf{N}}\ Error) \star_{\mathsf{N}} \lambda v. \\
 & \qquad\quad \mathbf{case}\ v\ \mathbf{of}\ \{\ (Ok\, x)\ \to\ \eta_{\mathsf{N}}\ (f\ x)\ ;\ Error\ \to\ \eta_{\mathsf{N}}\ Error\ \} \\
\{\text{left unit}\} & = \mathbf{case}\ Error\ \mathbf{of}\ \{\ (Ok\, x)\ \to\ \eta_{\mathsf{N}}\ (f\ x)\ ;\ Error\ \to\ \eta_{\mathsf{N}}\ Error\ \} \\
 & = \eta_{\mathsf{N}}\ Error\ =\ throw_{\mathsf{E}}
\end{array}$$

$$\begin{array}{ll}
 & throw_{\mathsf{R}} \star_{\mathsf{R}} f \\
\{\text{def } throw_{\mathsf{R}}\} & = (step\ throw_{\mathsf{E}}) \star_{\mathsf{R}} f \\
\{\text{def } step\} & = (Pause\ (throw_{\mathsf{E}} \star_{\mathsf{E}} (\eta_{\mathsf{E}}\ \circ\ Done)) \star_{\mathsf{R}} f \\
\{\text{def } \star_{\mathsf{R}}\} & = Pause\ (throw_{\mathsf{E}} \star_{\mathsf{E}} \lambda\kappa.\ (\eta_{\mathsf{E}}\ (\kappa \star_{\mathsf{R}} f))) \\
\{\text{prev. case}\} & = Pause\ (throw_{\mathsf{E}} \star_{\mathsf{E}} (\eta_{\mathsf{E}}\ \circ\ Done)) =\ step\ throw_{\mathsf{E}}\ =\ throw_{\mathsf{R}}
\end{array}$$

□

Theorem 5 states that the *run* operation is the inverse of *step*. A consequence of this theorem is that $run\ throw_{\mathsf{R}}\ =\ throw_{\mathsf{E}}$.

Theorem 5. $run\ (step\ \varphi)\ =\ \varphi$

Proof.

$$\begin{array}{ll}
 & run\,(step\varphi) \\
\{\text{def } step\} & = run\ (Pause\ (\varphi \star_{\mathsf{E}} (\eta_{\mathsf{E}} \circ Done)) \\
\{\text{def } run\} & = (\varphi \star_{\mathsf{E}} (\eta_{\mathsf{E}} \circ Done)) \star_{\mathsf{E}} run \\
\{\text{assoc.}\} & = \varphi \star_{\mathsf{E}} \lambda v.\ \eta_{\mathsf{E}}(Done\ v) \star_{\mathsf{E}} run \\
\{\text{left unit}\} & = \varphi \star_{\mathsf{E}} \lambda v.\ run\ (Done\ v) \\
\{\text{def } run\} & = \varphi \star_{\mathsf{E}} \lambda v.\ \eta_{\mathsf{E}}\ v \\
\{\text{eta red}\} & = \varphi \star_{\mathsf{E}} \eta_{\mathsf{E}} \\
\{\text{rt unit}\} & = \varphi
\end{array}$$

□

4 The MMAE as a Semantic Framework

This section presents the semantics for the exception language of Hutton [12] in terms of the MMAE. Hutton defines a natural semantics for a small language combining arithmetic expressions and synchronous exceptions (e.g., *Catch* and *Throw*). The natural semantics of this synchronous fragment of the language is just what one would expect. What makes the language interesting is the presence of an asynchronous interrupt and its manifestation within the semantics. Hutton's language and its natural semantics are found in Figure 2. A monadic semantics for Hutton's language is given below and then the equivalence of both semantics is formulated in Theorem 6.

$$\frac{}{\mathit{Val}\ n \Downarrow^i \mathit{Val}\ n}\ \text{Val} \qquad \frac{}{\mathit{Throw} \Downarrow^i \mathit{Throw}}\ \text{Throw} \qquad \frac{x \Downarrow^i \mathit{Val}\ n \quad y \Downarrow^i \mathit{Val}\ m}{\mathit{Add}\ x\ y \Downarrow^i \mathit{Val}\ (n{+}m)}\ \text{Add1}$$

$$\frac{x \Downarrow^i \mathit{Throw}}{\mathit{Add}\ x\ y \Downarrow^i \mathit{Throw}}\ \text{Add2} \qquad \frac{y \Downarrow^i \mathit{Throw}}{\mathit{Add}\ x\ y \Downarrow^i \mathit{Throw}}\ \text{Add3} \qquad \frac{y \Downarrow^i v}{\mathit{Seqn}\ x\ y \Downarrow^i v}\ \text{Seqn1}$$

$$\frac{x \Downarrow^i \mathit{Throw}}{\mathit{Seqn}\ x\ y \Downarrow^i \mathit{Throw}}\ \text{Seqn2} \qquad \frac{x \Downarrow^i \mathit{Val}\ n}{\mathit{Catch}\ x\ y \Downarrow^i \mathit{Val}\ n}\ \text{Catch1} \qquad \frac{x \Downarrow^i \mathit{Throw} \quad y \Downarrow^i v}{\mathit{Catch}\ x\ y \Downarrow^i v}\ \text{Catch2}$$

$$\frac{x \Downarrow^B v}{\mathit{Block}\ x \Downarrow^i v}\ \text{Block} \qquad \frac{x \Downarrow^U v}{\mathit{Unblock}\ x \Downarrow^i v}\ \text{Unblock} \qquad \frac{}{x \Downarrow^U \mathit{Throw}}\ \text{Int}$$

Fig. 2. *Hutton's Expression Language, Expr, with Asynchronous Exceptions. The essence of asynchronous exceptions in this model is captured in the Int rule.*

This section presents the formulation of Hutton's language within the monadic framework of Section 3. The evaluation relation is annotated by a B or U, indicating whether interrupts are blocked or unblocked, respectively. Ignoring this flag for the moment, the first three rows in Figure 2 are a conventional, natural semantics for a language combining arithmetic with synchronous exceptions. Note, for example, that the interaction of *Add* with *Throw* is specified by three rules (*Add*1, *Add*2, *Add*3), the first for the case of exception-free arguments and the second two for the cases when an argument evaluates to *Throw*. The effect of (*Block* e) [(*Unblock* e)] is to turn off [on] asynchronous exceptions in the evaluation of e.

To understand how Hutton's semantics works and to compare it with the monadic semantics given here, the expression (*Add* (*Val* 1) (*Val* 2)) will be used as a running example. There are four possible evaluations of (*Add* (*Val* 1) (*Val* 2)) when the interrupt flag is U (shown in Figure 3). The evaluation in the upper left is what one would expect. But the three other cases involve asynchronous exceptions, evaluating instead to *Throw* via the *Int* rule from Figure 2. The bottom row shows what happens when the first and second arguments, respectively, evaluate to *Throw* and, in the upper right corner, the evaluation is interrupted "before" any computation takes place. The *Int* rule may be applied because the

$$\dfrac{\dfrac{}{\mathit{Val\ 1} \Downarrow^U \mathit{Val\ 1}}\ \text{Val} \qquad \dfrac{}{\mathit{Val\ 2} \Downarrow^U \mathit{Val\ 2}}\ \text{Val}}{\mathit{Add\ (Val\ 1)\ (Val\ 2)} \Downarrow^U \mathit{Val\ 3}}\ \text{Add1} \qquad \dfrac{}{\mathit{Add\ (Val\ 1)\ (Val\ 2)} \Downarrow^U \mathit{Throw}}\ \text{Int}$$

$$\dfrac{\dfrac{}{\mathit{Val\ 1} \Downarrow^U \mathit{Throw}}\ \text{Int}}{\mathit{Add\ (Val\ 1)\ (Val\ 2)} \Downarrow^U \mathit{Throw}}\ \text{Add2} \qquad \dfrac{\dfrac{}{\mathit{Val\ 2} \Downarrow^U \mathit{Throw}}\ \text{Int}}{\mathit{Add\ (Val\ 1)\ (Val\ 2)} \Downarrow^U \mathit{Throw}}\ \text{Add3}$$

Fig. 3. *There are four possible evaluations of* $(\mathit{Add}\ (\mathit{Val}\ 1)\,(\mathit{Val}\ 2))$ *according to the unblocked semantics (i.e., with* U *annotation) in Figure 2.*

flag is U and consequently there are four possible evaluations. When the flag is B, the *Int* rule may not be applied, and, hence, exceptions are blocked. Thus, there is one and only one evaluation when the flag is B:

$$\dfrac{\dfrac{}{\mathit{Val\ 1} \Downarrow^B \mathit{Val\ 1}}\ \text{Val} \qquad \dfrac{}{\mathit{Val\ 2} \Downarrow^B \mathit{Val\ 2}}\ \text{Val}}{\mathit{Add\ (Val\ 1)\ (Val\ 2)} \Downarrow^B \mathit{Val\ 3}}\ \text{Add1}$$

Figure 4 presents the semantics of *Expr* using the MMAE framework. The semantics consists of two semantic functions, the "blocked" semantics $\mathcal{B}[\![-]\!]$ and the "unblocked" semantics $\mathcal{U}[\![-]\!]$. The blocked (unblocked) semantics provides the meaning of an expression when interrupts are off (on) and corresponds to the natural semantic relation $\Downarrow^B$ ($\Downarrow^U$). The first five semantic equations for $\mathcal{B}[\![-]\!]$ are a conventional monadic semantics for arithmetic, sequencing and synchronous exceptions [21]. The definitions for *Block* and *Unblock* require comment, however. The meaning of $(\mathit{Block}\ e)$ in the $\mathcal{B}[\![-]\!]$ has no effect, because asynchronous exceptions are already blocked. The meaning of $(\mathit{Unblock}\ e)$, however, is $\mathcal{U}[\![e]\!]$, signifying thereby that asynchronous exceptions are unblocked in the evaluation of e. The unblocked semantics is similar to the blocked, except that *fork* is applied to each denoting computation. This has the effect of creating an asynchronous exception at each application of *fork*. $\mathcal{U}[\![\mathit{Block}\,e]\!]$ is defined as $\mathcal{B}[\![e]\!]$ to "turn off" asynchronous exceptions.

Example: Blocked Semantics. In the following example, the "blocked" denotation of *Add (Val 1) (Val 2)* is simplified with respect to the theorems of Section 3.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{B}[\![\mathit{Add\ (Val\ 1)\ (Val\ 2)}]\!]} \\
\{\text{def } \mathcal{B}[\![-]\!]\} & = (\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.(\mathit{step}(\eta_{\mathsf{E}}\ 2)) \star_{\mathsf{R}} \lambda v_2.\eta_{\mathsf{R}}(v_1 + v_2) \\
\{\text{def } \star_{\mathsf{R}}\} & = (\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.(\mathit{Pause}\ ((\eta_{\mathsf{E}}\ 2) \star_{\mathsf{E}} (\eta_{\mathsf{E}} \circ \mathit{Done}))) \star_{\mathsf{R}} \lambda v_2.\eta_{\mathsf{R}}(v_1 + v_2) \\
\{\text{left unit}\} & = (\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.(\mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{Done}\ 2))) \star_{\mathsf{R}} \lambda v_2.\eta_{\mathsf{R}}(v_1 + v_2) \\
\{\text{def } \star_{\mathsf{R}}\} & = (\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.\mathit{Pause}\ (\eta_{\mathsf{E}}\ ((\mathit{Done}\ 2) \star_{\mathsf{R}} \lambda v_2.\eta_{\mathsf{R}}(v_1 + v_2))) \\
\{\text{def } \star_{\mathsf{R}}\} & = (\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.(\mathit{Pause}\ (\eta_{\mathsf{E}}\ (\eta_{\mathsf{R}}(v_1 + 2)))) \\
 & = \mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{Done}\ 3))))
\end{array}$$

$$
\begin{array}{lll|lll}
\mathcal{B}[\![-]\!] : \mathit{Expr} \to \mathsf{R}\,\mathit{Int} & & & \mathcal{U}[\![-]\!] : \mathit{Expr} \to \mathsf{R}\,\mathit{Int} & & \\
\mathcal{B}[\![\mathit{Val}\ i]\!] & = & \mathit{step}(\eta_{\mathsf{E}} i) & \mathcal{U}[\![\mathit{Val}\ i]\!] & = & \mathit{fork}(\mathit{step}(\eta_{\mathsf{E}} i)) \\
\mathcal{B}[\![\mathit{Add}\, e_1\ e_2]\!] & = & \mathcal{B}[\![e_1]\!] \star_{\mathsf{R}} \lambda v_1. \; \mathcal{B}[\![e_2]\!] \star_{\mathsf{R}} \lambda v_2. \; \eta_{\mathsf{R}}(v_1 + v_2) & \mathcal{U}[\![\mathit{Add}\, e_1\ e_2]\!] & = & \mathit{fork}\begin{pmatrix}\mathcal{U}[\![e_1]\!] \star_{\mathsf{R}} \lambda v_1. \\ \mathcal{U}[\![e_2]\!] \star_{\mathsf{R}} \lambda v_2. \\ \eta_{\mathsf{R}}(v_1 + v_2)\end{pmatrix} \\
\mathcal{B}[\![\mathit{Seqn}\, e_1\ e_2]\!] & = & \mathcal{B}[\![e_1]\!] \gg_{\mathsf{R}} \mathcal{B}[\![e_2]\!] & \mathcal{U}[\![\mathit{Seqn}\, e_1\ e_2]\!] & = & \mathit{fork}(\mathcal{U}[\![e_1]\!] \gg_{\mathsf{R}} \mathcal{U}[\![e_2]\!]) \\
\mathcal{B}[\![\mathit{Throw}]\!] & = & \mathit{throw}_{\mathsf{R}} & \mathcal{U}[\![\mathit{Throw}]\!] & = & \mathit{fork}\ \mathit{throw}_{\mathsf{R}} \\
\mathcal{B}[\![\mathit{Catch}\, e_1\ e_2]\!] & = & \mathit{catch}_{\mathsf{R}}\,(\mathcal{B}[\![e_1]\!])\,(\mathcal{B}[\![e_2]\!]) & \mathcal{U}[\![\mathit{Catch}\, e_1\ e_2]\!] & = & \mathit{fork}(\mathit{catch}_{\mathsf{R}}\,(\mathcal{U}[\![e_1]\!])\,(\mathcal{U}[\![e_2]\!])) \\
\mathcal{B}[\![\mathit{Block}\, e]\!] & = & \mathcal{B}[\![e]\!] & \mathcal{U}[\![\mathit{Block}\, e]\!] & = & \mathit{fork}\ \mathcal{B}[\![e]\!] \\
\mathcal{B}[\![\mathit{Unblock}\, e]\!] & = & \mathcal{U}[\![e]\!] & \mathcal{U}[\![\mathit{Unblock}\, e]\!] & = & \mathit{fork}\ \mathcal{U}[\![e]\!]
\end{array}
$$

Fig. 4. Monadic Semantics for Hutton's Language using MMAE Framework

The last line follows by a similar argument to the first five steps. This last R-computation, $\mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{Done}\ 3))))$, captures the operational content of $\mathcal{B}[\![\mathit{Add}\,(\mathit{Val} 1)\,(\mathit{Val} 2)]\!]$. It is a single thread with two steps in succession, corresponding to the evaluation of $(\mathit{Val}\ 1)$ and $(\mathit{Val}\ 2)$, followed by the return of the computed value 3.

Example: Unblocked Semantics. The next example considers the same expression evaluated under the "unblocked" semantics, $\mathcal{U}[\![-]\!]$. As in the previous example, the denotation is "normalized", so to speak, according to the theorems of Section 3. Before beginning the example of the unblocked semantics, we state and prove a useful simplification in Lemma 1.

Lemma 1. $\mathit{step}\ (\eta_{\mathsf{E}}\ v) \star_{\mathsf{R}} f \ = \ \mathit{Pause}\ (\eta_{\mathsf{E}}\ (f\ v))$.

Proof.

$$
\begin{array}{ll}
 & \mathit{step}\,(\eta_{\mathsf{E}} v) \star_{\mathsf{R}} f \\
\{\text{def } \mathit{step}\} & = \mathit{Pause}\ ((\eta_{\mathsf{E}}\ v) \star_{\mathsf{E}} (\eta_{\mathsf{E}} \circ \mathit{Done})) \star_{\mathsf{R}} f \\
\{\text{left unit}\} & = \mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{Done}\ v)) \star_{\mathsf{R}} f \\
\{\text{def } \star_{\mathsf{R}}\} & = \mathit{Pause}\ ((\eta_{\mathsf{E}}\ (\mathit{Done}\ v)) \star_{\mathsf{E}} \lambda\kappa.\ \eta_{\mathsf{E}}(\kappa \star_{\mathsf{R}} f)) \\
\{\text{left unit}\} & = \mathit{Pause}\ (\eta_{\mathsf{E}}\ ((\mathit{Done}\ v) \star_{\mathsf{R}} f)) \\
\{\text{def } \star_{\mathsf{R}}\} & = \mathit{Pause}\ (\eta_{\mathsf{E}}\ (f\ v))
\end{array}
$$

□

The unblocked semantics for *Add (Val 1) (Val 2)* unfolds as follows:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{U}[\![\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2)]\!]} \\
 & = \mathit{fork}\,(\mathit{fork}\,(\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.\mathit{fork}\,(\mathit{step}(\eta_{\mathsf{E}}\ 2)) \star_{\mathsf{R}} \lambda v_2.\eta_{\mathsf{R}}(v_1 + v_2)) \\
\{\text{thm 1,def 6}\} & \\
 & = \mathit{fork}\,(\mathit{fork}\,(\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.\mathit{merge}_{\mathsf{R}}\,\{(\mathit{step}(\eta_{\mathsf{E}}\ 2)) \star_{\mathsf{R}} f,\ \mathit{throw}_{\mathsf{R}} \star_{\mathsf{R}} f\}) \\
 & \qquad \text{where } f = \lambda v_2.\eta_{\mathsf{R}}(v_1 + v_2) \\
\{\text{thm 4}\} & = \mathit{fork}\,(\mathit{fork}\,(\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.\mathit{merge}_{\mathsf{R}}\,\{(\mathit{step}(\eta_{\mathsf{E}}\ 2)) \star_{\mathsf{R}} f,\ \mathit{throw}_{\mathsf{R}}\ \}) \\
\{\text{lem 1}\} & = \mathit{fork}\,(\mathit{fork}\,(\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.\mathit{merge}_{\mathsf{R}}\,\{\mathit{Pause}\ (\eta_{\mathsf{E}}\ (f\ 2)),\ \mathit{throw}_{\mathsf{R}}\ \})
\end{array}
$$

$$
\begin{aligned}
&\{\text{thm 1,def 6}\}\\
&\quad = \mathit{fork}\,(\mathit{merge}_{\mathsf{R}}\,\{\,(\mathit{step}(\eta_{\mathsf{E}}\,1)) \star_{\mathsf{R}} f',\mathit{throw}_{\mathsf{R}} \star_{\mathsf{R}} f'\})\\
&\qquad\quad \text{where } f' = \lambda v_1.\mathit{merge}_{\mathsf{R}}\,\{\mathit{Pause}\,(\eta_{\mathsf{E}}\,(f\,2)),\mathit{throw}_{\mathsf{R}}\}\\
&\{\text{thm 4}\} \quad = \mathit{fork}\,(\mathit{merge}_{\mathsf{R}}\,\{\,(\mathit{step}(\eta_{\mathsf{E}}\,1)) \star_{\mathsf{R}} f',\mathit{throw}_{\mathsf{R}}\,\})\\
&\{\text{lem 1}\} \quad = \mathit{fork}\,(\mathit{merge}_{\mathsf{R}}\,\{\,\mathit{Pause}\,(\eta_{\mathsf{E}}\,(f'\,1),\mathit{throw}_{\mathsf{R}}\,\})\\
&\{\text{def } f,f'\} \quad = \mathit{fork}\,(\mathit{merge}_{\mathsf{R}}\,\{\,\mathit{Pause}\,(\eta_{\mathsf{E}}\,(\mathit{merge}_{\mathsf{R}}\,\{\mathit{Pause}\,(\eta_{\mathsf{E}}\,3),\ \mathit{throw}_{\mathsf{R}}\,\})),\mathit{throw}_{\mathsf{R}}\,\})\\
&\{\text{def } \mathit{fork}\} = \mathit{merge}_{\mathsf{R}} \left\{\begin{array}{l} \mathit{merge}_{\mathsf{R}} \left\{\begin{array}{l}\mathit{Pause}\,(\eta_{\mathsf{E}}\,(\mathit{merge}_{\mathsf{R}} \left\{\begin{array}{l}\mathit{Pause}\,(\eta_{\mathsf{E}}\,(\eta_{\mathsf{R}}\,3)),\\ \mathit{throw}_{\mathsf{R}}\end{array}\right\})),\\ \mathit{throw}_{\mathsf{R}}\end{array}\right\},\\ \\ \mathit{throw}_{\mathsf{R}}\end{array}\right\}
\end{aligned}
$$

The last term in the above evaluation is written in a manner that emphasizes the underlying tree-like structure of denotations in R. The $\mathit{merge}_{\mathsf{R}}$ operators play the rôle of a "tree branch" constructor with $\mathit{throw}_{\mathsf{R}}$ and $\mathit{Pause}\,(\eta_{\mathsf{E}}\,(\eta_{\mathsf{R}}\,3))$ as the "leaves". This term exposes the operational content of $\mathcal{U}[\![\mathit{Add\ (Val\ 1)\ (Val\ 2)}]\!]$. To wit, either an asynchronous exception occurs at *Add* or it doesn't (left-most $\mathit{merge}_{\mathsf{R}}$); or, an asynchronous exception occurs at *(Val 1)* or it doesn't (middle $\mathit{merge}_{\mathsf{R}}$); or, an asynchronous exception occurs at *(Val 2)* or it doesn't (right-most $\mathit{merge}_{\mathsf{R}}$); or, it returns the integer 3. Indeed, this single algebraic term captures all four evaluations from Figure 3. The term displayed in Example (†) can be recovered by projecting $\mathcal{U}[\![(\mathit{Add}\,(\mathit{Val}\,1)\,(\mathit{Val}\,2))]\!]$ to the E monad via *run*:

$$
\begin{aligned}
&\mathit{run}\,\mathcal{U}[\![(\mathit{Add}\,(\mathit{Val}\,1)\,(\mathit{Val}\,2))]\!]\\
&\qquad = \mathit{run}\,(\mathit{merge}_{\mathsf{R}}\,\{\,\mathit{merge}_{\mathsf{R}}\,\{\,\varphi,\ \mathit{throw}_{\mathsf{R}}\,\},\ \mathit{throw}_{\mathsf{R}}\,\})
\end{aligned}
$$

where $\varphi = \mathit{Pause}(\eta_{\mathsf{E}}(\mathit{merge}_{\mathsf{R}}\{\mathit{Pause}(\eta_{\mathsf{E}}(\eta_{\mathsf{R}}\,3)),\mathit{throw}_{\mathsf{R}}\}))$

$$
\begin{aligned}
&\{\text{thm 2}\} \quad = \mathit{merge}_{\mathsf{E}}\,\{\,\mathit{run}\,(\mathit{merge}_{\mathsf{R}}\,\{\,\varphi,\ \mathit{throw}_{\mathsf{R}}\,\},\ \mathit{run}\ \mathit{throw}_{\mathsf{R}}\,\})\\
&\{\text{thms 2,5}\} = \mathit{merge}_{\mathsf{E}}\,\{\,\mathit{merge}_{\mathsf{E}}\,\{\,\mathit{run}\,\varphi,\ \mathit{run}\ \mathit{throw}_{\mathsf{R}}\,\},\ \mathit{throw}_{\mathsf{E}}\,\})
\end{aligned}
$$

By repeated applications of Theorems 2 and 5, the definition of *run*, and left unit monad law, $\mathit{run}\,\varphi\ =\ \mathit{merge}_{\mathsf{E}}\{\eta_{\mathsf{E}}3,\mathit{throw}_{\mathsf{E}}\}$. Continuing:

$$
\{\text{thm 5}\} \quad = \mathit{merge}_{\mathsf{E}}\,\{\,\mathit{merge}_{\mathsf{E}}\,\{\,\mathit{merge}_{\mathsf{E}}\,\{\,\eta_{\mathsf{E}}\,3,\ \mathit{throw}_{\mathsf{E}}\,\}\,,\ \mathit{throw}_{\mathsf{E}}\,\},\ \mathit{throw}_{\mathsf{E}}\,\})
$$

Theorem 6 establishes an equivalence between Hutton's natural semantics (Figure 2) and the MMAE semantics in Figure 4. Due to the presence of non-determinism in both semantic specifications, there are more than one possible outcome for any expression. The theorem states that, for any expression e, a particular outcome (i.e., $(\mathit{Ok}\ v)$ or *Error*) may occur in the set of all possible outcomes for e (i.e., $\mathit{run}\,(\mathcal{U}[\![e]\!])$ or $\mathit{run}\,(\mathcal{B}[\![e]\!])$) if, and only if, the corresponding term (respectively, $(\mathit{Val}\ v)$ or *Throw*) can be reached via the natural semantics. The proof of Theorem 6 is straightforward and long, so rather than including it here, it has been made available online [8].

Theorem 6 (Semantic Equivalence). *For any expression e:*

$$
\begin{array}{cc}
(\mathit{Ok}\ v) \in \mathit{run}(\mathcal{U}[\![e]\!]) \text{ iff } e \Downarrow^{U} (\mathit{Val}\ v) & (\mathit{Ok}\ v) \in \mathit{run}(\mathcal{B}[\![e]\!]) \text{ iff } e \Downarrow^{B} (\mathit{Val}\ v)\\
\mathit{Error} \in \mathit{run}(\mathcal{U}[\![e]\!]) \text{ iff } e \Downarrow^{U} \mathit{Throw} & \mathit{Error} \in \mathit{run}(\mathcal{B}[\![e]\!]) \text{ iff } e \Downarrow^{B} \mathit{Throw}
\end{array}
$$

5 The MMAE as a Programming Model

This section demonstrates the use of the MMAE in the functional programming of applications with asynchronous behaviors. In particular, the extension of synchronous concurrent kernels with asynchronous, interrupt-driven features is described. The approach elaborated here develops along the same lines as modular interpreters [15] and compilers [11]: an existing specification is extended with a "building block" consisting of a monadic "module" and related operators. By applying the interrupts building block to a kernel without interrupts (middle, Figure 1), a kernel with interrupt-driven input is produced with virtually no other changes.

Recent research has demonstrated how kernels with a broad range of OS behaviors (e.g., message-passing, synchronization, forking, etc.) may be formulated in terms of resumption monads [9]. This section outlines the extension of such resumption-monadic kernels with asynchronous behaviors. The presentation is maintained at a high-level in order to expose the simplicity of the asynchronous extension. Interested readers may refer to the code base [8] or to Harrison [9] for the more details concerning kernel construction. The presentation here will frequently appeal to the reader's intuition in order to remain at a high-level.

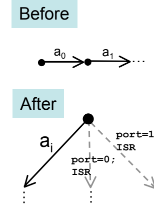
The kernel and its enhanced functionality are described in Figure 1. The enhanced kernel is kept as simple as possible. It contains no mechanism for blocking interrupts and there is only one interrupt service routine (ISR) connecting the kernel to the input port. Such enhancements are, however, quite simple to make. Both kernels (Figure 1, middle and right) support synchronous message-passing, maintaining a message queue, msgQ, to hold messages "in-flight". The extended kernel (Figure 1, right) adds an interrupt-driven serial-to-parallel input port. When a bit is ready at the port, an interrupt is thrown, causing the ISR to run. This ISR inserts the new bit into its local bit queue, bitQ. If there are eight bits in bitQ, then it assembles them into a byte and inserts this new datagram into msgQ, thereby allowing user threads to receive messages through the port. Pseudocode for the ISR is:

```
insertQ(port,bitQ);
if (length(bitQ) > 7) {
     make_datagram(x);
     insertQ(x,msgQ);
}
```

Here, `port` is the bit register in the port that holds the most recent input bit.

A resumption-monadic kernel is an $\mathsf{R}\,()$-valued function that takes as input a list of ready threads. In other words, $\mathit{kernel}\,\langle\text{ready}\rangle : \mathsf{R}\,()$, where $\langle\text{ready}\rangle$ is a Haskell representation of the thread list. The output of the kernel is, then, a (possibly infinite) R-computation consisting of a sequence of atomic actions a_i executed in succession as shown in the inset figure (top, Before). These actions are drawn from user threads in ⟨ready⟩ that have been woven together into a schedule by the kernel. The notion of computation associated with R will also be changed

as outlined in Section 5.1. The key to "turning on" asynchronous interrupts is to view each break-point "•" in the inset figure (top, Before) as a branching point. This branch contains three possible histories (inset figure (bottom, After)): one history in which the scheduled atom a_i executes, another where the ISR executes with the input bit 0, and a third where the ISR executes with the input bit 1. The branching can be achieved as in Section 4 with the use of an appropriate *merge* operator. The operation of the enhanced kernel is a computation that elaborates a tree.



There are only two changes necessary to introduce asynchronous interrupts to the synchronous kernel. The first is simply to refine the monad by incorporating non-determinism and state for the port device into the monad R. The second modification is to apply the appropriate *resumption map* to ($kernel\ \langle \text{ready} \rangle$). Resumption maps arise from the structure of R in an analogous manner to the way the familiar list function, $map : (a \to b) \to [a] \to [b]$, operates over lists. The resumption map used here is discussed below in Section 5.2.

5.1 Monad Hierarchy

Two monads associated with the kernel are the kernel monad, K, and the concurrency monad R. K encapsulates the imperative aspects of the system. These are kernel-level operations that update system data structures (*Sys*) and user-level operations that update the user-level storage (*Sto*). K is constructed with the application of two state monad transformers to the identity monad, Id.

$$\mathsf{K} = StateT\ Sys\ (StateT\ Sto\ \mathsf{Id})$$

The concurrency monad R is then built on top of K with an application of the resumption monad transformer [21]:

$$\begin{aligned} &\mathsf{R} &&= \mathsf{ResT}\ \ \mathsf{K} \\ &\mathbf{data}\ \mathsf{ResT}\ \mathsf{m}\ a &&= Done\ a \mid Pause\ (\mathsf{m}\ (ResT\ \mathsf{m}\ a)) \end{aligned}$$

Note that this monad transformer provides an alternative means of defining the R monad from Section 4: $\mathsf{R}\ =\ \mathsf{ResT}\ \mathsf{E}$.

5.2 Resumption Maps

There is a close analogy between computations in R and lists and streams. A non-trivial computation in R consists of a (possibly infinite) sequence of atomic actions in K separated by *Pause* constructors. Definition 7 specifies a resumption map that takes a function h and an R-computation γ and transforms γ one atom at a time:

Definition 7 (Map on R).

$$\begin{aligned} &map_{\mathsf{R}}\ :\ (\mathsf{R}\ a \to \mathsf{K}(\mathsf{R}\ a)) \to \mathsf{R}\ a \to \mathsf{R}\ a \\ &map_{\mathsf{R}}\ h\ (Done\ v)\ \ = Done\ v \\ &map_{\mathsf{R}}\ h\ (Pause\ \varphi) = Pause\ (h\ (Pause\ \varphi) \star_{\mathsf{K}} (\eta_{\mathsf{K}} \circ map_{\mathsf{R}}\ h)) \end{aligned}$$

5.3 Adding Asynchronicity in Two Steps

Now, the stage is set to add asynchronicity to the synchronous kernel. The first step provides additional computational resources to K and the second uses these resources to "turn on" asynchronicity.

First Step: Adding Non-determinism and a Device to K. The first change to the synchronous kernel is to add non-determinism and storage for the port device to the K and R monads. This is accomplished by replacing Id with N in the definition of K and with another application of the state monad transformer:

$$
\begin{aligned}
\mathsf{K} &= \mathit{StateT}\,\mathit{Dev}\,(\mathit{StateT}\,\mathit{Sys}\,(\mathit{StateT}\,\mathit{Sto}\,\mathsf{N}))\\
\mathit{Dev} &= [\mathit{Bit}] \times \mathit{Bit}
\end{aligned}
$$

The N is defined exactly as in Section 4. A device state is a pair, $(\mathit{bitQ}, \mathit{new})$, consisting of a queue of bits, bitQ, and a single bit denoting the current input to the port. The K monad has separate operators for reading and writing the *Dev*, *Sys* and *Sto* states along with a merge operation, $\mathit{merge}_{\mathsf{K}} : \mathcal{P}_{\mathrm{fin}}(\mathsf{K}\ a) \rightarrow \mathsf{K}\ a$, that is defined in terms of $\mathit{merge}_{\mathsf{N}}$. Note also that R is also affected by this refinement of K, but that the text of its definition does not (i.e., it is still the case that $\mathsf{R} = \mathsf{ResT}\ \mathsf{K}$). Details may be found in the code base [8].

Second Step: Asynchronicity via Resumption Mapping. First, a resumption mapping is defined that creates a branch with three possible histories.

$$
\begin{aligned}
&\mathit{branches} : \mathsf{R}\,a \rightarrow \mathsf{K}(\mathsf{R}\,\mathsf{a})\\
&\mathit{branches}\ (\mathit{Pause}\ \varphi) = \mathit{merge}_{\mathsf{K}} \left\{ \begin{array}{l} \varphi, \\ (\texttt{port}{=}0\ ;\texttt{ISR}) >\!\!>_{\mathsf{K}} \varphi, \\ (\texttt{port}{=}1\ ;\texttt{ISR}) >\!\!>_{\mathsf{K}} \varphi\ \} \end{array} \right\} \begin{array}{l} (1)\\ (2)\\ (3) \end{array}
\end{aligned}
$$

Here, $(\texttt{port}{=}0; \texttt{ISR})$ represents a K-computation that performs the indicated actions. History (1) is one in which no interrupt occurs. In history (2), an interrupt occurs when the input bit of the port is set to 0 and the ISR is then executed. Finally, the interrupted thread, φ, is executed. History (3) is analogous to (2).

Flipping the Port Off and On. It is now a simple matter to turn the port off and on. To operate the kernel with the port off, execute: *kernel* ⟨ready⟩. To operate the kernel with the port on, execute: $\mathit{map}_{\mathsf{R}}$ *branches* (*kernel* ⟨ready⟩). The two sample system runs from Figure 1 are repeated below. There are two threads, a producer and a consumer. The producer thread broadcasts integer messages starting at 1001 and incrementing successively with each broadcast. The consumer thread consumes these messages. When either thread performs a broadcast or receive, the instrumented kernel prints it out. Also, when the ISR

creates a new datagram, it also announces the fact.

Port off: *kernel* ⟨ready⟩	Port on: map_{R} *branches* (*kernel* ⟨ready⟩)

```
Haskell> producer_consumer        Haskell> producer_consumer
broadcasting  1001                broadcasting  1001
broadcasting  1002                   new datagram: 179
         receiving  1001          broadcasting  1002
broadcasting  1003                         receiving  179
         receiving  1002 ...         new datagram: 204 ...
```

6 Related Work

Implementations of threaded applications use monads to structure the threaded code. The Haskell libraries for concurrency use IO level primitives to provide a simple and robust user interface [22]. There have been other threading implementations that use other monads, including [13], which uses a continuation passing style monad, giving very high numbers of effectively concurrent threads.

There have been a number of efforts to model the concurrency provided by the Haskell IO monad. One successful effort is reported by Swierstra and Altenkirch [26], where they model the concurrency inside the Haskell IO monad using a small stepping scheduler combined with the QuickCheck framework [3] to provide random interleaving of pseudo-threads. In Dowse and Butterfield [5], an operational model of shared state, input/output and deterministic concurrency is provided. One fundamental difference between these models and our work is one granularity. Both these models assume each computation can not be interrupted, and threads only are scheduled at the IO interaction points. Our model allows interrupts to happen inside computation, capturing the pre-emptive implementations of concurrency described in [16, 23], and provided by the Glasgow Haskell compiler.

The language modeled with MMAE in Section 4 was adopted from recent research on the operational semantics of asynchronous interrupts [12]. That research also identified an error within a published operational semantics for interrupts in Haskell [16]. Morris and Tyrrell [19] offer a comprehensive mathematical theory of nondeterminacy, encompassing both its demonic and angelic forms. Their approach adds operators for angelic and demonic choice to each type in a specification or programming language. They present refinement calculus axioms demonic and angelic nondeterminacy and define a domain model to demonstrate the soundness of these axioms. One open question is whether there exists common ground between their work and the monadic approach presented here. In particular, if the domain theoretic constructions underlying Morris and Tyrrell's semantics may be expressed usefully as monads.

7 Future Work & Conclusions

There has been considerable interest of late in using monads to structure system software [13, 7], model it formally [9], and to enforce and verify its security [10].

None of this previous research contains an integrated model of asynchronous behavior and the MMAE was developed as a means of rectifying this situation. The present work is part of an ongoing effort to use monads as an organizing principle for formally specifying kernels. Currently, the direct compilation of monadic kernel designs to both stock and special purpose hardware is being investigated. This research agenda considers these monadic specifications as a source language for semantics-directed synthesis of high-assurance kernels. The ultimate goal of this agenda is to produce high-confidence systems automatically.

What possible applications might our specification methodology have? We have shown is that it is possible to take a MMS constructed executable specification, add a non-deterministic monad giving an accurate monadic based semantics for interrupts. Here we briefly sketch a possible way of using our semantics in practical applications.

Non-determinism is used to model *all* possible outcomes. In an implementation, a lower-level model could replace the use of non-determinism with the exception monad. The exception monad would be used to communicate an interrupt in exactly the same way it would be used to communicate an exception. The fork primitive in our semantics would be implemented in a lower-level model using a simple poll of an interrupt flag. Specifically, in our kernel example, we might use this MMS for K in the lower-level model.

$$\mathsf{K} \;=\; \mathit{StateT}\ \mathit{Dev}\ (\mathit{StateT}\ \mathit{Sys}\ (\mathit{StateT}\ \mathit{Sto}\ \mathit{Maybe}))$$
— K = Dev + Sys + User + Exception

This gives us a basis for an efficient implementation from our monadic primitives. We use the code locations that perform interrupt polling in the lower-level model as *safe-points*. If an interrupt request is made of a specific thread, this thread can be single-stepped to any safe-point, a common implementation technique for interruptible code.

Thus as an implementation path we have our high-level model using non-determinism, a low-level model using exceptions to simulate interrupts, and an possible implementation with precisely specified safely interruptible points that can be implemented using direct hardware support. This chaining of specifications toward implementation is a long standing problem in building non-deterministic systems, and this research provides an important step toward a more formal methodology of software development of such systems.

The MMAE confronts one of the "impurities" of the infamous "Awkward Squad" [22]: language features considered difficult to accommodate within a pure, functional setting—concurrency, state, and input/output. Most of these impurities have been handled individually via various monadic constructions (consider the manifestly incomplete list [17, 24, 21]) with the exception of asynchronous exceptions. 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 [14, 6]. While it is not the intention of

the current work to model either the Haskell *IO* monad or Concurrent Haskell, it is believed that the techniques and structures presented here provide some important building blocks for such models.

References

1. J.W. de Bakker. *Mathematical Theory of Program Correctness.* International Series in Computer Science. Prentice-Hall, 1980.
2. K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. A. Henzinger, and J. Palsberg. Stack size analysis for interrupt-driven programs. *Inf. Comput.*, 194(2):144–174, 2004.
3. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of International Conference on Functional Programming (ICFP).* ACM SIGPLAN, 2000.
4. E. W. Dijkstra. My recollections of operating system design. *SIGOPS Oper. Syst. Rev.*, 39(2):4–40, 2005.
5. M. Dowse and A. Butterfield. Modelling deterministic concurrent i/o. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 148–159, New York, NY, USA, 2006. ACM.
6. D. Espinosa. *Semantic Lego.* PhD thesis, Columbia University, 1995.
7. T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in haskell. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP05)*, pages 116–128, New York, NY, USA, 2005. ACM Press.
8. W. Harrison. The Asynchronous Exceptions As An Effect Codebase. Available from `www.cs.missouri.edu/~harrisonwl/AsynchronousExceptions`.
9. W. Harrison. The essence of multitasking. In *11th International Conference on Algebraic Methodology and Software Technology (AMAST 2006)*, pages 158–172, July 2006.
10. W. Harrison and J. Hook. Achieving information flow security through monadic control of effects. Invited submission to: *Journal of Computer Security*, 2008. 46 pages. Accepted for Publication.
11. W. Harrison and S. Kamin. Metacomputation-based compiler architecture. In *5th International Conference on the Mathematics of Program Construction, Ponte de Lima, Portugal*, volume 1837 of *Lecture Notes in Computer Science*, pages 213–229. Springer-Verlag, 2000.
12. G. Hutton and J. Wright. What is the Meaning of These Constant Interruptions? To appear in the Journal of Functional Programming, 2007.
13. P. Li and S. Zdancewic. 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*, pages 189–199, New York, NY, USA, 2007. ACM Press.
14. S. Liang. *Modular Monadic Semantics and Compilation.* PhD thesis, Yale University, 1998.
15. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 333–343. ACM Press, 1995.

16. S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 274–285, 2001.
17. E. Moggi. An Abstract View of Programming Languages. Technical Report ECS-LFCS-90-113, Department of Computer Science, Edinburgh University, 1990.
18. E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
19. J. M. Morris and M. Tyrrell. Terms with unbounded demonic and angelic nondeterminacy. *Sci. Comput. Program.*, 65(2):159–172, 2007.
20. J. Palsberg and D. Ma. A typed interrupt calculus. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 291–310, London, UK, 2002. Springer-Verlag.
21. N. S. Papaspyrou. A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. In *Proceedings of the 3rd Panhellenic Logic Symposium*, 2001. Expanded version available as a tech, report from the author by request.
22. S. Peyton Jones. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell. In *Engineering Theories of Software Construction*, volume III 180 of *NATO Science Series*, pages 47–96. IOS Press, 2000.
23. S. Peyton Jones, A. Reid, A. Hoare, S. Marlow, and Fergus Henderson. A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 25–36, May 1999.
24. S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 71–84. ACM Press, January 1993.
25. D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Boston, 1986.
26. W. Swierstra and T. Altenkirch. Beauty in the beast. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 25–36, New York, NY, USA, 2007. ACM.
27. A. Tolmach and S. Antoy. A monadic semantics for core curry. In *Proceedings of the 12th International Workshop on Functional and (Constraint) Logic Programming*, June 2003.
28. P. Wadler. The essence of functional programming. In *Proceedings of the 19th Symposium on Principles of Programming Languages (POPL)*, pages 1–14. ACM Press, 1992.