Achieving Information Flow Security Through Monadic Control of Effects

William L. Harrison
Department of Computer Science
University of Missouri
Columbia, Missouri, USA
harrisonwl@missouri.edu

James Hook
Department of Computer Science
Portland State University
Portland, Oregon, USA
hook@cs.pdx.edu

January 8, 2009

Abstract

This paper advocates a novel approach to the construction of secure software: controlling information flow and maintaining integrity via monadic encapsulation of effects. This approach is *constructive*, relying on properties of monads and monad transformers to build, verify, and extend secure software systems. We illustrate this approach by construction of abstract operating systems called *separation kernels*. Starting from a mathematical model of shared-state concurrency based on monads of resumptions and state, we outline the development by stepwise refinements of separation kernels supporting Unix-like system calls, interdomain communication, and a formally verified security policy (domain separation). Because monads may be easily and safely represented within any pure, higher-order, typed functional language, the resulting system models may be directly realized within a language such as Haskell.

1 Introduction

Confidentiality and integrity concerns within the setting of shared-state concurrency are primarily addressed by controlling interference and interaction between threads. Several investigators have attempted to achieve control of interference through language mechanisms that systematically separate information. Most of these approaches have been security-specific extensions to type systems for existing languages [11, 69, 28, 68, 57, 52].

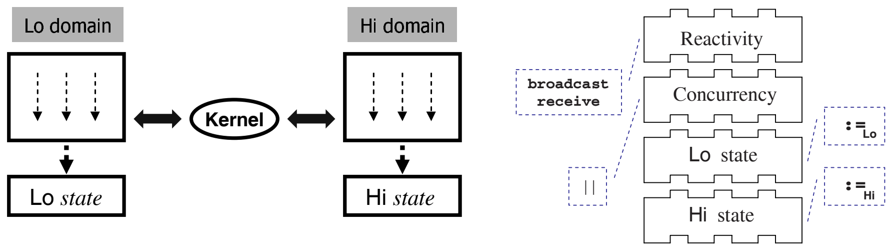

Figure 1: (Left) *Separation Kernel*: Threads within each domain can only access their own state, and all interdomain communication is mediated by the kernel. The kernel enforces a "no write down" security policy. (Right) *Layering Monads for Separation*: Combining fine control of stateful effects with concurrency into Layered Monads have important properties "by construction".

In this investigation we take a different approach. We do not use a domain-specific extension to the type system. We use a standard pure functional language, with its existing type system, as our base language. Within that language and type system we characterize the effects that are at play in an operating system kernel using the semantic technique of monadic encoding of effects. Most importantly, we construct the effect model in a modular manner using constructions called monad transformers [46, 36]. This modularity clearly distinguishes within the type system those facets of the global effect system on which a program fragment acts. This permits the expression of a kernel that has provable global separation policies, while still enabling the expression of policy functions in specific, identifiable contexts in which otherwise separated effects are allowed to interfere.

The development proceeds by developing three model kernels, the complete code of which may be downloaded from our website [23]. These kernels build on one another. The first provides the reference point for thread behavior in isolation—the model of integrity of thread execution. The second and third kernels provide more sophisticated concurrency and communication primitives with sufficient power to be vulnerable to exploitation if separation is not achieved.

Precise Control of Effects. Monads support an "abstract data type approach" to language definition [12], capturing distinct computational paradigms as algebras. A helpful metaphor is that a monad is a programming language with (at least) sequencing (`;`) and "no-op" (`skip`) constructs where (`;`) is associative and has `skip` as its right and left unit. Monad "languages" may contain other language features corresponding to their computational paradigms: the state monad, for example, has assignment (`:=`) and resumption monads [53] define concurrent execution (`||`) and, in some formulations, "reactive" programming features [46] such as message passing, synchronization, etc. Monad transformers [46, 36] are monad language "constructors" that add new features to a monad language with each monad transformer application; such modularly-constructed monads are referred to as *layered* monads. This metaphor is not a precise characterization of monads or monadic semantics; it is imprecise in the following sense. The metaphor is stated in terms of a statement language, like Algol,

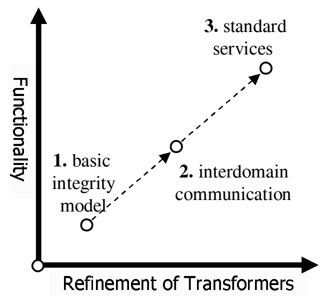

Figure 2: *Scalability*: Kernel specifications based on fine control of effects achieve a significant level of scalability in two important respects: they are easily extended and modified and the impact of such extensions on the security verification is minimized.

in which statements are executed for their effects, communicating values only through effects (such as updating the store). In the general case monads allow the description of expression languages, such as C, in which subexpressions may contribute both a value and an effect.

Monad transformers add new features while preserving the behavior of existing ones; this is the essence of modularity and extensibility in interpreters and compilers based on layered monads [12, 36, 27]. Less well-known is that "layering" effects controls the *interaction* of features from different layers. These are "free" properties in the sense that they come cost-free as a result of structuring by monad transformers. One such structural property of layered state monads is the commutation of imperative operations from separate layers; that is, if h and l are imperative operations on different layers, then $h; l = l; h$. From the point of view of integrity and information flow security, this relationship precisely captures operation-level noninterference (called *atomic noninterference* here) and provides a flexible foundation for the development of software with information flow security.

We demonstrate this approach through the construction of abstract operating systems called *separation kernels*. Separation kernels [64, 63] enforce a noninterference-based security property by partitioning the state into separate user spaces or "domains" (see Figure 1, left); the kernel mediates all interdomain communication, thereby enforcing its security policy. The state partitioning is easily achieved through multiple applications of the state monad transformer, and this, when combined with appropriate models of concurrency, provides all the raw material necessary for building separation kernels (as shown in Figure 1, right). However, layering effects is more than an implementation technique: properties arise from the underlying structure of layered monads that prove useful in verifying the integrity and security of such kernels—separation is, in a sense, "built-in" to layered state monads.

This approach emphasizes scalability; Figure 2 illustrates the refinement process of the three separation kernels, and each step along that arrow marks an extension to kernel functionality. The kernel at point (1), in which threads are executable in complete isolation on separate domains, is not interesting from an information security point of view in itself. However, it does provide basic separation entirely as a consequence of its layered monadic structure and serves as a foundation for the other two kernels. The kernel at point (2) extends point (1) with *inter*-domain functionality: message-passing obeying a "no-write-down" security policy. Point (3) extends (2) with *intra*-domain functionality: a Unix-like system call that duplicates the calling process. Point (3) illustrates the scalability of the approach; its new functionality, being irrelevant to security, has little impact on security verification. For the sake of simplicity, we assume there are exactly two user domains, Hi and Lo, but all of our results generalize easily to n user domains and security lattices. Monad transformers are well-known tools for writing modular and extensible programs [35, 24]. Less frequently recognized is their value for formal specification and verification; the impact of the kernel refinements on the verification is minimal.

The remainder of this section surveys related work. Section 2 summarizes the background on separation kernels and formulates three process languages corresponding to points (1) through (3) in Figure 2. Section 3 begins with an overview of monads and monad transformers, then develops the theory of layered state monads describing how it addresses integrity and information flow concerns. Section 4 illustrates how layered state monads express the basic model of integrity when combined with a sequential theory of concurrency based on resumption monads; this section begins with an overview of resumption-based concurrency and ends with the formulation of separation security in this setting—what we call *take separation*. Based on a refinement to the concurrency model allowing expression of reactive programs, Section 5 explores the implementation and verification of interdomain extensions to the basic model of integrity; the section begins with a description of reactivity in monadic form. Section 6 considers intradomain kernel extension and verification. Finally, Section 7 summarizes the present work and outlines future directions.

1.1 Related Work

Many techniques in language-based security [69, 28, 68, 57, 52] are *proscriptive*, meaning that they rely on sophisticated type systems to reject programs with security flaws. Other models [17, 41, 76, 42, 64] are *extensional* in that, broadly speaking, they characterize security properties in terms of subsets of possible system executions. The approach to language-based security advocated here is, in contrast to both of these, *constructive*, relying on structural properties of monads and monad transformers to build, verify, and extend secure software systems.

A closely related approach to this work is that of Joshi and Leino [31] in which relational semantics is applied to the control of information flow. Their approach also involves partitioning the state variables of a concurrent, guarded command language (similar to the Point 1 language from Figure 2) according to security levels. The definition of security is similar to take-separation; a program s is secure means that $hh\,;s\,;hh = s\,;hh$, where hh (called "havoc on h") sets the high-security state to an arbitrary value. Here, hh plays a similar rôle to $mask_{H}$ in that it nullifies the effects of s on the high state. A drawback of their approach (according to the authors Joshi and Leino [31]) is that their definition of security requires careful fixed-point calculations in the semantics of iteration and recursion. Structuring our system specifications by resumptions avoids this issue in that proofs of take-separation resemble operational techniques (e.g., bisimulation) more than purely denotational techniques (e.g., fixed point induction).

Abadi, et al., [1] formulate the *dependency core calculus* (DCC) as an extension of Moggi's computational lambda calculus [47]. They show many notions of program dependency (from program slicing to noninterference) may be recast in terms of DCC. For example, noninterference within a single-threaded while language (a fragment of the Smith-Volpano calculus [69]) is characterized via a translation into DCC. An encoding of DCC into system F is presented in Tse, et al. [72]. In [1], the Smith-Volpano fragment has a conventional store passing semantics of state, except that the denotational model of DCC (like those in [44, 28]) uses parametricity [60] to restrict the store transformers to those respecting the security discipline. The state monad transformer provides a means of constructing such store transformer functions (namely, g and u) as evidenced by Theorems 1-3; this is central our approach.

Crary et al. [10] consider a logical characterization of information flow security that, like DCC, incorporates Moggi's computational lambda calculus at its core. With their approach, monads are, in effect, logical modalities signifying the potential presence of effects at a security level. In contrast, the current work is more semantic and model-theoretic than Crary's logical and type-theoretic approach, relying as it does on structural properties of layered monads to construct secure systems. An interesting open line of inquiry would explore this comparison and make the relationship between the semantic and logical views precise.

There has been a growing emphasis on such *language-based* techniques for information flow security [69, 28, 68, 56, 57]; please see Sabelfeld and Myers [65] for an excellent survey of this work. The chief strength of this type-based approach is that the well-typedness of terms can be checked statically and usually automatically, yielding high assurance at low cost. Unfortunately, this type-based approach is not as general as one might wish: first, there will be programs that are secure but will be rejected by the type system due to lack of precision, and second, there will be programs that have information flow leaks that we want to allow (e.g., a declassification program [77]) that would be rejected by the type system.

Certain desirable behaviors may require weakening the Goguen-Meseguer notion of non-interference, with declassification being a notable example. Abstract interpretation has been applied to define a notion of non-interference parametrized relative to what an attacker can observe [14]; this approach—called *abstract non-interference*—has been extended to accommodate declassification [15]. What an attacker may observe is characterized by an abstraction function with this approach; this abstraction function limits the amount of information observable by an attacker. An interesting open problem is how well the current approach accommodates declassification and other relaxed formulations of non-interference [34]. One possible approach would be to integrate abstract non-interference with the current one. In this scenario, the kernel implements system services providing information downgrading with abstraction functions. Another approach currently being investigated views declassification as a system event that changes the shape of the store by moving a downgraded memory cell from high-security storage to low-security storage. With this approach, both before and after the declassification event the system would have to obey the same security specification developed here.

An *implicit flow* is a covert channel arising through branching or iterative control constructs in programming languages [11, 65]. For example, the following program contains an implicit flow of information from variable x to variable y due to the if-then-else branching construct:

```
if x==0 then y:=1 else y:=2
```

There is clearly an information flow as the value of y depends on x, but this flow is implicit because there is no explicit assignment of y from x. Implicit flows can also arise via iteration due to the potential of non-termination. Such prohibited flows cannot arise within the separation kernels developed here. Implicit flows arising via branching constructs cannot occur as a consequence of language design: there are simply no inter-domain branching constructs within the *Event* languages described below in Section 2. Intradomain branching constructs can be easily added (e.g., see Papaspyrou [53]) and these have no danger of introducing implicit flows as all storage within a domain is at the same security level. Non-terminating computations certainly occur within our setting as well (indeed, we assume that threads are all non-terminating), but only in a interruptible form that removes the possibility of an implicit flow.

Separation logic [50, 61] incorporates the notion of disjoint regions of state into the specification logic of Reynolds [59]; the fine-grained distinctions concerning storage allow for more modular reasoning about imperative programs. There is clearly a connection between the storage model of separation logic [50] and the layering of stateful effects in this work, although we have not, as yet, explored the formal relationship.

Moggi showed that most known semantic effects could be naturally expressed monadically, and, in particular, how a sequential theory of concurrency could be expressed in the resumption monad [46]. The formulation of basic resumptions in terms of monad transformers used here is that of Papaspyrou [53]; the reactive resumption monad transformer originates with Moggi [46]. Concurrency may also be modeled by the continuation-passing monad [9]; resumptions can be viewed as a disciplined use of continuations allowing for simpler reasoning about our system. Resumptions, being computational traces, lend themselves to an observational equivalence style of reasoning, as evidenced by the security verification outlined in the previous section.

Monads of resumptions are a powerful and expressive abstraction from concurrency theory [53] that were previously found only in the literature of programming language semantics. Recent research [25] demonstrates that they can play a dual rôle as a programming tool for concurrent applications: a wide variety of typical operating systems behaviors may be specified in terms of resumption monads. The separation kernel described in this article makes critical use of the expressive power of resumption monads as both a formal basis for concurrency and for controlling information flow.

There have been many previous attempts to develop secure OS kernels: PSOS [48], KSOS [40], UCLA Secure Unix [75], KIT [6], and EROS [67] among many others. There has also been work using functional languages to develop high confidence system software: the Fox project at CMU [22] is a case in point of how typed functional languages can be used to build reliable system software (e.g., network protocols, active networks); the Ensemble project at Cornell [8] uses a functional language to build high performance networking software; and the Switchware project [2] at the University of Pennsylvania is developing an active network in which a key part of their system is the use of a typed functional language. The Programatica project at Portland State University [58] is working to develop and formally verify *OSKer* (Oregon Separation Kernel), a kernel for MLS applications [21]. To formally verify security properties of such a system is a formidable task, and the current work arose as an exemplary design for *OSKer*.

There has been increased interest of late in the formal construction and verification of software systems with verified separation properties. Martin et al. [39] describe formal development of a separation kernel called MASK (Mathematically Analyzed Separation Kernel) using the SPECWARE methodology [70]. Greve et al. [19, 18] consider the specification and analysis of a formal separation policy within the ACL-2 theorem proving system [32]. Heitmeyer et al. [29] relates the verification of a data separation property for small embedded separation kernel written in C. The verification combines traditional Floyd-Hoare style program proofs of the kernel code along with analysis of the security policy using the TAME library [3] of the PVS theorem proving system [51]. In contrast to the aforementioned research, the current approach uses a single mathematical structure—the modular monad—as an organizing principle for the design and verification of secure software. Monads are, for this work, a design tool, an approach to rapid prototyping, and a basis for formal verification. Also in contrast to the aforementioned work, the kernel verification described here is "by hand." Huffman and Matthews [30] report the application of the Isabelle theorem proving system [49] to reasoning about monads and, in particular, to concurrency monads integral to the separation kernel synthesis. Their work forms a basis for mechanizing the verification of the separation kernels as described here.

2 Separation Kernels

A *separation kernel* enforces process isolation by partitioning the state into separate user spaces (Rushby [64, 63] calls these "colours"), allowing reasoning about the processes in each user space as if they were physically distributed. The security property—*separation*—is then specified using finite-state machines, and separation (i.e., that differently-colored processes do not interfere) is characterized in terms of traces arising from executions of these machines.

A separation kernel [64, 63], $M = (S, I, O, \mathit{next}, \mathit{inp}, \mathit{out})$, is an abstract machine formally characterizing a multi-user operating system. Here, S, I, and O are sets of states, inputs, and outputs, respectively, and there are functions $\mathit{next} : S \to S$, $\mathit{inp} : I \to S$, and $\mathit{out} : S \to O$ to represent state transition and the observable input and output of M. The functions next, inp, and out are total because each individual machine action is assumed to terminate. There are different user domains or "colours" $\{1, \ldots, m\}$ and the input and output sets are partitioned according to user domain: $I = I^1 \times \ldots \times I^m$ and $O = O^1 \times \ldots \times O^m$. A *computation* from initial input $i \in I$ is an infinite sequence $\langle s_0, s_1, \ldots \rangle$ such that $s_0 = \mathit{inp}(i)$ and $s_{j+1} = \mathit{next}(s_j)$ for all $0 \leq j$.

The behavior of processes in user domain c is *separable* from other user domains in M if, and only if, c's outputs depend only on its inputs. If this fails, then M allows interference between c and some other user domain and is considered insecure.

There are several functions defined on computations that allow this idea to be formalized. The function $res(i)$ maps out onto each state in a computation starting from input i: $res(i) = \langle out(s_0), out(s_1), \ldots \rangle$. Function $ext(c, x)$ projects[1] all of the c-coloured objects from x, so $ext(c, res(i))$ is the trace of all c-outputs in $res(i)$. Function $condense(s)$ removes all "stutters" from s: $condense(\langle 1, 2, 2, 2, 3 \rangle) = \langle 1, 2, 3 \rangle$. Stuttering may occur because the scheduler represented in $next$ is not completely fair, and allowing stuttering introduces the possibility of a timing channel [33] with which the separation property does not attempt to represent. The formal statement of separation security is:

$$\begin{aligned} &ext(c, i) = ext(c, j) \Rightarrow \\ &\quad condense(ext(c, res(i))) = condense(ext(c, res(j))) \end{aligned}$$

for all colours $c \in C$ and inputs $i, j \in I$. It requires that, on any user domain c and for any inputs i, j indistinguishable by c (i.e., $ext(c, i) = ext(c, j)$), c produces the same condensed output (i.e., $condense(ext(c, res(i))) = condense(ext(c, res(j)))$).

Separation (both in Rushby's formulation [64] and ours) confronts storage and legitimate (i.e., using system resources to transfer information) channels, but not covert or timing channels [33].

Process Languages for Separation Kernels. This section formulates an abstract syntax for separation kernel processes. Processes are infinite sequences of events; in BNF, this is: $Process = Event\,;Process$. It is straightforward to include finite (i.e., terminating) processes as well, but it suffices for our presentation to assume non-terminating, infinite processes. The $Event$ languages defined below contain simple expressions. Expressions are constants, the contents of a location, or a binary operation: $Exp = Int \mid Loc \mid Exp \odot Exp$.

Events are abstract machine instructions—they read from and write to locations and signal requests to the operating system. We have three event languages, each corresponding to a point in Figure 2 and is an extension of its predecessor:

— **1.** basic integrity:
$Event = Loc \texttt{:=} Exp$
— **2.** interdomain communication:
$Event = Loc \texttt{:=} Exp \mid \texttt{bcast}(Loc) \mid \texttt{recv}(Loc)$
— **3.** standard services:
$Event = Loc \texttt{:=} Exp \mid \texttt{bcast}(Loc) \mid \texttt{recv}(Loc) \mid \texttt{dupl}$

Each event language has a simple assignment statement, $l\texttt{:=}e$, that evaluates its right-hand side, $e \in Exp$, and stores it in the location, $l \in Loc$, on the left-hand side. The second and third event languages extend the first with broadcast and receive primitives: $\texttt{bcast}(l)$ and $\texttt{recv}(l)$. The event $\texttt{bcast}(l)$ broadcasts the contents of location l, while $\texttt{recv}(l)$ store a message, if available, in location l. These primitives introduce the possibility of interdomain communication; it is the responsibility of the separation kernel to ensure that a broadcast $\texttt{bcast}(l)$ event on a high security domain cannot be received

[1]The function $ext(c, x)$ is overloaded in the original work; x may be an input, output, or infinite sequence of inputs or outputs.

by a `recv`(l) event on a low security domain. The third language extends the first two with a new primitive, `dupl`, that produces a duplicate child process executing in the same address space.

None of these languages is "security conscious"—their syntax does not reflect security level or domain—and, therefore, the maintenance of integrity and security concerns is entirely the responsibility of the kernel. Note also that information flow security is non-trivial as the message passing primitives have the potential for insecure leaks. The `dupl` primitive has no impact on security or integrity concerns at all; it was included so that, later in this article, we may illustrate the negligible impact that such security-irrelevant features have on security verification due to the monadic encapsulation of effects.

3 Layered State Monads & Separation

This section considers the representation and construction of separated domains. The principle tool applied to these tasks is the *layered state monad* (defined below). It is shown how "by construction" properties of layered state monads give rise naturally to an algebraic foundation for domain separation. In Sections 4-6, a separation semantics for the process and event languages will be given partly in terms of layered state monads.

Monads and their uses in the denotational semantics of languages with effects are essential to this work, and we assume of necessity that the reader possesses familiarity with them. This section begins with a quick review of the basic concepts of monads and monad transformers [46, 35], and readers requiring more should consult the references for further background. Monads for concurrency—i.e., resumption monads—are described in Sections 4-6.

Monads are algebras just as groups or rings are algebras; that is, a monad is a type constructor (functor) with associated operators obeying certain equations. These equations—the "monad laws"—are defined below. There are several formulations of monads, and we use one familiar to functional programmers called the Kleisli formulation: a monad M is given by an eponymous type constructor M and the *unit* and *bind* operators, $\eta\ :\ a \rightarrow M\ a$ and $\star\ :\ M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$, respectively. The η and $\star$ are polymorphic, meaning that the variables a and b range over any type. In the context of a typed functional language like Haskell, type variables such as a and b range over any simple type (i.e., a base type like *Int* or a type constructed from other simple types with standard type constructors for products, functions, etc.). See below for further explanation of Haskell-style syntax. The (η) and $(\star)$ operators correspond to the "`skip`" and "`;`" constructs in the "monads as programming languages" metaphor from the introduction. Monads are typically extended with additional operators called *non-proper* morphisms; in monadic semantics for languages with effects, each language effect is modeled by one or more such additional operator.

As observed earlier, the metaphor comparing the (η) and $(\star)$ operators with "`skip`" and `;` is a restriction on the general formulation of monads. In the simplification for a statement language the types of (η) and $(\star)$ are specialized to (where $()$ is both the single element unit type and its single element):

$$\begin{array}{rcl} \eta & : & () \to M() \\ \star & : & M() \to (() \to M()) \to M() \end{array}$$

The isomorphism $X \cong () \to X$ and abbreviation, $Command = M\ ()$, yield:

$$\begin{array}{rcl} skip & : & Command \\ ; & : & Command \to Command \to Command \end{array}$$

Following Moggi [47], it is conventional within monadic semantics to view a monad as a formulation of a particular *notion of computation*. In this article, we consider three distinct, but interrelated, notions of computation formulated as monads. Specifically, the notions of computation manifested here are imperative computations, non-interactive concurrent computations, and interactive concurrent computations. The monadic formulation of imperative computation is given below in this section, while those for the non-interactive and interactive concurrent notions of computation are given in Section 4 and Section 5, respectively.

Also conventional within monadic semantics is to observe that a monad gives rise to a distinction between values and computations. As one would expect, for a type t, any object of type t is known as a $value$ of type t. For a monad M, a *computation of t in M* is simply an object of type Mt. Such an object may also be referred to as an M-*computation* or simply as a *computation* when M is clear from context. The metaphor here is that, within a computation of type t in M, M is a "black box" encapsulating some variety of computation that ultimately produces a value of type t.

Monads play a dual rôle here as a mathematical structure and as a programming abstraction—this duality supports both precise reasoning and executability. We represent the monadic constructions here in the pure functional language Haskell [54] although we would be equally justified in using categorical notation. The choice of Haskell is somewhat arbitrary as any higher-order functional programming language will do, and so we suppress details of Haskell's concrete syntax when they seem unnecessary to the presentation (in particular, instance declarations and class predicates in types). We also continue to use η and $\star$ for monadic unit and bind instead of Haskell's **return** and >>=. The Haskell code for these constructions is available online [23]. We follow the standard convention that (:) stands for "has type" and (::) for list concatenation (Haskell reverses this notation).

Defining a monad in Haskell typically consists of declaring a data type and an instance declaration of the built-in $Monad$ class [54]; note, however, that Haskell does not guarantee that members of $Monad$ obey the monad laws. All of the constructions presented here, however, produce monads [46, 36, 53]. The data type declaration defines the computational "raw materials" encapsulated by the monad. The identity monad I, containing no raw materials, is defined below. The identity monad is not of interest in itself, but it is used later as an "initial" monad for constructing monads with monad transformers.

$$\begin{array}{ll}
\mathbf{data}\, I\ a & = \ I\ a \\
deI\ (I\ x) & = \ x \\
\eta\ v & = \ I\ v \\
(I\ x) \star f & = \ f\ x
\end{array}$$

The state monad *St*. The state monad St adds a state type (Sto below) as "raw material". The state type Sto can, in general, be any type whatsoever. As it is used in this article, Sto is assumed to be a function type mapping locations to integer values, $Loc \rightarrow Int$. The internal structure of St is a function type, $Sto \rightarrow (a, Sto)$, taking an input store to a pair consisting of a value of type a and an output store. In this article, the value type a will be either $()$ or Sto, where $()$ is both the single element unit type and its single element. The unit computation, $\eta\ v$, is a state-passing computation that takes an input store, σ, and returns the pair, (v, σ).

The bind operator, $\star$, is a kind of application operator. In the definition of $\star$ below, the input store, σ, threaded first through x, which returns a pair, (y, σ'). The function f is applied to value y and then to output store σ using $deST$. The state monad St is declared as:

$$\begin{array}{ll}
\mathbf{data}\, St\ a & = \ ST\ (Sto \rightarrow (a, Sto)) \\
deST\ (ST\ x) & = \ x \\
\eta\ v & = \ ST\ (\lambda\sigma.\ (v, \sigma)) \\
(ST\ \ x) \star f & = \ ST(\lambda\sigma.\ \mathbf{let}\ (y, \sigma') =\ x\ \sigma\ \mathbf{in}\ deST\ (f\ y)\ \sigma')
\end{array}$$

The state monad has operators for reading the state, $g\ :\ St\ Sto$ (pronounced "get"), and writing the state, $u\ : (Sto {\rightarrow} Sto) \rightarrow St\ ()$ (pronounced "update"):

$$\begin{array}{ll}
g & = \ ST\ (\lambda\sigma.\ (\sigma, \sigma)) \\
u\ \delta & = \ ST\ (\lambda\sigma.\ ((), \delta\ \sigma))
\end{array}$$

The "null" bind operator, $(>>) : M\ a \rightarrow M\ b \rightarrow M\ b$, is useful when the result of $\star$'s first argument is ignored: $x >> y\ =\ x\ \star\ \lambda\ \_.\ y$.

To illustrate the state monad (for $Sto\ =\ Loc \rightarrow Int$) and its non-proper morphisms, consider the following location reading and incrementing operations defined in terms of g and u:

$$\begin{array}{ll}
getloc & :\ \ Loc \rightarrow St\ Int \\
getloc\ l & = (g \star \lambda\ \sigma.\ \eta\ (\sigma\ l)) \\
inc & :\ \ Loc \rightarrow St\ () \\
inc(x) & = getloc\ x \star \lambda v.\ u[x {\mapsto} v{+}1]
\end{array}$$

Here, $[x{\mapsto}v]$ is the update function ($\lambda\sigma.\mathbf{if}\ n{=}{=}x\ \mathbf{then}\ v\ \mathbf{else}\ \sigma n$). Using g, the operation, $getloc(l)$, first reads the current store, σ, and then returns the current contents of l, $\sigma\, l$, using η. To increment the contents of x, $inc(x)$ first reads its current contents, v, with $getloc$, and then updates location x to $v + 1$ using update u.

An Aside on Haskell Notation. There are several ways in which the notation used for monadic and semantic definitions throughout this article differs from the concrete syntax of Haskell [54]. Variables in Haskell are always written in lower case, and this includes the type variables representing monads in monad transformer definitions. The literature on monads [45, 47, 46] follows the standard convention of capitalizing the functor of a monad, that, in turn, follows the convention within category theory [37, 4] of capitalizing functors generally. Accordingly, the names of particular monads (e.g., *St* above) are capitalized in italic font. Monad transformers follow this convention as well with an additional capital *T* suffix to distinguish them from monads. Types in Haskell can include class predicates on type variables that reflect the use of overloading. Monad transformers, when implemented in Haskell, are sometimes encoded using the class system [36] because this can automate monad transformer application and "lifting" (described below) in some circumstances. The monadic definitions in this article are written in an explicit, mathematical style rather than in terms of the Haskell class system.

As a means of clarifying the Haskell terminology in this article, let us briefly compare the state monad as defined above along with its definitions in Standard ML [43] and category theory:

$$\begin{array}{ll}\mathbf{data}\ \mathit{St}\ a & = \mathit{ST}\ (\mathit{Sto} \rightarrow (a, \mathit{Sto}))\\ \mathbf{datatype}\ {}'a\ \mathit{st} & = \mathit{ST}\ \mathbf{of}\ (\mathit{Sto} \rightarrow ({}'a\ *\ \mathit{Sto}))\\ \mathit{St}\ A & = (A \times \mathit{Sto})^{\mathit{Sto}}\end{array}$$

In Haskell and ML, St is known as a *type constructor*: it takes a type argument (a or $'a$, respectively) and returns a new data type ($(St\ a)$ or $('a\ st)$, respectively). In each of these first two cases, the ST on the right-hand side of each is known as a *data constructor*; it takes a value of the appropriate type ($(Sto \rightarrow (a, Sto))$ or $(Sto \rightarrow ('a\ *\ Sto))$, respectively) and produces a value in the data type. The third definition represents the state monad as an endofunctor on a suitable category (i.e., one with products and exponentials). In each case, Sto is some type or object whose exact structure is typically unspecified. In the denotational semantics of imperative languages [71, 66], Sto would typically be an arrow type mapping locations to storable values (e.g., $Loc \rightarrow Int$).

The state monad transformer *StateT*. A monad transformer takes an existing monad and adds new computational "raw materials" to it [46, 12, 35]. The state monad transformer adds updatable storage to an existing monad. It takes two type parameters as input—the type constructor M representing an existing monad and a store type s—and from these creates a monad adding single-threaded s-passing to the computational raw material of M. The new monad $(StateT\ s\ M)$ is defined in terms of the bind and return of M:

$$\begin{array}{lll}\mathbf{data}\ \mathit{StateT}\ s\ M\ a & = & \mathit{ST}(s \rightarrow M(a, s))\\ \mathit{deST}\ (\mathit{ST}\ x) & = & x\\ \eta\ v & = & \mathit{ST}(\lambda\sigma.\ \eta_{M}\ (v, \sigma))\\ (\mathit{ST}\ x) \star f & = & \mathit{ST}(\lambda\sigma.\ (x\ \sigma) \star_{M} \lambda\ (y, \sigma').\ \mathit{deST}\ (f\ y)\ \sigma')\\ \mathit{lift}\ x & = & \mathit{ST}(\lambda\sigma.\ x \star_{M} \lambda\ v.\ \eta_{M}\ (v, \sigma))\end{array}$$

The bind and return of the given monad M are distinguished from those being defined by attaching a subscript (e.g., η_M). We adopt this convention throughout, eliminating such ambiguities by subscripting when it seems helpful. The "lifting" function, $\mathit{lift} : M\ a \to T\ M\ a$, enriches computations in monad M as computations in $T\ M$ for monad transformer T (e.g., the type for the state monad transformer above is $\mathit{lift} : M\ a \to \mathit{StateT}\ s\ M\ a$). The non-proper morphisms are redefined as:

$$\begin{array}{lcl} g & : & \mathit{StateT}\ s\ M\ s \\ g & = & \mathit{ST}\ (\lambda\sigma.\ \eta_M\ (\sigma, \sigma)) \\ u & : & (s{\to}s) \to \mathit{StateT}\ s\ M\ () \\ u\ \delta & = & \mathit{ST}\ (\lambda\sigma.\ \eta_M\ ((), \delta\ \sigma)) \end{array}$$

As an example, consider the case when $M = I$ and $s = \mathit{Sto}$. The definitions of get and update in this case are given by:

$$\begin{array}{lcl} g & : & \mathit{StateT}\ \mathit{Sto}\ I\ \mathit{Sto} \\ g & = & \mathit{ST}\ (\lambda\sigma.\ I\ (\sigma, \sigma)) \\ u & : & (\mathit{Sto}{\to}\mathit{Sto}) \to \mathit{StateT}\ \mathit{Sto}\ I\ () \\ u\ \delta & = & \mathit{ST}\ (\lambda\sigma.\ I\ ((), \delta\ \sigma)) \end{array}$$

These definitions are equivalent to those given above for the St monad, containing a bit of inconsequential Haskell bureaucracy (namely the I constructors). The instantiations of the bind and unit operations are also equivalent.

A *layered monad* is one constructed from multiple applications of monad transformers; one such construction that we will use shortly is the two-state monad:

$$\mathit{Two} \triangleq \mathit{StateT}\ D\ (\mathit{StateT}\ E\ I)$$

where D and E are fixed types (whose exact structure need not concern us yet). The monad Two has two states with corresponding update and get operations. In Two, the update and get operations corresponding to the D state, defined as u_D and g_D, are given by the application of the $(\mathit{StateT}\ D)$ transformer. The E operators are first created through applying the monad transformer $(\mathit{StateT}\ E)$ to the identity monad I creating $M = \mathit{StateT}\ E\ I$. These initial versions of the E operators are called u_0 and g_0 below. The operators u_0 and g_0 are then "lifted through" $(\mathit{StateT}\ D)$, thereby redefining them for the "bigger" monad $\mathit{Two} = \mathit{StateT}\ D\ (\mathit{StateT}\ E\ I)$; their lifted versions are called u_E and g_E:

$$\begin{array}{ll} u_D : (D{\to}D) \to \mathit{Two}\ () & u_D\ \delta \triangleq \mathit{ST}\ (\lambda d.\ \eta_M\ ((), \delta\ d)) \\ g_D : \mathit{Two}\ D & g_D \quad \triangleq \mathit{ST}\ (\lambda d.\ \eta_M\ (d, d)) \\ u_E : (E{\to}E) \to \mathit{Two}\ () & u_E \quad \triangleq \mathit{lift} \circ u_0 \\ g_E : \mathit{Two}\ E & g_E \quad \triangleq \mathit{lift}\ g_0 \\ u_0 : (E{\to}E) \to M\ () & u_0\ \delta \triangleq \mathit{ST}\ (\lambda e.\ \eta_I\ ((), \delta\ e)) \\ g_0 : M\ E & g_0 \quad \triangleq \mathit{ST}\ (\lambda e.\ \eta_I\ (e, e)) \end{array}$$

A notational convention used throughout attaches a subscript l to any operator acting exclusively on the state layer (e.g., D or E) corresponding to l.

It is important to note that order of application of the state monad transformers is of no consequence—this follows directly from Theorems 1-3 proved below in Section 3.2. An equivalent definition of Two would be produced by applying $(StateT\ D)$ first and then $(StateT\ E)$. This kind of symmetry is the essence of modularity in Modular Monadic Semantics [46, 12, 35]. There are, however, monad transformers for which order of application does matter [35], but none of them are used in this article.

The kernel monad *K*. The remainder of this article assumes a particular two state monad with state types H and L representing the high and low security states in Figure 1 (left):

$$K \triangleq StateT\ H\ (StateT\ L\ I)$$

In the definition of K, $H = L = Sto$. Like Two, the monad K comes equipped with two sets of update and get operators:

$$\begin{aligned}
&u_H : (H \to H) \to K\ ()\\
&g_H : K\ H\\
&u_L : (L \to L) \to K\ ()\\
&g_L : K\ L
\end{aligned}$$

The monad and lifting laws. The $\star$ and η operators satisfy the *monad laws* [35]:

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

Applying a monad transformer T to a monad M extends the notion of computation expressed by M and this extension is captured directly by the lifting operation, *lift*, associated with T. The *lift* operation satisfies the *lifting laws* [35]:

$$\begin{aligned}
&\mathit{lift} : Ma \to TMa\\
&\mathit{lift} \circ \eta_M \quad = \eta_{(TM)}\\
&\mathit{lift}\,(x \star_M f) = (\mathit{lift}\,x) \star_{(TM)} (\mathit{lift} \circ f)
\end{aligned}$$

3.1 Separability via Layered State Monads

Thread execution in a separation kernel is ultimately reflected as a sequence of updates on the Hi and Lo domains and this section describes how separation may be defined monadically and how layering supports separation verification. The Lo domain must be *separable* [64, 63] from the Hi domain; that is, the outputs of threads on Lo should depend only on inputs to Lo threads. Rather than rely on explicit access to input and output states of K computations (which would "break" the monadic abstractions), we characterize separability in terms of interactions between effects in K.

Asymmetry is essential to information flow security in that low security actions are permitted to affect high security computations, but high security actions may not affect low security computations. The original formulation of non-interference [17] characterized this asymmetrical information flow by comparing the “views” of system outputs from the vantage point of low security computations. Given a system trace with both high and low security actions, $m = h_0\,;l_0\,;\ldots;h_n\,;l_n$, and its low security projection, $l = l_0\,;\ldots;l_n$, assume that they respectively produce system outputs o_m and o_l when executed from the same initial state. Goguen-Meseguer non-interference requires that the low security “views” of o_m and o_l are identical because this demonstrates that the execution of the high security actions, h_i, within m had no effect on the low security computations.

Goguen-Meseguer non-interference assumes the ability to filter out high security actions from arbitrary threads; this capability is defined easily as system traces for Goguen and Meseguer are simply objects in an abstract syntax. Filtering out the high security actions in the denotational setting of this article requires a different approach. The formulation of separation via layered state monads requires that high security actions must cancel—what this means is described below. The technique applied here is analogous to that of Joshi and Leino [31] where the cancellation operator (called “havoc on high”) reinitializes the high security storage to some fixed arbitrary value.

Separating Lo from Hi means that Lo-events are unaffected by the execution of Hi-events. The terms “events” or “operations” stand for state-modifying actions such as assignments or message receptions; they are specified precisely by the event semantics, $\mathcal{E}_H[\![-]\!]$ and $\mathcal{E}_L[\![-]\!]$, given below in Sections 4-6. For any sequence of interleaved Hi and Lo operations, $h_0\,;l_0\,;\ldots;h_n\,;l_n$, the effect of its execution on the Lo state should be identical to that of executing the Lo events in isolation, $l_0\,;\ldots;l_n$. If we have a K operation, $mask_H$, that commutes with the Lo operations and cancels the Hi ones (i.e., $h_i; mask_H = mask_H$), then we may extract the Lo effects by erasing the Hi ones:

$$\begin{array}{rcl} h_0\,;l_0\,;\ldots;h_n\,;(l_n\,;mask_H) & = & h_0\,;l_0\,;\ldots;(h_n\,;mask_H)\,;l_n \\ & = & h_0\,;l_0\,;\ldots;mask_H\,;l_n \\ & \vdots & (\text{masking out all } h_i) \\ & = & l_0\,;\ldots;l_n\,;mask_H \end{array}$$

The key insight is that *all* layered state monads have the necessary structure and properties to encode this argument! Each of the operations above may be interpreted within a layered state monad; in particular, “;” is the monadic bind operation. Each equation may then be established via properties of layered state monads, where “=” is ordinary denotational equality of state computations. Layered state monads have intra-layer properties (called *sequencing* and *cancellation* below) guaranteeing the existence of an effect-canceling $mask$ operation. Layered state monads also have inter-layer properties (called *atomic non-interference* below) delimiting the scope of stateful effects: the $mask$ operation from one layer is guaranteed to commute with operations from other layers. This precise control of effects greatly facilitates the verification of separability: the very construction of K provides much of the power to verify separability. The next section defines the intra- and inter-layer properties of layered state monads and then states theorems showing how these properties are inherited by construction.

The cancellation operation, $mask_H$, will be seen to be the monadic analogue of Joshi and Leino's HH (i.e., "havoc on high") operator [31] in that it resets the Hi state to an arbitrary, fixed value. It should be noted that there is a corresponding cancellation operator on the Lo-domain, $mask_L$. It is never used because separation security is asymmetrical: Lo-operations may affect Hi ones but not vice versa.

3.2 Layered State Monads.

This section presents an algebraic characterization of layered state monads. First, a characterization of necessary structure for state monads is given in Definition 1. Then, the required intra-layer behavior of this structure (*sequencing* and *cancellation*) is captured in Definition 2; this is intended to capture the necessary behavior for separation verification and is not meant to be a complete axiomatization of state monads. Then, the necessary inter-layer behavior (*atomic noninterference*) of these non-proper morphisms required later in the proofs is captured by axioms below in Definition 4.

Definition 1. *A* **state monad structure** *is a quintuple* $\langle M, \eta, \star, u, g, s\rangle$ *where* $\langle M, \eta, \star\rangle$ *is a monad, and the* update *and* get *operations on* s *are:* $u : (s \to s) \to M()$ *and* $g : M s$.

We will refer to a state monad structure $\langle M, \eta, \star, u, g, s\rangle$ simply as M if the associated operations and state type are clear from context.

Definition 2. *A* **state monad** *is a state monad structure* $\langle M, \eta, \star, u, g, s\rangle$ *such that the following equations hold for any* $f, f' : s \to s$ *and* $\varphi : M a$,

$$\begin{array}{rcll} u\,f >> u\,f' & = & u\,(f' \circ f) & \qquad\qquad \text{(sequencing)} \\ g >> \varphi & = & \varphi & \qquad\qquad \text{(cancellation)} \end{array}$$

The (sequencing) axiom shows how updating by f and then updating by f' is the same as updating by their composition $(f' \circ f)$. The (cancellation) axiom specifies that g operations whose results are ignored have no effect on the rest of the computation.

A consequence of (sequencing) is that $uf >> u\ (\lambda\_.\sigma_0) = u\ (\lambda\_.\sigma_0)$ where σ_0 is some arbitrary fixed state. This fact justifies the general definition of a $mask$ operator:

Definition 3. *A* **mask** *for state monad* $\langle M, \eta, \star, u, g, s\rangle$ *is defined as*

$$mask_s = u\ (\lambda\_.\sigma_0)$$

for any arbitrary fixed state σ_0.

The (clobber) rule captures the defining property of mask operators:

$$uf >> mask_s \quad = \quad mask_s \qquad\qquad \text{(clobber)}$$

Atomic non-interference is a symmetric form of non-interference used (as in the proof outline Section 3.1) to commute actions on different security domains. Formally, atomic non-interference is a binary relation on $M\,()$ that holds for two computations, φ and γ, when they commute with respect to the operator $>>_M$. That $\varphi >>_M \gamma$ is identical to $\gamma >>_{M} \varphi$ indicates that there is no observable interaction between the effects of φ and γ. Atomic non-interference could be defined at $(Ma \times Mb)$ for arbitrary value types a and b, but only at the expense of extra complexity as one must also specify that the returned values are unaffected by commutation. As the threads considered here only return $()$, the less general form below is sufficient. The general form of atomic non-interference may be found in a non-security context in Harrison [24].

Definition 4. *For monad M with bind operation $\star$, define the* **atomic noninterference** *relation $\# \subseteq M\,() \times M\,()$ so that, for $\varphi, \gamma : M\,()$, $\varphi \# \gamma$ holds if, and only if, the equation $\varphi >>_{M} \gamma = \gamma >>_{M} \varphi$ holds.*

Theorems 1-3 support the construction of modular theories of stateful effects using the state monad transformer. Theorem 1 shows that $StateT$ creates and preserves state monads. Theorems 2 and 3 show that $StateT$ creates and preserves the atomic noninterference relation. These theorems follow by straightforward induction on the type structure of $(StateT\ s\ M)$, assuming M is an arbitrary monad; they are proved in the appendix. Note also that, in each of these results, order of application for $StateT$ is irrelevant. A consequence of these theorems is that the kernel monad K has all of the desired properties supporting separability reasoning as outlined above.

Theorem 1 shows that the state monad transformer creates state monads from arbitrary monads; a consequence of this theorem is that both state layers in K obey sequencing and cancellation.

Theorem 1. *Let M be any monad and $M' = StateT\ s'\ M$ with operations η', $\star'$, lift, g', and u' defined by $(StateT\ s')$. Then:*

1. *$\langle M', \eta', \star', u', g', s' \rangle$ is a state monad.*
2. *$\langle M, \eta, \star, u, g, s \rangle$ is a state monad $\Rightarrow \langle M', \eta', \star', lift \circ u, lift\, g, s \rangle$ is also.*

Theorem 2 shows that, in layered state monads, the update operations are noninterfering; thus, the operations u_{H} and u_{L} in K do not interfere. Note that this theorem establishes a symmetry property in that the order of application of state monad transformers is immaterial—one obtains the same state monad properties regardless of the order of application!

Theorem 2. *Let M be the state monad $\langle M, \eta, \star, u, g, s \rangle$. Let M' be the state monad structure, $\langle StateT\ s'\ M, \eta', \star', u', g', s' \rangle$, defined by $(StateT\ s')$ with operations η', $\star'$, lift, g', and u'. By Theorem 1, M' is also a state monad. Then, for all $f : s \to s$ and $f' : s' \to s'$, $lift(u\, f) \#_{M'} (u'\, f')$ holds.*

Theorem 3 gives a sufficient condition for atomic non-interference to be inherited through monad transformer application.

Theorem 3. *Let M be a monad with two operations, $o : M\,()$ and $p : M\,()$ such that $o \#_{M} p$. Let T be a monad transformer with operator, $lift : M\ a \to (T\ M)\ a$, obeying the lifting laws. Then, $(lift\, o)\ \#_{(T\,M)}\ (lift\, p)$*

Taken together, Theorems 1-3 show that the inter- and intra-layer properties of layered state monads extend to an arbitrarily large number of layers; this allows us to construct and verify separation kernels with more than two domains.

3.3 High-level Architecture of Monadic Separation Kernels.

Monadic separation kernels are constructed from three monads: the kernel monad K, the process monad Re and the scheduling monad R. Illustrated in Figure 3, the kernel architecture consists of these monads and the morphisms that define the relationships between their respective notions of computation. Each domain in the separation kernel is a layer in the kernel monad K. Each such layer is constructed with an application of a state monad transformer, proceeding along the lines described earlier in this section. By construction, K possesses imperative operations associated with each domain/layer and these operations are related by the state monad axioms and the atomic non-interference relation. In Figure 3, K has two layers, corresponding to domains Hi and Lo in Figure 1 (Left), with corresponding imperative operations. More layers may be added with additional state monad transformer applications. The imperative operations on K are precisely the atomic operations in the system. The process monad, Re, encapsulates a notion of computation for reactive concurrency. Its construction is described in detail below in Section 5. Computations in Re are threads whose operations may send request signals to the kernel and affect the state of the domain where they reside. Imperative operations on the Hi and Lo domains are lifted from K to Re using the $step_H$ and $step_L$ morphisms, respectively. The process semantics, $\mathcal{P}_H[\![-]\!]$ and $\mathcal{P}_L[\![-]\!]$, are defined on Re in part with these lifted K operations. The scheduling monad R encapsulates a notion of basic concurrency—that is, threads are sequences of imperative operations on all domains. The kernel rr takes a number of reactive Re-threads and, after servicing thread requests, constructs the scheduling of these threads in R. The $(take_L\ n)$ and run morphisms, defined below in Section 5.2, are used primarily to specify and prove the separation property. The $(take_L\ n)$ and run morphisms project schedulings constructed by the kernel to the K monad where the properties of state monads described in Section 3.2 are used with good effect to prove separation.

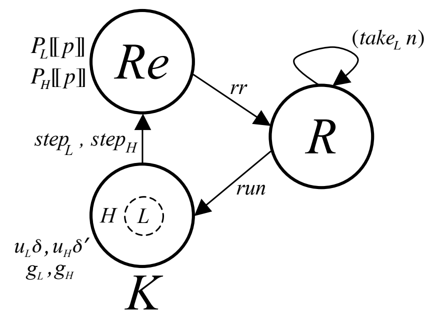

Figure 3: Monadic Architecture

4 The Basic Model of Integrity

While confidentiality policies seek to eliminate inappropriate disclosure of information, integrity policies seek to eliminate inappropriate modification of data. This section demonstrates how monadic fine control of effects addresses integrity concerns. To do so, we present the *basic model of integrity* in Section 4.1. In this kernel, threads in different domains are totally separate—they cannot modify storage in another domain. This complete separation is a direct consequence of the properties of layered state monads developed in Section 3.1. Before the basic integrity model may be described, however, we must formulate its concurrency model, and for this, we use monads of resumptions.

Layered Resumption Monads & Separation. A natural model of concurrency is the trace model [62]. The trace model views threads as (potentially infinite) streams of atomic operations and the meaning of concurrent thread execution as the set of all possible thread interleavings[2]. Resumption monads [46, 53, 25] formalize essentially this model of concurrent computation:

$$\begin{array}{ll}
\mathbf{data}\, R\ a & = \mathit{Done}\ a \mid \mathit{Pause}\ (K\ (R\ a)) \\
(\mathit{Done}\ v) \star_R f & = f\ v \\
(\mathit{Pause}\ r) \star_R f & = \mathit{Pause}\ (r \star_K \lambda\kappa.\ \eta_K\ (\kappa \star_R f)) \\
\eta & = \mathit{Done} \\
\mathit{step} & :\ K\ a \to R\ a \\
\mathit{step}\ x & = \mathit{Pause}\ (x \star_K (\eta_K \circ \mathit{Done}))
\end{array}$$

Here, the bind operator for R is defined recursively in terms of the bind and unit for K. Recall that the kernel monad K was defined above as: $K \triangleq \mathit{StateT}\ H\ (\mathit{StateT}\ L\ I)$ where H and L are fixed types representing the high and low security states and I is the identity monad. In the second clause defining $\star_R$, "$r \star_K \lambda\kappa$" sets κ to be the continuation of r. The recursive call, $\kappa \star_R f$, ensures that the "rest of r" is executed before anything within f. A useful non-proper morphism, step, recasts a K computation as an R computation. We refer to this as the *basic* resumption monad to distinguish it from the more expressive *reactive* variety defined later.

In the trace model, if we have two threads $a = [a_0, a_1]$ and $b = [b_0]$ (where a_0, a_1, and b_0 are atomic operations), then the concurrent execution of threads a and b is denoted by the set of all their interleavings. The basic resumption monad has lazy constructors Pause and Done that play the rôle of the lazy list constructors cons (::) and nil ([ ]) in the trace model. If the operations of a and b are computations of type K (), then any interleaving of a and b may be represented as a computation of type $R()$:

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

where $>>$ and η are the bind and unit operations of the monad K. Where the trace version implicitly uses a lazy cons operation $(h::t)$, the monadic version uses something similar: $\mathit{Pause}\ (h >> \eta\ t)$. 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. In the present work, we consider only infinite threads (i.e., those with no occurrence of Done) as it simplifies our presentation somewhat; however, finite threads may be accommodated easily.

[2]This is a slight simplification that suffices for our presentation.

Basic Monad Transformer. With this discussion in mind, the previous construction may be generalized as a monad transformer [53] defined as:

$$\begin{array}{ll}
\mathbf{data}\ \mathit{ResT}\ M\ a = \mathit{Done}\ a \mid \mathit{Pause}\ (M\ (\mathit{ResT}\ M\ a)) & \\
(\mathit{Done}\ v) \star f & = f\ v \\
(\mathit{Pause}\ r) \star f & = \mathit{Pause}\ (r \star_M \lambda\kappa.\ \eta_M\ (\kappa \star f)) \\
\eta & = \mathit{Done} \\
\mathit{step} & :\ M\ a \rightarrow \mathit{ResT}\ M\ a \\
\mathit{step}\ x & = \mathit{Pause}\ (x \star_M (\eta_M \circ \mathit{Done}))
\end{array}$$

Security-conscious Basic Monad Transformer. In the kernel specifications presented in Sections 4.1, 5.1 and 6.1, we use a "security-conscious" form of basic resumptions. The basic monad transformer above is made security-conscious by reflecting the Hi and Lo security levels in the Pause constructors:

$$\begin{array}{lcl}
\mathbf{data}\ \mathit{ResT}\ M\ a & = & \mathit{Done}\ a \mid \mathit{Pause}_L\ (M\ (\mathit{ResT}\ M\ a)) \mid \mathit{Pause}_H\ (M\ (\mathit{ResT}\ M\ a)) \\
(\mathit{Done}\ v) \star f & = & f\ v \\
(\mathit{Pause}_d\ r) \star f & = & \mathit{Pause}_d\ (r \star_M \lambda\kappa.\ \eta_M\ (\kappa \star f)) \\
\eta & = & \mathit{Done} \\
\mathit{step}_d & : & M\ a\ \rightarrow\ \mathit{ResT}\ M\ a \\
\mathit{step}_d\ x & = & \mathit{Pause}_d\ (x \star_M (\eta_M\ \circ \mathit{Done}))
\end{array}$$

Here, d ranges over security labels: $d \in \{H, L\}$. A low (high) security thread will be formed entirely with Pause_L (Pause_H) constructors; this construction is symmetric in the sense that these constructors are the only distinction between high and low security threads.

Atomic Operations & Thread Construction. It is assumed here and for the remainder of the article (unless otherwise noted) that $R = \mathit{ResT}\ K$ where ResT is the security-conscious basic monad transformer. From the point of view of the separation kernels presented in this article, all operations on K are atomic. In particular, all computations in K built from u_d, g_d, and the bind and unit of K are atomic. Referring to computations "built" from other computations as atomic would seem to be rather unconventional, so let us first illustrate thread construction and then motivate this choice of terminology.

To illustrate atomicity and thread construction, first consider the following "counter" operations on K. Operations for reading, incrementing and decrementing locations, getloc, inc, and dec, respectively, can be defined for each domain in K as:

$$\begin{array}{ll}
\mathit{getloc}_d & :\ \mathit{Loc} \rightarrow K\ \mathit{Int} \\
\mathit{getloc}_d\ l & = (g_d \star_K \lambda\,\sigma.\ \eta_K\ (\sigma\ l)) \\
\mathit{inc}_d & :\ \mathit{Loc} \rightarrow K() \\
\mathit{inc}_d(x) & = \mathit{getloc}_d\ x \star \lambda v.\ u_d[x{\mapsto}v{+}1] \\
\mathit{dec}_d & :\ \mathit{Loc} \rightarrow K() \\
\mathit{dec}_d(x) & = \mathit{getloc}_d\ x \star \lambda v.\ u_d[x{\mapsto}v{-}1]
\end{array}$$

where $[x{\mapsto}v]$ is the function $(\lambda\sigma.\mathbf{if}\ n{=}{=}x\ \mathbf{then}\ v\ \mathbf{else}\ \sigma n)$. The location reading computation $(\mathit{getloc}_d\ l)$ returns the contents of location x on domain d and $(\mathit{inc}_d\ x)$ reads the current value of x on domain d and updates x with that value plus 1. On different domains, these operations refer to different locations. For example, $(inc_{H}\, x)$ and $(inc_{L}\, x)$ increment locations within the H and L address spaces, respectively. The decrement operations, dec_{H} and dec_{L}, are analogous to the increment operations.

Consider now the computation $nop = (inc_{H}(x) >>_{K} dec_{H}(x))$. Monadic semantics is a form of denotational semantics, and it is easy to see that $nop = \eta_{K}()$ as, eschewing Haskell syntax for the moment, they are both the function $\lambda h.\lambda l.((), l, h)$. If nop is to be considered as a thread, one must be able to decompose it into its component actions, $inc_{H}(x)$ and $dec_{H}(x)$. Otherwise, how could one write a scheduler? But, for any such "decomposer" function f, it must be the case that $f(nop) = f(\eta_{K}())$. Generally, a K computation, being an object in type $H \to L \to (t \times L \times H)$ for some type t, does not have an internal structure conducive to writing thread manipulation functions. A computation in K is atomic precisely because it has no such internal structure.

Resumptions were first introduced into denotational semantics [55, 66] to support the partitioning of computations into "steps," thereby providing a suitable foundation for concurrency. A resumption computation, such as $step_{d}\, a_0 >>_{R} step_{d}\, a_1$, can be interrupted at the $>>_{R}$ by pattern-matching on the $Pause$ and $Done$ constructors within the underlying structure of the thread. Pattern-matching on the internal structure of resumption-monadic computations (e.g., $Done$, $Pause_{H}$ and $Pause_{L}$) allows access to the "steps" in a thread:

$$\begin{aligned} f\ (Done\ v) &= \ldots \\ f\ (Pause_{H}\ \varphi) &= \ldots \\ f\ (Pause_{L}\ \varphi) &= \ldots \end{aligned}$$

Thread manipulation functions (such as the kernels developed below and in Sections 5 and 6) are simply co-recursive functions on resumption monads.

Threads are constructed by chaining together lifted atoms with the bind of a resumption monad. *Lifted atoms* are computations $step_{d}(a)$ for some atomic a. For example, a thread on the Hi domain with three successive increment operations on locations x, y and z looks like:

$$t_H = step_{H}(inc_{H}(x)) >>_{R} step_{H}(inc_{H}(y)) >>_{R} step_{H}(inc_{H}(z))$$

A thread that increments location x repeatedly is written:

$$loop\ =\ step_{H}\ (inc_{H}\ x) >>_{R} loop$$

Note that threads t_H and *loop* will interfere with one another because they both update location x on the Hi domain. However the following Lo thread cannot interfere with either of these two threads:

$$t_L = step_{L}(inc_{L}(x)) >>_{R} step_{L}(inc_{L}(y)) >>_{R} step_{L}(inc_{L}(z))$$

The reason for this, as will be shown later, is that the atoms in the threads cannot interfere, or, more precisely, because $inc_{L}(x) \# inc_{H}(x)$ holds.

One might reasonably ask how it can be that *loop* does not "go off into an infinite loop"? To understand why this is, it is helpful to expose the underlying structure of threads in R. This can be accomplished by unfolding the definitions of $step_H$ and $>>_R$ in the $loop$ thread. Let $f \;=\; \lambda\_.\ loop$, then:

$$
\begin{array}{ll}
loop & = step_H(inc_H(x)) >>_R loop \\
\{\text{def. } step_H\} & = Pause_H(inc_H(x) \star_K (\eta_K \circ Done)) \star_R f \\
\{\text{def. } \star_R\} & = Pause_H((inc_H(x) \star_K (\eta_K \circ Done)) \star_K \lambda\kappa.\eta_K(\kappa \star_R f)) \\
\{\text{assoc. } \star_K\} & = Pause_H(inc_H(x) \star_K \lambda v.\ (\eta_K(Done\ v) \star_K \lambda\kappa.\eta_K(\kappa \star_R f))) \\
\{\text{left-unit } K\} & = Pause_H(inc_H(x) \star_K \lambda v.\ \eta_K((Done\ v) \star_R f)) \\
\{\text{left-unit } R\} & = Pause_H(inc_H(x) \star_K \lambda v.\ \eta_K(f\ v)) \\
\{\text{def. f}\} & = Pause_H(inc_H(x) >>_K \eta_K(loop))
\end{array}
$$

Haskell's default lazy semantics for constructors (like $Pause_H$) freezes the recursive calls to $loop$ until its structure is exposed by a pattern match. The occurrence of $loop$ in the last line above could be unfolded yet further:

$$= Pause_H(inc_H(x) >>_K \eta_K(Pause_H(inc_H(x) >>_K \eta_K(loop))))$$

Clearly, the unfolding process on $loop$ could proceed *ad infinitum*.

This example also helps explain how the kernel can distinguish individual atoms in a thread with pattern-matching (i.e., interrupt them). The computation $loop$ has the form $(Pause_H\ \varphi)$ for the K computation φ. Computation φ has the side effect of incrementing location x on the Hi domain, but the value it returns differs from $inc_H(x)$. Instead of the unit value $()$, φ returns the remainder of the thread: namely, $loop$. Thus, the kernel can get access to both the "head" of the thread, $inc_H(x)$, and the "tail" of the thread, $loop$.

Approximation Lemma for Basic Resumptions. There are well-known techniques for proving equalities of infinite lists; these are the *take lemma* [7] and the more general *approximation lemma* [16]. Both are based on the fact that, if all initial prefixes of two lists are equal, then the lists themselves are equal; it does not matter whether the lists are finite, infinite, or partial. The approximation lemma may be stated as: for any two lists xs and ys,

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

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

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

A consequence of excluding the 0^{th} case in this definition is that: $approx\ 0\ \_ = \bot$.

A similar result holds for basic resumption computations as well. Here, we consider R defined more generally than in the previous section; that is, its type constructor is written in terms of an arbitrary monad M (rather than K):

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

The $\star$ and η below in the definition of $approx$ are those of the monad M. The

Haskell function approx approximates R computations:

$$\begin{array}{l} \mathit{approx} \;:\; \mathit{Int} \to R\,a \to R\,a \\ \mathit{approx}\ (n{+}1)\ (\mathit{Done}\ v) \ \ = \mathit{Done}\ v \\ \mathit{approx}\ (n{+}1)\ (\mathit{Pause}\ \varphi) = \mathit{Pause}\ (\varphi \star (\eta\ \circ\ \mathit{approx}\ n)) \end{array}$$

Both versions of approx—for lists and resumptions—are undefined for the $n = 0$ case; the coarsest-grained approximations of any two elements in a pointed domain is always $\bot$ (i.e., $\mathit{approx}\ 0\ \_ = \bot$ for basic resumptions as well). Note that, for any finite resumption-computation φ, $\mathit{approx}\ n\ \varphi = \varphi$ for any sufficiently large n—that is, $(\mathit{approx}\ n)$ approximates the identity function on resumption computations. The approximation lemma for basic resumptions is stated:

Theorem 4 (Approximation Lemma for Basic Resumptions)**.** For any $\varphi, \gamma : R\ a$, $\varphi = \gamma \Leftrightarrow$ for all $n \in \omega,\ \mathit{approx}\ n\ \varphi = \mathit{approx}\ n\ \gamma$.

Theorem 4 is proved in Appendix B.

4.1 Point 1: Basic Model of Integrity

We now have all the necessary raw materials to build the basic model of integrity (point 1 of Figure 2); the good news is that, modulo a simple refinement to the resumption-monadic concurrency model, we have what we need to build the other kernels as well. Constructing the basic model entails giving monadic semantics to the Event, $\mathit{Process}$, and Exp languages (defined previously in Section 2), as well as specifying a scheduler. It is assumed the monad K is defined as in Section 3, that the H and L types model disjoint address spaces, and that R is defined from K using the resumption monad transformer:

$$\begin{array}{ll} \mathbf{type}\ \mathit{Loc} = \mathit{String} \qquad & \mathbf{type}\ L\ = \mathit{Loc}{\to}\mathit{Int} \\ \mathbf{type}\ R \ \ = \mathit{ResT}\ K & \mathbf{type}\ H = \mathit{Loc}{\to}\mathit{Int} \end{array}$$

These constructions provide the following non-proper morphisms: lift, step_L, step_H, g_L, g_H, u_L, and u_H. For the sake of convenience, we define the following helper functions; $(\mathit{getloc}_d\ l)$ reads the contents of location l on domain d and $(\mathit{setloc}_d\ l\ v)$ stores v at location l on domain d:

$$\begin{array}{ll} \mathit{getloc}_d & :\ \mathit{Loc}\ \to\ K\ \mathit{Int} \\ \mathit{getloc}_d\ l & = (g_d \star_K \lambda\,\sigma.\ \eta_K\ (\sigma\ l)) \\ \mathit{setloc}_d & :\ \mathit{Loc}\ \to\ \mathit{Int}\ \to\ K\ a \\ \mathit{setloc}_d\ l\ v & = u_d\ [l{\mapsto}v] \\ [i{\mapsto}v] & = \lambda\,\sigma.\ \lambda\,n.\ \mathit{if}\ i{=}n\ \mathit{then}\ v\ \mathit{else}\ \sigma\ n \end{array}$$

The expression semantics, $\mathcal{V}_d[\![-]\!]$, is a standard definition for expressions in the presence of state (and is included to create more expressive system demonstrations as appear later in Figure 5):

$$
\begin{array}{ll}
\mathcal{V}_d[\![-]\!] & : \ Exp \ \to \ K\ Int \\
\mathcal{V}_d[\![i]\!] & = \eta\ i \\
\mathcal{V}_d[\![l]\!] & = getloc_d\ l \\
\mathcal{V}_d[\![e_1 \odot e_2]\!] & = \mathcal{V}_d[\![e_1]\!] \star \lambda v_1.\ \mathcal{V}_d[\![e_2]\!] \star \lambda v_2.\ \eta\ (v_1 \odot v_2)
\end{array}
$$

The operation $\odot$ refers to any standard binary function on integers.

There is only one event in this system—an assignment $trg\,{:}{=}src$. Its semantics, given below, computes the expression src and $setloc$s the result in the trg location; this K computation is cast as an atomic action in R using $step_d$:

$$
\begin{array}{ll}
\mathcal{E}_d[\![-]\!] & : \ Event \ \to \ R\,() \\
\mathcal{E}_d[\![l\,{:}{=}e]\!] & = step_d\ (\mathcal{V}_d[\![e]\!] \star setloc_d\ l) \\
\mathcal{P}_d[\![-]\!] & : \ Process \ \to \ R\,() \\
\mathcal{P}_d[\![ev\ ;\ evs]\!] & = \mathcal{E}_d[\![ev]\!] >> \mathcal{P}_d[\![evs]\!]
\end{array}
$$

The process semantics, $\mathcal{P}_d[\![-]\!]$, gives rise to a precise notion of thread: a *thread* is defined as any R computation, φ, for which there is a $p \in Process$ such that $\mathcal{P}_d[\![p]\!] = \varphi$. Notice that the notion of thread includes any "tail" portion of a process denotation; for example, if $(step_d\,\gamma >>_R t)$ is a thread, then so is t. This follows directly from the structure of the process semantics. We expand on this notion in Section 5.3 below.

The corecursive function, rr, defines a scheduler for the basic integrity model. A round-robin scheduling of threads, $rr\ ts$, is created by interleaving the threads on the waiting thread list ts:

$$
\begin{array}{ll}
\multicolumn{2}{l}{rr\ :\ [R\,()]\ \to\ R\,()} \\
rr\ [] & = Done\ () \\
rr\ ((D\ \_){::}ts) & = rr\ ts \\
rr\ ((Pause_d\ t){::}ts) & = Pause_d\ (t \star \lambda\kappa.\ \eta\ (rr\ (ts {+\!\!+} [\kappa])))
\end{array}
$$

We assume that rr is applied to threads (i.e., elements within the range of $\mathcal{P}_d[\![-]\!]$) and that ts is a finite list. A process p is executed on domain d of this separation kernel by including $\mathcal{E}_d[\![p]\!]$ in ts; it is assumed for the remainder that all system executions arise in this manner.

Each kernel in the present work uses this round-robin scheduling strategy, although the separation security specification developed below in Section 5.2 does not rely on this particular scheduling regime in any essential way. A promising approach to generalizing the monadic view of information flow security advocated here begins with refining the kernel structure with an additional monadic effect, namely non-determinism [46, 74, 73, 12, 35]. While resumptions without non-determinism resemble streams, resumptions with non-determinism resemble trees. This additional effect allows the expression of a "tree of all possible schedulings" computation. The generalized form of the current work would reformulate the separation property of Section 5.2 in terms of this "tree". The monadic foundations for this generalized approach are developed in detail by Harrison et al. [26], but, as of this writing, the details with respect to security have not been published.

5 Allowing Secure Interdomain Interaction

This section extends the basic integrity model to include primitives for interdomain interaction—in this case, asynchronous message broadcast and blocking receive events—that introduce the possibility of insecure information flow. Interdomain interactions are mediated entirely through the separation kernel as in Rushby's original conception [64, 63] and it is in the kernel that the "no write down" security policy is enforced. This extension follows the pattern of modular language definitions [35, 24] as well in that the text of the basic integrity model remains almost entirely intact within the enhanced kernel; the increased system functionality comes about through refinements to the underlying monads. The encapsulation of the new reactive features (i.e., message-passing primitives) by a monad transformer aids the security verification by isolating them from the other kernel building blocks.

Before the interdomain communication kernel (i.e., point 2 of Figure 2) is presented in Section 5.1, the necessary refinement—adding reactivity—to the monadic theory of concurrency is outlined. Then, the kernel itself is presented and the security property for monadic separation kernels is specified and verified.

Reactive Concurrency & Separation. A *reactive* program [38] is one that interacts continually with its environment and may be designed to not terminate (e.g., an operating system). We coin the term *reactivity* to mean the notion of computation given by reactive programs. We now consider a refinement to the concurrency model presented in Section 4 that allows computations to signal requests to the kernel and receive responses from it; we coin the term *reactive* resumption monad to distinguish this structure from the previous one. Although the reactive resumption monadic structure is mentioned in passing in the literature [46, 13], it was never named to the authors' best knowledge. A reactive resumption monad has constructors for pausing computations just as basic resumption monads do, and it also extends the basic resumption structure with Unix-like system requests and responses.

As with basic resumptions, reactive resumption monads may also be generalized as a monad transformer [46]. The following monad transformer abstracts over the request and response data types (q and r, respectively) as well as over the given monad M. The D and P constructors correspond to the $Done$ and $Pause$ constructors in a basic resumption monad:

$$\begin{array}{lcl}
\mathbf{data}\ \mathit{ReactT}\ q\ r\ M\ a & = & D\ a \mid P\ (q,\ r {\rightarrow} (M\ (\mathit{ReactT}\ q\ r\ M\ a))) \\
\eta\ v & = & D\ v \\
(D\ v) \star f & = & f\ v \\
P\ (\mathit{req}, \omega) \star f & = & P\ (\mathit{req},\ \lambda \mathit{rsp}.\ (\omega\ \mathit{rsp}) \star_{M} \lambda \kappa.\ \eta_{M}\ (\kappa \star f))
\end{array}$$

The response rsp to request req is passed to the rest of the computation ω in the last clause. In the definition of bind above, ω is a function taking a system response of type r to the "rest of the computation"; κ is the continuation produced by ω when it receives the system response rsp.

The response and request data types (hereafter, Req and Rsp) are required to have certain minimal structure. The continue request, $Cont$, signifies merely that the computation wishes to continue, while the acknowledge response, Ack, is an information-free acknowledgment. Req and Rsp encapsulate the interaction interface between threads and the kernel:

$$\begin{array}{l}\mathbf{data}\ Req = Cont \mid \langle\text{other requests}\rangle \\ \mathbf{data}\ Rsp = Ack \mid \langle\text{other responses}\rangle\end{array}$$

Enhancing kernel functionality typically begins by extending these data types with other request/response signals. The Req and Rsp data types define the system call interface. Adding useful kernel functionality (e.g., page faults) begins by extending this interface.

Reactive Resumption Monad Transformer. Reactive resumption monads have two non-proper morphisms. The first of these, $step$, is defined analogously to its definition in $ResT$. The definition of $step$ shows why we require that Req and Rsp have a particular shape including $Cont$ and Ack, respectively; namely, there must be at least one request/response pair for the definition of $step$. Another non-proper morphism provided by $ReactT$ allows a computation to raise a signal; its definition is given below. Furthermore, there are certain cases where the response to a signal is intentionally ignored, for which we define $signull$:

$$\begin{array}{ll}Re & = ReactT\ Req\ Rsp\ M \\ step & :\ M\ a \rightarrow Re\ a \\ step\ x & = P\,(Cont,\ \lambda Ack.\ x \star_{M} (\eta_{M} \circ D)) \\ signal & :\ Req \rightarrow Re\ Rsp \\ signal\ q & = P\,(q,\ \eta_{M} \circ \eta_{Re}) \\ signull & :\ Req \rightarrow Re\,() \\ signull\ q & = P\,(q,\ \eta_{M} \circ \eta_{Re} \circ (\lambda\_.\,()))\end{array}$$

In the definition of $(signal\ q)$ above, the effect of the composition, $\eta_{M} \circ \eta_{Re}$, is that the system response ultimately passed to this function will be returned as the value of the computation. The pre-composition of $(\lambda\_.\,())$ in the definition of $(signull\ q)$, in contrast, will replace this system response by the nil value, $()$.

Different versions of $step$ with R and Re are used in this paper, but without ambiguity as the version of $step$ is determined by the type context of its use. Within the atom $(step\ x)$, a $Cont$ request by a user thread expects the response Ack and this expectation is encoded by the pattern "λAck." This imposes constraint on the kernel to always respond to a $Cont$ request with an Ack acknowledgment. Each kernel in this article obeys this simple, sensible constraint. It is a simple matter to design a semantics in which processes handle any response from the kernel, although, for the purposes of the present article, this was seen as a needless distraction.

The "λAck" pattern introduces the possibility of pattern-match failure for a kernel that does not handle requests as expected; for example, were a kernel to handle a request to continue, $Cont$, by passing a message to the thread instead of an Ack, that would cause a pattern match failure. The kernels in this article, however, avoid these nonsense cases and it is a simple matter to guarantee the absence of pattern match failure.

Security-conscious Reactive Resumption Monad Transformer. We make the monad transformer $(ReactT\ q\ r)$ security-conscious as before by including a high and low security pause; this is the version used hereafter:

$$\begin{aligned}\mathbf{data}\ ReactT\ q\ r\ M\ a\ =\ & D\ a\\ \mid\ & P_L\,(q,\ r \to (M\ (ReactT\ q\ r\ M\ a)))\\ \mid\ & P_H\,(q,\ r \to (M\ (ReactT\ q\ r\ M\ a)))\end{aligned}$$

The bind and unit operations are defined analogously to $ResT$ as are the high and low security versions of the $step$, $signal$, and $signull$ [23]. Note that the $ResT$ monad transformer is a special case of reactive monad transformer; for any monad M, $ResT\ M\ a\ \cong\ ReactT\ ()\,()\ M\ a$.

5.1 Point 2: Interdomain Communication

This section considers point 2 in Figure 2: the extension of the basic model of integrity of Section 4.1 to express interdomain communication. Any such extension requires demonstration that Hi domain threads cannot affect Lo threads—in this case that the system obeys a "no write down" security policy.

The $Event$ language is extended with two new events, $\mathtt{bcast}(l)$ and $\mathtt{recv}(l)$, and accommodating them requires the introduction of reactivity. To this end, Req is extended with broadcast and receive request tags ($Bcst\ Int$ and Rcv, respectively) and Rsp is extended with the response to a receive request, $(Msg\ Int)$:

$$\begin{aligned}&\mathbf{type}\ Re\ \ = ReactT\ Req\ Rsp\ K\\ &\mathbf{data}\ Req = Cont \mid Bcst\ Int \mid Rcv\\ &\mathbf{data}\ Rsp = Ack \mid Msg\ Int\end{aligned}$$

The types of the process and event semantics have changed to reflect the new monad Re (i.e., $\mathcal{P}_d[\![-]\!]\ :\ Process \to Re()$ and $\mathcal{E}_d[\![-]\!]\ :\ Event \to Re\ ()$), but the text of the semantic equations for the existing event, $l\,{:=}\,e$, has not:

$$\begin{aligned}&\mathcal{E}_d[\![l\,{:=}\,e]\!] &&= step_d\ (\mathcal{V}_d[\![e]\!] \star_K setloc_d\ l)\\ &\mathcal{E}_d[\![\mathtt{bcast}(x)]\!] &&= step_d\ (getloc_d\ x) \star_{Re} (signull_d\ \circ\ Bcst)\\ &\mathcal{E}_d[\![\mathtt{recv}(x)]\!] &&= (signal_d\ Rcv) \star_{Re} \lambda(Msg\ m).\ step_d(setloc_d\ x\ m)\end{aligned}$$

The $\mathtt{bcast}(x)$ event reads the contents of x and requests its broadcast through a $Bcst$ signal. The $\mathtt{recv}(x)$ event signals a Rcv request, and, once message m is received, writes it to location x. Notice that threads requesting to receive a message expect a message as a response. This imposes constraint on kernels for these threads to always respond to a Rcv request with a $(Msg\ m)$. Each kernel in this article obeys this simple, sensible constraint.

The kernel, $rr : ([Re()], [Int], [Int]) \to R()$, is defined in Figure 4. The kernel is a corecursive function taking a tuple, (ts, l, h), consisting of a list of threads (ts) and a message buffer for Lo (l) and Hi (h) as input; it extends its predecessor with cases handling the message-passing requests. Figure 4 introduces some shorthand useful in defining the scheduler rr. If a request may be handled by rr without affecting K, then

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{type}\ \mathit{System}\ =\ ([\mathit{Re}\,()], [\mathit{Int}], [\mathit{Int}])} \\
\multicolumn{2}{l}{\quad \text{— } \mathit{System} = (\langle \mathit{Runnable\ Threads}\rangle, \langle \mathsf{Lo}\ \mathit{message\ queue}\rangle, \langle \mathsf{Hi}\ \mathit{message\ queue}\rangle)} \\
\multicolumn{2}{l}{rr : \mathit{System} \to R\,()} \\
rr\ ([], \_, \_) & = \mathit{Done}\ () \\
rr\ ((D\ \_){::}ts, l, h) & = rr\ (ts, l, h) \\
rr\ (P_d(\mathit{Cont}, r){::}ts, l, h) & = \mathit{Pause}_d\ ((r\ \mathit{Ack}) \star_K \lambda\kappa.\ \eta_K\ (rr\ (ts{\oplus}\kappa, l, h))) \\
rr\ (P_H(\mathit{Bcst}\ m, r){::}ts, l, h) & = \mathit{Pause}_H\ ((r\ \mathit{Ack}) \star_K \lambda\kappa.\ \eta_K\ (rr\ (ts{\oplus}\kappa, l, h{\oplus}m))) \\
rr\ (P_L(\mathit{Bcst}\ m, r){::}ts, l, h) & = \mathit{Pause}_L\ ((r\ \mathit{Ack}) \star_K \lambda\kappa.\ \eta_K\ (rr\ (ts{\oplus}\kappa, l{\oplus}m, h{\oplus}m))) \\
rr\ (P_H(\mathit{Rcv}, r){::}ts, l, []) & = \mathit{next}_H\ (ts{\oplus}P_H(\mathit{Rcv},\ r), l, []) \\
rr\ (P_H(\mathit{Rcv}, r){::}ts, l, (m{::}hs)) & = \mathit{Pause}_H\ ((r\ (\mathit{Msg}\ m)) \star_K \lambda\kappa.\ \eta_K\ (rr\ (ts{\oplus}\kappa, l, hs))) \\
rr\ (P_L(\mathit{Rcv}, r){::}ts, [], h) & = \mathit{next}_L\ (ts{\oplus}P_L(\mathit{Rcv},\ r), [], h) \\
rr\ (P_L(\mathit{Rcv}, r){::}ts, (m{::}ls), h) & = \mathit{Pause}_L\ ((r\ (\mathit{Msg}\ m)) \star_K \lambda\kappa.\ \eta_K\ (rr\ (ts{\oplus}\kappa, ls, h)))
\end{array}
$$

$$
\begin{array}{l}
\oplus \quad\ :\ [a]{\to}a{\to}[a] \\
\oplus\ \ l\ a = l{+\!\!+}[a] \\
\mathit{next}_d\ :\ \ \mathit{System}{\to}R\ () \\
\mathit{next}_d\ = \mathit{Pause}_d \circ \eta_K \circ rr
\end{array}
$$

Figure 4: *Kernel for interdomain communication.* The type *System* encapsulates the kernel resources; these are the runnable threads and the Lo and Hi message queues, respectively.

next_d is used. Recall that each process is of infinite "length," so, strictly speaking, neither the [] nor D cases of *rr* are necessary in Figure 4. We retain these cases for the sake of generality; Section 6, in particular, introduces thread-terminating functionality for which the first of these cases is necessary.

Note that the Hi broadcast affects the Hi buffer only, while the Lo affects both Hi and Lo—this is precisely where the "no write down" policy is manifested. If a thread tries to receive on an empty buffer, it delays. Note also that both varieties of resumption monad occur—the reactive for threads and the basic for schedulings.

5.2 The Security Property: Take-Separation

This section develops the noninterference style security specification for monad structured separation kernels. Separation in the resumption-monadic setting resembles a well-known technique for proving infinite streams equal based on the *take lemma* [7]—whence it takes its name. Two streams are equal, according to the take lemma, if, and only if, the first n elements of each are equal for every $n{\geq}0$. *Take equivalence* is similar in that it quantifies over initial segments of R computations—two R computations are take equivalent if the "masking out" of effects on the Hi domain within the initial segments of each leaves the Lo events unaffected. Such initial segments are compared by projecting them to the K monad, thereby allowing reasoning in the style of Section 3.1. We make this notion precise below, but it is interesting to note that this technique has much the same flavor as observational or behavioral equivalence proof techniques.

Two morphisms useful in formulating take equivalence are run and take_L from Figure 3; they are used to capture and project the aforementioned initial sequences. The

run morphism projects basic resumption computations to K. The run morphism is not factored according to security level as some other operations have been; it projects a system execution down to a single K computation consisting of interwoven Hi and Lo operations. This facilitates algebraic reasoning along the lines described in the beginning of Section 3.1. The operation $(take_{L}\ n)$ may be thought of as a security-conscious version of $(approx\ n)$ from Section 4. It partitions a thread t into two parts. The first part of $(take_{L}\ n\ t)$ is the smallest initial segment or "prefix" of t containing n operations on Lo while the second part returns the rest of t as its value. The run morphism is defined as:

$$
\begin{array}{l}
run\ :\ R\,a\ \rightarrow\ K\,a\\
run\ (Done\ v)\quad = \eta_{K}\ v\\
run\ (Pause_{d}\ \varphi) = \varphi \star_{K} run
\end{array}
$$

The $take_{L}$ morphism is defined as:

$$
\begin{array}{l}
take_{L}\ :\ Int\ \rightarrow\ R\,a\ \rightarrow\ R\,(R\,a)\\
take_{L}\,0\,x \qquad\qquad\ \ = Done\ x\\
take_{L}\ n\ (Pause_{L}\ \varphi) = Pause_{L}\ (\varphi \star_{K} (\eta_{K} \circ (take_{L}\ (n{-}1))))\\
take_{L}\ n\ (Pause_{H}\ \varphi) = Pause_{H}\ (\varphi \star_{K} (\eta_{K} \circ (take_{L}\ n)))
\end{array}
$$

Two properties of run and $take_{L}$ allow us to examine the resumption computations arising from execution of monadic separation kernels. Property (1) shows how run distributes over R computations to produce K computations. The run morphism is a kind of inverse for the $step$ morphism of the basic resumption monad R; this is the essence of Property (1). Property (2) demonstrates how an initial segment of an infinite R computation may be separated into "head" and "tail" parts. Both of these properties are useful in structuring separation proofs; they are:

$$
\begin{array}{rcl}
run(x \star_{R} f) & = & (run\ x) \star_{K} (run \circ f) \qquad (1)\\
take_{L}\ (n{+}1)\ \varphi & = & (take_{L}\ 1\ \varphi) \star_{R} (take_{L}\ n) \qquad (2)
\end{array}
$$

where φ is an infinite resumption (i.e., one constructed entirely without $Done$). Property (1) may be proved easily by induction on the length of its argument if it is finite. If the resumption computation $(x \star_{R} f)$ is infinite, then the property is trivially true, because, in that case, both sides of (1) are denoted by $\perp$. Note also that $(take_{L}\ n\ \varphi)$ is always finite. Property (2) follows by induction on the n parameter.

Definition 5 makes the notion of take equivalence precise. $(take_{L}\ n\ \varphi)$ is the smallest finite initial segment of φ containing n operations on Lo. Applying run to this segment projects it to K, where the Hi operations may be "erased" as in Section 3.1. Two R computations, for which all of these erased initial segments are equal, are take equivalent.

Definition 5 (Take Equivalence)**.** *Let* $\varphi,\ \gamma\ :\ R\,()$ *be two computations, then* φ *and* γ *are* take equivalent (*written* $\equiv_{te}$) *if, and only if, for each* $n{\geq}1$, *the following holds*

$$run\ (take_{L}\ n\ \varphi) >>_{K} mask_{H}\ =\ run\ (take_{L}\ n\ \gamma) >>_{K} mask_{H}$$

Recall from Definition 3 that $mask_{H} = u_{H}(\lambda\_.\sigma_0)$ *for some arbitrary fixed state* σ_0.

For $(ts, l, h) : System$, its restriction to the Lo domain, $(ts, l, h)\downarrow_{\mathsf{Lo}}$, is defined as: $(ts, l, h)\downarrow_{\mathsf{Lo}} = (ts', l, [])$ for ts' containing only the Lo threads occurring in ts (in identical order of occurrence). To put it another way, $(ts, l, h)\downarrow_{\mathsf{Lo}}$ differs from $(ts, l, [])$ only in that any Hi threads are filtered out of ts. We will sometimes use set theoretic notation with values $s : System$ when the meaning is clear. In particular, "$(ts, l, h) \neq \emptyset$" means that the ready list, ts, is non-empty (i.e., $ts \neq []$).

Using the $(\equiv_{te})$ relation, we may define domain separation:

Definition 6 (Take-Separation Property). *Domain separation holds for the kernels* (*i.e., one of points 1-3*) *if, and only if,* $rr\ sys\ \equiv_{te}\ rr\ sys\downarrow_{\mathsf{Lo}}$ *for every* $sys{:}System$ *such that* $sys\downarrow_{\mathsf{Lo}}{\neq}\emptyset$.

Definition 6 requires that the combined effect on Lo of running ts on the operating system is the same as running the Lo threads of ts in isolation—precisely what one would expect from Rushby's original formulation.

Proving the take-separation property for the interdomain communication kernel is considered below in Section 5.3. Theorem 5 is not equivalent to general take-separation, but it does capture the salient issue with respect to information security for this kernel: Hi broadcasts have no effect on Lo receives.

Theorem 5 (no write down). *For* $x, y : Loc$ *and* $l, h : [Int]$,

$$\begin{aligned}&run\ (take_L\ 1\ (rr\ ([\mathcal{E}_H[\![\mathtt{bcast}(x)]\!] >>_{Re} \mathcal{E}_L[\![\mathtt{recv}(y)]\!]], l, h))) >>_K mask_H \\ &\qquad = run(take_L\ 1\ \ (rr\ ([\mathcal{E}_L[\![\mathtt{recv}(y)]\!]], l, h))) >>_K mask_H\end{aligned}$$

Proof. Below are three properties used in this proof, each of which may be proved in a straightforward manner:

$$\begin{array}{lll} rr([\mathcal{E}_L[\![\mathtt{recv}(y)]\!]], l, h) & = rr([\mathcal{E}_L[\![\mathtt{recv}(y)]\!]], l, h') & (i) \\ run\ (Pause_d(\varphi \star_K \lambda v.\ \eta_K\ \gamma)) & = \varphi \star_K \lambda v.\ run\ \gamma & (ii) \\ run\ (Pause_d(\eta_K\ \varphi)) & = run\ \varphi & (iii) \end{array}$$

The first observation—that Lo receives are oblivious to the contents of the Hi message buffer—is proved by inspection of the kernel rr itself. The second follows by the definition of run and the associativity and left unit monad laws; while the third follows by the definition of run and the left unit monad law.

First, we note that the r.h.s. of the theorem reduces to $mask_H$. How the kernel services receive signals on the Lo domain depends on the contents of the Lo message queue l. Below, we assume that $l = []$, although the proof for the case when l is non-empty is similar:

$$\begin{aligned}&run\ (take_L\ 1\ (rr\ ([\mathcal{E}_L[\![\mathtt{recv}(y)]\!]], [], h))) >>_K mask_H \\ &\{\text{def. } \mathcal{E}_L[\![-]\!], \kappa = \lambda(Msg\ v).step_L(setloc_L\ y\ v)\} \\ &\quad = run\ (take_L\ 1\ (rr\ ([signal_L(Rcv) \star_{Re} \kappa], [], h))) >>_K mask_H \\ &\{\text{defs. } rr, signal_L \ \&\ next_L\} \\ &\quad = run\ (take_L\ 1\ (Pause_L\ (\eta_K\ (rr\ ([signal_L(Rcv) \star_{Re} \kappa], [], h)))) >>_K mask_H \\ &\{\text{def. } take_L\}\end{aligned}$$

$$= run\ (Pause_L\ (\eta_K\ (take_L\ 0\ (rr\ ([signal_L(Rcv) \star_{Re}\ \kappa], [], h)))) >\!\!>_K mask_H$$

$\{$def. $take_L$, $\kappa' = rr([signal_L(Rcv) \star_{Re} \kappa], [], h)\}$

$$= run\ (Pause_L\ (\eta_K\ (Done\ \kappa')) >\!\!>_K mask_H$$

$\{$def. $run\}$

$$= (\eta_K\ (Done\ \kappa') \star_K run) >\!\!>_K mask_H$$

$\{$left unit$\}$

$$= run\ (Done\ \kappa') >\!\!>_K mask_H$$

$\{$defn. $run\}$

$$= \eta_K(\kappa') >\!\!>_K mask_H$$

$\{$left unit$\}$

$$= mask_H$$

Now, reducing the l.h.s. along similar lines:

$$run(take_L\ 1\ (rr([\mathcal{E}_H[\![\texttt{bcast}(x)]\!] >\!\!>_{Re} \mathcal{E}_L[\![\texttt{recv}(y)]\!]], l, h))) >\!\!>_K mask_H$$

$\{$def. $\mathcal{E}_H[\![-]\!]$, $\kappa_0 = \lambda v.signal_H(Bcst\ v) >\!\!>_{Re} \mathcal{E}_L[\![\texttt{recv}(y)]\!]\}$

$$= run\ (take_L\ 1\ (rr\ ([step_H(getloc_H(x)) \star_{Re}\ \kappa_0], l, h))) >\!\!>_K mask_H$$

$\{$defn. rr, $\kappa_1 = \mathcal{E}_L[\![\texttt{recv}(y)]\!]\}$

$$= run\ (take_L\ 1\ (Pause_H\ (getloc_H(x) \star_K \lambda v.$$
$$\eta_K\ (rr\ ([signal_H(Bcst\ v) >\!\!>_{Re} \kappa_1], l, h))))) >\!\!>_K mask_H$$

$\{$defn. $take_L\}$

$$= run\ (Pause_H\ (getloc_H(x) \star_K \lambda v.$$
$$\eta_K\ (take_L\ 1\ (rr\ ([signal_H(Bcst\ v) >\!\!>_{Re} \kappa_1], l, h))))) >\!\!>_K mask_H$$

$\{(ii)\}$

$$= getloc_H(x) \star_K \lambda v.$$
$$run\ (take_L\ 1\ (rr\ ([signal_H(Bcst\ v) >\!\!>_{Re} \kappa_1], l, h))) >\!\!>_K mask_H$$

To process the Hi broadcast, repeating the same reductions on rr, $take_L$, and run:

$$= getloc_H(x) \star_K \lambda v.\ run\ (take_L\ 1\ (rr\ ([\kappa_1], l, h \oplus v))) >\!\!>_K mask_H$$

$\{$defn. $\kappa_1\}$

$$= getloc_H(x) \star_K \lambda v.\ run\ (take_L\ 1\ (rr\ ([\mathcal{E}_L[\![\texttt{recv}(y)]\!]], l, h \oplus v))) >\!\!>_K mask_H$$

$\{(i)\}$

$$= getloc_H(x) \star_K \lambda v.\ run\ (take_L\ 1\ (rr\ ([\mathcal{E}_L[\![\texttt{recv}(y)]\!]], l, h))) >\!\!>_K mask_H$$

$\{$see above$\}$

$$= getloc_H(x) \star_K \lambda v.\ mask_H$$

$\{$cancellation$\}$

$$= mask_H$$

$\square$

5.3 Proving Separation

In this section, the take-separation property of the kernel for interdomain communication (pictured in Figure 4) is verified in the proof of Theorem 6 below. Before proceeding to that verification, we must first elaborate on what is meant by "thread" and "kernel state." Definition 7 exhibits the structure of any thread running on this kernel. Thread structure is determined by the fact that threads are formed from those Re-computations in the range of $\mathcal{P}_d[\![-]\!]$ and it allows the case analysis of thread computations. Because of Haskell's lazy semantics, there are expressions of type $System$ that do not sensibly correspond to any kernel state and these are excluded in the definition of *system configuration* below. Lemmas 1, 2, and 3 aid the verification of Theorem 6.

Recall that in Section 4.1, we defined the notion of thread as those elements in the range of the process semantics $\mathcal{P}_d[\![-]\!]$ and their "tails." Definition 7 refines this notion for the system for interdomain communication, exhibiting the forms that such thread computations in Re may take. Any process denotation is a thread (as in items (i)-(iii) below) as is any "tail" of a process denotation (as in items (iv) and (v) below).

Definition 7 (Thread Cases)**.** *A* thread *is a computation in Re() equal to one of the following:*

$$
\begin{array}{ll}
(i) & step_d\,(\mathcal{V}_d[\![e]\!] \star_{\mathrm{K}} setloc_d\ x\ ) >\!\!>_{\mathrm{Re}} \mathcal{P}_d[\![p]\!] \\
(ii) & step_d\,(getloc_d\, x) \star_{\mathrm{Re}} (signull_d \circ Bcst) >\!\!>_{\mathrm{Re}} \mathcal{P}_d[\![p]\!] \\
(iii) & (signull_d\,(Bcst\, m)) >\!\!>_{\mathrm{Re}} \mathcal{P}_d[\![p]\!] \\
(iv) & step_d\,(setloc_d\ x\ m) >\!\!>_{\mathrm{Re}} \mathcal{P}_d[\![p]\!] \\
(v) & (signal_d\, Rcv) \star_{\mathrm{Re}} \lambda(Msg\, m).\ step_d(setloc_d\ x\ m) >\!\!>_{\mathrm{Re}} \mathcal{P}_d[\![p]\!]
\end{array}
$$

for some process p, location x, expression e, and integer m.

Because the semantics of our metalanguage Haskell allows partial and infinite values, we must formulate precisely which values of type $System$ constitute valid descriptions of the kernel state—such valid descriptions are referred to hereafter as *system configurations* (or simply *configurations*). An expression, $(t, l, h) : System$, is a system configuration when t is a finite list of threads and l and h are finite lists of non$-\bot$ values of type Int. Furthermore, the lists t, h, and l must be fully defined, meaning that, within each "cons" application $(x{::}xs)$ in t, h, and l, neither x nor xs are $\bot$. Consider the system execution $rr\ (t_0, l_0, h_0)$ where (t_0, l_0, h_0) is a system configuration. It is apparent from Definition 7 and the definition of rr that, in any subsequent corecursive call $rr\ (t_i, l_i, h_i)$, (t_i, l_i, h_i) is also a system configuration.

The system execution, $rr\ sys_0$, for configuration $sys_0 = (t_0, l_0, h_0)$ describes an infinite sequence of those system configurations arising from the calls to rr:

$$(t_0,l_0,h_0) \xrightarrow{\ \Delta\ } (t_1,l_1,h_1) \xrightarrow{\ \Delta\ } (t_2,l_2,h_2) \cdots$$

Here, (t_i, l_i, h_i) is the argument to rr in its i^{th} corecursive call within the system execution $rr\ sys_0$. The function, Δ, may be extended to a transition function on the set of all system configurations.

Within the sequence $\{\Delta^i sys_0\}$, consider the configurations that are preceded by a "Lo" configuration (i.e., a configuration in which the next thread to be executed by rr is in the Lo domain). They form a subsequence of $\{\Delta^i sys_0\}$ described by the partial transition function, Δ_L:

$$
\begin{array}{ll}
\Delta_L^0\ sys_0 & = sys_0 \\
\Delta_L^{(n+1)}\ sys_0 & = \Delta^m\ (\Delta_L^n\ sys_0)
\end{array}
$$

where $m > 0$ is the least integer such that, the next thread to be executed in the previous configuration, $\Delta^{(m-1)}(\Delta_L^n sys_0)$, is in the Lo domain. Note that such an m will always exist for $(rr\ sys_0)$ whenever $sys{\downarrow}_{\mathsf{Lo}} \neq \emptyset$. Here, the f^n notation stands for the iterated composition of f: $f^{(i+1)} = f \circ f^i$ and $f^0 = id$.

Lemma 1 relates the transition function Δ_{L} to $take_{L}$ in a form similar to Equation (2). In the computation "$take_{L}\ 1\ (rr\ sys) \star_{R} (take_{L}\ n)$", the "tail" of the system execution $(rr\ sys)$ is passed to "$(take_{L}\ n)$" by the $\star$ operator. Lemma 1 replaces this implicit argument passing with an explicit transition using Δ_{L}.

Lemma 1. *For any system configuration sys such that $sys{\downarrow}_{\mathsf{Lo}} \neq \emptyset$,*

$$take_{L}\ (n{+}1)\ (rr\ sys) = \ take_{L}\ 1\ (rr\ sys) >>_{R} (take_{L}\ n\ (rr\ (\Delta_{L}\ sys)))$$

Proof. Let sys be a system configuration such that $sys{\downarrow}_{\mathsf{Lo}} \ \neq\ \emptyset$, then

$$\begin{array}{l} take_{L}\ (n+1)\ (rr\ sys)) \\ \{\text{Equation (2)}\} \\ \quad = \ take_{L}\ 1\ (rr\ sys) \star_{R} (take_{L}\ n) \\ \{\text{eta-expansion}\} \\ \quad = \ take_{L}\ 1\ (rr\ sys) \star_{R} \lambda\kappa.\ (take_{L}\ n\ \kappa) \end{array}$$

We know from the definitions of rr and $take_{L}$ that $take_{L}\ 1\ (rr\ sys)$ has the form:

$$Pause_{H}(\varphi_1 \star \lambda v_1.\ \eta\ (\ldots Pause_{H}(\varphi_n \star \lambda v_n.\ \eta\ (Pause_{L}\ (\varphi_l \star \lambda v_l.\ \eta\ (Done\ (rr\ s))))) \ldots))$$

for some system configuration $s = \Delta^{(n+1)}sys$. Note that the configuration preceding s, $\Delta^n sys$, was a Lo-configuration—i.e., it produced the Lo-action φ_l. Note further that each preceding configuration (i.e., those in $\{\Delta^i sys \mid 0 \leq i < n\}$) is a Hi-configuration. So, $(n{+}1) > 0$ is the least number such that the predecessor of s is a Lo-configuration; i.e., $s = \Delta_{L} sys$. Continuing with the proof:

$$\begin{array}{l} \{\text{defn. } \star_{R} \text{ and } \eta v \star f \ = \ \eta v >> f v\} \\ \quad = \ take_{L}\ 1\ (rr\ sys) \star_{R} \lambda\_.\ (take_{L}\ n\ (rr\ s)) \\ \quad = \ take_{L}\ 1\ (rr\ sys) >>_{R} (take_{L}\ n\ (rr\ (\Delta_{L}\ sys))) \end{array}$$

□

Lemma 2 asserts that the execution of Lo-threads is independent of the initial contents of the Hi-message queue in a system configuration. In other words, such Lo-system executions are "parametric" with respect to the Hi-queue.

Lemma 2. *Let $s = (w, l, h)$ and $s' = (w, l, h')$ be any two system configurations containing only* Lo*-processes and differing, if at all, only in the* Hi*-message queue component. Then, $rr\ s \ = \ rr\ s'$.*

Proof. This proof uses the approximation lemma for basic resumptions (Theorem 4). Theorem 4 applies to the non-"security-conscious" version of basic resumptions—i.e., the monad with the single pause $Pause$ rather than the high and low pauses $Pause_{H}$ and $Pause_{L}$. However, because s and s' contain only Lo-processes, the resulting schedulings $(rr\ s)$ and $(rr\ s')$ may be mapped injectively into the single-pause monad; call this injection ι. The equality demonstrated in the single-pause setting, $\iota\ (rr\ s) \ = \ \iota\ (rr\ s')$, reflects back to the security-conscious setting due to the injection, $(rr\ s) = (rr\ s')$. Therefore, we may assume in this instance without loss of generality that the approximation lemma applies directly to the security-conscious setting.

This argument proceeds by induction on the approximation and by case analysis of the next thread to be executed. Consider $n = 0$, then

$$approx\ 0\ (rr\ s) \ = \ \perp \ = \ approx\ 0\ (rr\ s')$$

For $n = k{+}1$, assume without loss of generality that $w = (t{::}ws)$ and that

$$t = step_{L}\ (\mathcal{V}_{L}[\![e]\!] \star_{K} setloc_{L}\ x) >\!>_{R} \mathcal{P}_{L}[\![p]\!]$$

for some process p, location x, expression e, and integer m. Please note that $step_{L}$ belongs to the R monad rather than to the Re monad. We prove this case only; the other thread cases are analogous. Assuming $w = (t{::}ws)$:

$$\begin{aligned}
&approx\ (k{+}1)\ (rr\ (t{::}ws, l, h))\\
&\quad = step_{L}\ (\mathcal{V}_{L}[\![e]\!] \star_{K} setloc_{L}\ x) >\!>_{R} approx\ k\ (rr\ (ws {\oplus} \mathcal{P}_{L}[\![p]\!], l, h))\\
&\quad = step_{L}\ (\mathcal{V}_{L}[\![e]\!] \star_{K} setloc_{L}\ x) >\!>_{R} approx\ k\ (rr\ (ws {\oplus} \mathcal{P}_{L}[\![p]\!], l, h'))\\
&\quad = approx\ (k+1)\ (rr\ (w, l, h'))
\end{aligned}$$

□

A consequence of Lemma 2 is that, for any configuration s such that $s{\downarrow_{\mathsf{Lo}}} \neq \emptyset$, $rr\ (\Delta_{L}\ (s{\downarrow_{\mathsf{Lo}}})) = rr\ ((\Delta_{L}\ s){\downarrow_{\mathsf{Lo}}})$. Note that $\Delta_{L}\ (s{\downarrow_{\mathsf{Lo}}})$ and $(\Delta_{L}\ s){\downarrow_{\mathsf{Lo}}}$ only differ, if at all, in the Hi-message queue in the third component, because it is set to $[]$ in $(\Delta_{L}\ s){\downarrow_{\mathsf{Lo}}}$. Their wait queues are identical because rr, and hence Δ_{L}, maintains the relative position of Lo-threads in the queue irrespective of the presence of Hi-threads. This, in turn, implies that their Lo-message queues in the second component are also identical because only Lo-threads affect the Lo-queue in the same order and with identical messages in both $rr\ (\Delta_{L}\ (s{\downarrow_{\mathsf{Lo}}}))$ and $rr\ ((\Delta_{L}\ s){\downarrow_{\mathsf{Lo}}})$.

Lemma 3 shows how rr, run, $(take_{L}\ 1)$, and $mask_{H}$ interact. It is the $n = 1$ case for the proof of take separation in Theorem 6 below.

Lemma 3. *Given a system configuration $((t{::}ts), l, h)$, the following computation,*

$$run\ (take_{L}\ 1\ (rr\ (t{::}ts, l, h))) >\!>_{K} mask_{H}$$

may be simplified according to the form of thread t (i.e., cases (i)-(v) of Definition 7). The cases (i)-(v) are:

(i) $t \ = \ step_{d}\ (\mathcal{V}_{d}[\![e]\!] \star_{K} setloc_{d}\ x) >\!>_{R} \mathcal{P}_{d}[\![p]\!]$, *then*

$$run\ (take_{L}\ 1\ (rr\ (t{::}ts, l, h))) >\!>_{K} mask_{H} \ =$$
$$\begin{cases} (d{=}H) & (\mathcal{V}_{H}[\![e]\!] \star_{K} setloc_{H}\ l\ ) >\!>_{K} run\ (take_{L}\ 1\ (rr(ts {\oplus} \mathcal{P}_{H}[\![p]\!], l, h))) >\!>_{K} mask_{H}\\ (d{=}L) & (\mathcal{V}_{L}[\![e]\!] \star_{K} setloc_{L}\ l\ ) >\!>_{K} mask_{H}\end{cases}$$

(*ii*) $t \;=\; step_{_d}\,(getloc_{_d}\,x) \star_{_R} (signull_{_d}\;\circ\;Bcst) >>_{_R} \mathcal{P}_{_d}[\![p]\!]$, *then*

$$run\;(take_{_L}\,1\;(rr\;(t{::}ts,l,h))) >>_{_K} mask_{_H} \;=\;$$
$$\begin{cases} (d{=}H) & (getloc_{_H}\,x) \star_{_K} \lambda\,m.\;run\;(take_{_L}\,1\;(rr(ts{\oplus}\kappa,l,h))) >>_{_K} mask_{_H} \\ & \quad \text{when }\kappa \;=\; (signull_{_H}\,(Bcst\,m)) >>_{_R} \mathcal{P}_{_H}[\![p]\!] \\ (d{=}L) & (getloc_{_L}\,x) >>_{_K} mask_{_H} \end{cases}$$

(*iii*) $t \;=\; (signull_{_d}\,(Bcst\;m)) >>_{_R} \mathcal{P}_{_d}[\![p]\!]$, *then*

$$run\;(take_{_L}\,1\;(rr\;(t{::}ts,l,h))) >>_{_K} mask_{_H} \;=\;$$
$$\begin{cases} (d{=}H) & run\;(take_{_L}\,1\;(rr\;(ts{\oplus}\mathcal{P}_{_H}[\![p]\!],h{\oplus}m,l))) >>_{_K} mask_{_H} \\ (d{=}L) & mask_{_H} \end{cases}$$

(*iv*) $t \;=\; step_{_d}\,(setloc_{_d}\,x\;m) >>_{_R} \mathcal{P}_{_d}[\![p]\!]$, *then*

$$run\;(take_{_L}\,1\;(rr\;(t{::}ts,l,h))) >>_{_K} mask_{_H} \;=\;$$
$$\begin{cases} (d{=}H) & (setloc_{_H}\,x\;m) >>_{_K} run\;(take_{_L}\,1\;(rr(ts{\oplus}\mathcal{P}_{_H}[\![p]\!],l,h))) >> mask_{_H} \\ (d{=}L) & (setloc_{_L}\,x\;m) >>_{_K} mask_{_H} \end{cases}$$

(*v*) $t \;=\; (signal_{_d}\;Rcv) \star_{_R} \lambda(Msg\;m).\;step_{_d}(setloc_{_d}\,x\;m) >>_{_R} \mathcal{P}_{_d}[\![p]\!]$, *then*

$$run\;(take_{_L}\,1\;(rr\;(t{::}ts,l,h))) >> mask_{_H} \;=\;$$
$$\begin{cases} (d{=}H,h{=}[]) & run\;\;(take_{_L}\;\;1\;\;(rr\;\;(ts{\oplus}t,l,h)))\;\;>>_{_K}\;\;mask_{_H} \\ (d{=}H,h{=}(m{::}hs)) & setloc_{_H}\,x\,m >>_{_K} run\;\;(take_{_L}\;\;1\;\;(rr\;\;(ts{\oplus}t',l,hs)))\;\;>>_{_K}\;\;mask_{_H} \\ & \quad\text{where }t' \;=\; \mathcal{P}_{_H}[\![p]\!] \\ (d{=}L,l{=}[] & mask_{_H} \\ (d{=}L,l{=}(m{::}ls)) & setloc_{_L}\,x\,m\;\;>>_{_K}\;\;mask_{_H} \end{cases}$$

Proof. We demonstrate case (*iii*) for $d = L$ where t is $(signull_{_L}\,(Bcst\;m)) >>_{_R} \mathcal{P}_{_L}[\![p]\!]$. The other cases are completely analogous.

$$run\;\;(take_{_L}\;\;1\;\;(rr\;\;(t{::}ts,l,h)))\;\;>>_{_K}\;\;mask_{_H}$$
$\{$defn. rr and $t{=}P_L(Bcst\,m,\,\lambda\_.\eta_K(\mathcal{P}_L[\![p]\!]))\,\}$
$$= run\;(take_{_L}\,1\;(Pause_{_L}(\eta_{_K}(\mathcal{P}_{_L}[\![p]\!]) \star_{_K}\lambda\kappa.\;\eta_{_K}\,(rr\;(ts{\oplus}\kappa,\,l{\oplus}m,\,h{\oplus}m))))) >>_{_K} mask_{_H}$$

Proof of Lemma 3 continued:

$\{$left unit$\}$
$$= run\;(take_{_L}\,1\;(Pause_{_L}(\eta_{_K}\,(rr\;(ts{\oplus}\mathcal{P}_{_L}[\![p]\!],\,l{\oplus}m,\,h{\oplus}m))))) >>_{_K} mask_{_H}$$
$\{$defn. $take_L\}$
$$= run\;(Pause_{_L}(\eta_{_K}\,(rr\;(ts{\oplus}\mathcal{P}_{_L}[\![p]\!],\,l{\oplus}m,\,h{\oplus}m)) \star_{_K} (\eta_{_K}\;\circ\;take_{_L}\,0))) >>_{_K} mask_{_H}$$
$\{$left unit$\}$
$$= run\;(Pause_{_L}(\eta_{_K}\,(take_{_L}\,0\;(rr\;(ts{\oplus}\mathcal{P}_{_L}[\![p]\!],\,l{\oplus}m,\,h{\oplus}m))))) >>_{_K} mask_{_H}$$
$\{$defn. $take_L\}$

$$= run\ (Pause_L(\eta_K\ (Done\ (rr\ (ts{\oplus}\mathcal{P}_L[\![p]\!], l{\oplus}m, h{\oplus}m))))) \gg_K mask_H$$

$\{$defn. $run\}$

$$= (\eta_K\ (Done\ (rr\ (ts{\oplus}\mathcal{P}_L[\![p]\!], l{\oplus}m, h{\oplus}m))) \star_K run) \gg_K mask_H$$

$\{$left unit$\}$

$$= run\ (Done\ (rr\ (ts{\oplus}\mathcal{P}_L[\![p]\!], l{\oplus}m, h{\oplus}m))) \gg_K mask_H$$

$\{$defn. $run\}$

$$= \eta_K\ (rr\ (ts{\oplus}\mathcal{P}_L[\![p]\!], l{\oplus}m, h{\oplus}m)) \gg_K mask_H$$

$\{$left unit$\}$

$$=\ mask_H$$

□

Lemma 3 establishes the pattern that the proof of take separation will follow in Theorem 6 below. A consequence of Lemma 3 is that, if $sys{\downarrow}_{\mathsf{Lo}} \neq \emptyset$, then:

$$run\ (take_L\ 1\ (rr\ sys)) \gg_K mask_H\ =\ run\ (take_L\ 1\ (rr\ sys{\downarrow}_{\mathsf{Lo}})) \gg_K mask_H \quad (3)$$

Because $sys{\downarrow}_{\mathsf{Lo}} \neq \emptyset$, the l.h.s. will evaluate to a finite sequence of Hi-actions followed by a single Lo-action and $mask_H$:

$$h_1 \gg_K \cdots \gg_K h_n \gg_K l \gg_K mask_H$$

$\{mask_H \# l\}$

$$= h_1 \gg_K \cdots \gg_K h_n \gg_K mask_H \gg_K l$$

$\{$(clobber), n times$\}$

$$= mask_H \gg_K l$$

$\{mask_H \# l\}$

$$=\ l \gg_K mask_H$$

Here, for the sake of readability and without loss of generality, we ignore the cases where the Hi-action returns a value (i.e., where "$h_i \gg$" should be "$h_i \star \lambda v_i.$"). The proof of this equality formalizes the intuition described in Section 3.1. Interactions between effects, governed by the monadic constructions themselves, allow the Hi-actions h_i to clobbered by $mask_H$.

Theorem 6 (Interdomain Communication Kernel has Take-Separation)**.** *The kernel defined in Figure 5.1 has the take-separation property. That is, for every system configuration, sys, such that $sys{\downarrow}_{\mathsf{Lo}} \neq \emptyset$, $rr\ sys\ \equiv_{te}\ rr\ sys{\downarrow}_{\mathsf{Lo}}$.*

Proof. The take equivalence to be demonstrated in this theorem is:

$$run\ (take_L\ k\ (rr\ sys)) \gg_K mask_H\ =\ run\ (take_L\ k\ (rr\ sys{\downarrow}_{\mathsf{Lo}})) \gg_K mask_H$$

for each $k < \omega$. This is proved by induction on k.

Base Case: $k\ =\ 0$.

$$run\ (take_L\ 0\ (rr\ sys)) \gg mask_H$$

$\{$defn. $take_L\}$

$$=\ run\ (Done\ (rr\ sys)) \gg mask_H$$

{defn. run}

$= \eta_K\,(rr\ sys)) >> mask_H$

{left unit}

$= mask_H$

{left unit}

$= \eta_K\,(rr\ sys{\downarrow_{\mathsf{Lo}}})) >> mask_H$

{defn. run}

$= run\ (Done\ (rr\ sys{\downarrow_{\mathsf{Lo}}})) >> mask_H$

{defn. $take_L$}

$= run\ (take_L\ 0\ (rr\ sys{\downarrow_{\mathsf{Lo}}})) >> mask_H$

Inductive Case: $k\ =\ n+1$.

$run\ (take_L\ (n+1)\ (rr\ sys)) >> mask_H$

{Equation (2)}

$= run\ ((take_L\ 1\ (rr\ sys)) \star_R (take_L\ n)) >>_K mask_H$

{Lemma 1}

$= run\ ((take_L\ 1\ (rr\ sys)) >>_R (take_L\ n\ (rr\ (\Delta_L\ sys)))) >>_K mask_H$

{Equation (1)}

$= run\ (take_L\ 1\ (rr\ sys)) >>_K run\ (take_L\ n\ (rr(\Delta_L\ sys))) >>_K mask_H$

{Ind. Hyp.}

$= run\ (take_L\ 1\ (rr\ sys)) >>_K run\ (take_L\ n\ (rr((\Delta_L\ sys){\downarrow_{\mathsf{Lo}}}))) >>_K mask_H$

{Lemma 2}

$= run\ (take_L\ 1\ (rr\ sys)) >>_K run\ (take_L\ n\ (rr(\Delta_L\ (sys{\downarrow_{\mathsf{Lo}}})))) >>_K mask_H$

Now, because $run\ (take_L\ n\ (rr\ (\Delta_L\ (sys{\downarrow_{\mathsf{Lo}}}))))$ only includes Lo state actions, it commutes with $mask_H$ by atomic non-interference. Therefore, we can continue:

$= run\ (take_L\ 1\ (rr\ sys)) >>_K mask_H >>_K run\ (take_L\ n\ (rr\ (\Delta_L\ (sys{\downarrow_{\mathsf{Lo}}}))))$

{Equation (3)}

$= run\ (take_L\ 1\ (rr\ (sys{\downarrow_{\mathsf{Lo}}}))) >>_K mask_H >>_K run\ (take_L\ n\ (rr\ (\Delta_L\ (sys{\downarrow_{\mathsf{Lo}}}))))$

{atomic n.i.}

$= run\ (take_L\ 1\ (rr\ sys{\downarrow_{\mathsf{Lo}}})) >>_K run\ (take_L\ n\ (rr\ (\Delta_L\ (sys{\downarrow_{\mathsf{Lo}}})))) >>_K mask_H$

{Equation (1)}

$= run\ (take_L\ 1\ (rr\ sys{\downarrow_{\mathsf{Lo}}}) >>_R take_L\ n\ (rr\ (\Delta_L\ (sys{\downarrow_{\mathsf{Lo}}})))) >>_K mask_H$

{Lemma 1}

$= run\ (take_L\ (n{+}1)\ (rr\ sys{\downarrow_{\mathsf{Lo}}})) >>_K mask_H$

$\square$

6 Achieving Scalability

How are typical operating system behaviors (e.g., process creation, preemption, synchronization, etc.) achieved in this layered monadic setting and what impact, if any, do such enhancements to functionality have on the security verification? These are questions to which it is difficult to give final, definitive answers; however, by considering an example, one can get some indication as to what the relevant concerns are. This section considers such an extension—a process creation primitive called `dupl`—to the interdomain communication kernel of the previous section.

As it turns out, this additional functionality requires *no* change to the existing resumption monadic framework and has little impact on the security verification. The impact it does have on the verification is as limited as one could reasonably hope for and boils down simply to considering an extra case corresponding to the added functionality in each of the various proofs and definitions of Section 5.3. That is to say, the verified properties in Section 5.3 required no re-verification at all. This modularity and scalability in the verification would seem to arise from the fact that the new functionality is an orthogonal concern to the security property. That is, because the added functionality does not result in interdomain information flow, it has no impact on the system-wide security.

6.1 Point 3: Standard Services.

This section summarizes the necessary changes to the kernel from Section 5.1 required to add an intradomain service—in this case, a process creation primitive. The changes are quite minimal. First, add an additional request *Dupl* to *Req*; the *Rsp* type remains unchanged as the response to a process creation request will be Ack.

$$\mathbf{data}\; Req' \;=\; Cont \mid Bcst\; Int \mid Rcv \mid Dupl$$

Implicitly, this change to *Req* is actually a refinement to the reactive resumption monad transformer, but we assume now that $Re\ a\ = ReactT\ Req'\ Rsp\ K\ a$.

The only change to the kernel code is an additional clause for both $\mathcal{E}_d[\![-]\!]$ and rr [23]. The meaning of `dupl` is simply to signal the kernel with a *Dupl* request, and so, to define the `dupl` event, add the following clause to $\mathcal{E}_d[\![-]\!]$:

$$\mathcal{E}_d[\![\texttt{dupl}]\!] = signull_d\ Dupl$$

The corresponding kernel action simply duplicates the signaling thread within the thread list. So, the kernel rr is extended with the clause:

$$\begin{array}{l} rr\ (((P_d(Dupl, r){::}ts), l, h) = next_d\ (ts{+\!\!+}[r \bullet_d Ack, r \bullet_d Ack], l, h) \\ \quad \mathbf{where} \\ \quad\quad (\bullet_d) \quad :: \ (Rsp \to K\ (Re\ a)) \to Rsp \to Re\ a \\ \quad\quad r \bullet_d s = P_d(Cont, \lambda Ack.\ r\ s) \end{array}$$

$(r \bullet_d s)$ passes the response signal s to the "continuation" r; that is, r is the second component in an *Re* computation $P_d(q, r)$.

Impact on Security Verification. Let us now consider what impact this extension to the kernel functionality has on its security verification—in other words, what changes must occur to the proof in Section 5.3 to accomplish the "re-verification" of the kernel. This functional extension has very little impact on the re-verification effort, at least in part, because such functionality is orthogonal to the security of the system.

The changes to the proof in Section 5.3 are as follows. Definition 7 is extended with a new clause characterizing threads with `dupl` events:

$$(vi) \qquad (signull_{d}\ Dupl) >>_{Re} \mathcal{P}_{d}[\![p]\!] \text{ for some process } p$$

The definitions of the transition functions, Δ and Δ_{L}, remain the same as does the statement and proof of Lemma 1. The statement of Lemma 2 is unchanged, although the case when the next thread is of the form (vi) above must also be considered. The statement of Lemma 3 is extended to cover the new threads:

$$(vi)\ \ t\ =\ (signull_{d}\ Dupl) >>_{R} \mathcal{P}_{d}[\![p]\!], \text{then}$$
$$run\ (take_{L}\ 1\ (rr\ (t{::}ts, l, h))) >> mask_{H}\ =$$
$$\begin{cases} (d{=}H) & run\ (take_{L}\ 1\ (rr\ (ts{+\!\!+}[\mathcal{P}_{H}[\![p]\!], \mathcal{P}_{H}[\![p]\!]], l, h{\oplus}m))) >> mask_{H} \\ (d{=}L) & mask_{H} \end{cases}$$

The existing proof for parts (i)-(v) of Lemma 3 remains the same; the proof for (vi) is almost identical to that of case (iii) shown in Section 5.3. The proof of Theorem 6 is unchanged, because it relies on the monad laws, properties of run and $take_{L}$, atomic non-interference, and Lemmas 1-3.

Scalability. One advantage of structuring by monads and monad transformers is the extensibility of the resulting specifications. Adding additional domains and security levels or enhancing system functionality are manifested as refinements to the monad transformers underlying the system construction. To construct a system with n separated domains, one extends the monad transformers $ResT$ and $ReactT$ with n "pause" constructors each. If these n domains correspond to security levels represented as a lattice [5], the corresponding take and mask functions, "$take_i$" and "$mask_i$" must extract and clobber all events with security level j, where $j \sqsubseteq i$ in the security lattice.

7 Conclusion

Type constructions and their properties are the foundation of this approach to language-based security; this is fundamentally different from approaches based on information flow control via type checking. The approach reflects the semantic foundations of effects and effect interaction into a pure functional language in which provably separable computations can be constructed. At the same time, it allows explicit regions of the program in which the type system does not, by itself, guarantee separation. In the monadic approach it is clear from the type construction when information flow separation is established and when it is established by reasoning about program behavior.

This approach can be used either for direct implementation or as a modeling language. As a modeling language, these techniques can explain the effect separation provided by unprivileged execution modes in hardware, while at the same time modeling the potential interference of privileged execution. As an implementation language it provides, through the type constructions, ways to construct programs that achieve information flow separation. In this sense this work is similar to language-based security mechanisms based on type checking. However, such approaches are domain-specific

(a) Example Threads

```
// Broadcaster
brc::
 x=100;
 loop { x=x+1 ;
        bcast(x) }
// Receiver
rcv::
 loop { recv(x) }
```

(b) `brc` in Lo, `rcv` in Hi

```
broadcasting : 101
broadcasting : 102
 receiving : 101
broadcasting : 103
 receiving : 102
broadcasting : 104
 receiving : 103
        ⋮
```

(c) `brc` in Hi, `rcv` in Lo

```
broadcasting : 101
broadcasting : 102
broadcasting : 103
broadcasting : 104
broadcasting : 105
broadcasting : 106
broadcasting : 107
        ⋮
```

Figure 5: *Formal system models are executable.* The specifications developed here may be directly & faithfully realized in Haskell. Part (a) defines the "broadcaster" and "receiver" threads `brc` and `rcv` that generate an infinite number of broadcast and receive requests. Part (b) shows `brc` executing in Lo and `rcv` executing in Hi, while part (c) shows `rcv` in Lo and `brc` in Hi. The Haskell code has been instrumented to print out broadcast and receive events. N.B., Lo-generated broadcasts are received in the Hi domain in (b), while in (c), Hi-generated broadcasts are not received in Lo, illustrating the domain separation.

extensions of type systems to express information flow properties; the monadic approach uses concepts easily expressed in existing type systems for pure higher-order languages.

We have not explored the formal relationship between domain-specific type systems for information flow and monads. We suspect that in some cases it may be possible to prove the soundness of information flow extensions to other languages by embedding them into the monadic type systems presented here. This may be of particular interest when applied to recent enhancements to information flow type systems that allow for policy enabled downgrading functions to be defined.

Confidentiality and integrity concerns within the setting of shared-state concurrency are really about controlling interference and interaction between threads. It is a natural and compelling idea, therefore, to apply the mathematics of effects—monads—to this problem as monads provide precise control of such effects. In fact, layering monads—i.e., modularly constructing monads with monad transformers—yields fine-grained control of effects and their interactions. This paper demonstrates how the fine-grained tailoring of effects possible with monad transformers promotes integrity and information security concerns. As a proof of concept, we showed that a classic design in computer security (the separation kernel of Rushby [64]) can be realized and verified in a straightforward manner.

Monads with state constructed via multiple applications of the state monad transformer delimit the scope of imperative effects by construction, and this fact is expressed as atomic noninterference. Using this insight, we were able to construct and verify several separation kernel specifications of increasing and non-trivial functionality. There are a number of benefits arising from structuring these kernels with monad transformers. (1) The specifications are easily extended. Monad transformers have proved their usefulness in the construction of modular interpreters and compilers [35, 24], and the kernel refinements in Figure 2 are modular in precisely the same manner. Enhancing system functionality means refining the monad transformers. (2) It is also significant that the *verification* of these kernels share the benefits of modularity and extensibility in that the impact of the kernel refinements was minimal. As functionality was added to the kernels, no significant re-verification was required. (3) Formal models of security are sometimes difficult to relate to actual programs or systems; the separation kernel specifications presented here, being monadic, are readily implemented in a higher-order, functional programming language like Haskell (see Figure 5).

The separation kernel example illustrates the usefulness of monad transformers as a tool for formal methods. A number of very useful properties came by construction because the state monad transformer gives rise to modular theories of effects. Monad transformers have proved their usefulness for modularizing interpreters and compilers [35, 24], resulting in modular components from which systems can be created; the three separation kernels (i.e., Points 1-3 in Figure 2) are modular in precisely the same sense.

Acknowledgments

The authors wish to thank René Rydhof Hansen, Tom Harke, Peter White and the anonymous CSFW reviewers for suggestions that helped improve this paper and its presentation.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proceedings of the Twenty-sixth ACM Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 147–160, January 1999.

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

[3] M. Archer and C. Heitmeyer. TAME: A specialized specification and verification system for timed automata. In Azer Bestavros, editor, *Work In Progress (WIP) Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*, pages 3–6, Washington, DC, 1996.

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

[5] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corp., Bedford MA, May 1973.

[6] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[7] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall Series in Computer Science. Prentice-Hall Europe, London, UK, 2^{nd} edition, 1998.

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

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

[10] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15(2), March 2005.

[11] D. Denning and P. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[12] D. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[13] A. Filinski. Representing layered monads. In *Proceedings of the 26st ACM Symposium on Principles of Programming Languages (POPL)*, pages 175–188, 1999.

[14] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*, pages 186–197, 2004.

[15] R. Giacobazzi and I. Mastroeni. Adjoining declassification and attack models by abstract interpretation. In *European Symposium on Programming (ESOP'05).*, volume 3444 of *Lecture Notes in Computer Science*, pages 295–310. Springer-Verlag, 2005.

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

[17] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 Symposium on Security and Privacy (SSP '82)*, pages 11–20, Los Alamitos, Ca., USA, April 1990. IEEE Computer Society Press.

[18] D. Greve, R. Richards, and M. Wilding. A Summary of Intrinsic Partitioning Verification. In *Fifth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2004)*, November 2004.

[19] D. Greve, M. Wilding, and W. M. Vanfleet. A Separation Kernel Formal Security Policy. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*, July 2003.

[20] C. Gunter. *Semantics of Programming Languages: Programming Techniques*. The MIT Press, Cambridge, Massachusetts, 1992.

[21] 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.

[22] R. Harper, P. Lee, and F. Pfenning. The Fox project: Advanced language technology for extensible systems. Technical Report CMU-CS-98-107, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1998.

[23] W. Harrison. Haskell implementations of the monadic separation kernels for CSFW 2005. Available from `www.cs.missouri.edu/~harrison/csfw05`.

[24] W. Harrison. *Modular Compilers and Their Correctness Proofs*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[25] W. Harrison. The essence of multitasking. In *11th International Conference on Algebraic Methodology and Software Technology (AMAST 2006)*, pages 158–172, July 2006.

[26] W. Harrison, G. Allwein, A. Gill, and A. Procter. Asynchronous exceptions as an effect. In *Proceedings of the 9th International Conference on the Mathematics of Program Construction (MPC08)*, volume 5133 of *LNCS*, pages 153–176, 2008.

[27] 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.

[28] N. Heintze and J. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the Twenty-fifth ACM Symposium on Principles of Programming Languages*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[29] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 346–355, New York, NY, USA, 2006. ACM Press.

[30] B. Huffman, J. Matthews, and P. White. Axiomatic constructor classes in isabelle/holcf. In *18th International Conference on Theorem Proving in Higher Order Logics (TPHOL05)*, volume 3603 of *LNCS*, pages 147–162. Springer Verlag, 2005.

[31] R. Joshi and K. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, May 2000.

[32] M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Softw. Eng.*, 23(4):203–213, 1997.

[33] B. Lampson. A note on the confinement problem. In *Communications of the ACM*, pages 613–615. ACM press, October 1973.

[34] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 158–170, 2005.

[35] S. Liang. *Modular Monadic Semantics and Compilation*. PhD thesis, Yale University, 1998.

[36] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995*, pages 333–343. ACM Press, 1995.

[37] S. Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag, 1971.

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

[39] W. Martin, P. White, F. S. Taylor, and A. Goldberg. Formal construction of the mathematically analyzed separation kernel. In *ASE '00: Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, page 133, Washington, DC, USA, 2000. IEEE Computer Society.

[40] E. McCauley and P. Drongowski. KSOS—the design of a secure operating system. In *Proceedings of the American Federation of Information Processing Societies (AFIPS) National Computer Conference*, volume 48, pages 345–353, 1979.

[41] D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symposium on Security and Privacy*, pages 177–187, 1988.

[42] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.

[43] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[44] M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.

[45] E. Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[46] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ., 1990.

[47] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[48] P. Neumann, R. Boyer, R. Feiertag, K. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proof. Technical Report CSL-116, SRI, May 1980.

[49] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[50] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 268–280. ACM Press, 2004.

[51] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Proc. of 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer–Verlag.

[52] J. Palsberg and P. Ørbæk. Trust in the lambda-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997.

[53] N. Papaspyrou. A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. In *Proceedings of the 3rd Panhellenic Logic Symposium*, 2001. An expanded technical report is available from the author by request.

[54] S. Peyton Jones, editor. *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, 2003.

[55] G. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3), 1976.

[56] F. Pottier and S. Conchon. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP 99)*, pages 46–57, 2000.

[57] F. Pottier and V. Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 319–330, Portland, Oregon, January 2002.

[58] Programatica Home Page. `www.cse.ogi.edu/PacSoft/projects/programatica`.

[59] J. Reynolds. *The Craft of Programming*. Prentice Hall, Englewood Cliffs, 1981.

[60] J. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North-Holland, 1983.

[61] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, 2002.

[62] W. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[63] J. Rushby. Design and verification of secure systems. In *Proceedings of the ACM Symposium on Operating System Principles*, volume 15, pages 12–21, 1981.

[64] J. Rushby. Proof of separability: A verification technique for a class of security kernels. In *Proceedings of the* 5^{th} *International Symposium on Programming*, pages 352–362, Berlin, 1982. Springer-Verlag.

[65] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[66] D. Schmidt. *Denotational Semantics*. Allyn and Bacon, Boston, 1986.

[67] J. Shapiro, J. Smith, and D. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 170–185, Charleston, South Carolina, December 1999.

[68] G. Smith. A new type system for secure information flow. In *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 115–125. IEEE Computer Society Press, June 2001.

[69] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages, San Diego, California*, pages 355–364, New York, January 1998.

[70] SPECWARE Home Page. `http://www.specware.org/`.

[71] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press, Cambridge, Massachusetts, 1977.

[72] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP 04)*, pages 115–125, 2004.

[73] P. Wadler. Monads for functional programming. In *Proceedings of the 1992 Marktoberdorf International Summer School on Logic of Computation*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52, 1995.

[74] Philip Wadler. The Essence of Functional Programming. In *Proceedings of the 19th Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 19 – 22, 1992. ACM Press.

[75] B. Walker, R. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.

[76] A. Zakinthinos and E. Lee. A general theory of security properties. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 94–102. IEEE Computer Society, 1997.

[77] S. Zdancewic and A. Myers. Robust declassification. In *Proceedings of 14th IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia, Canada*, pages 15–23, June 2001.

A Theorems and Proofs

This appendix presents the proofs of Theorems 1-3 of Section 3.

Theorem 1. *Let M be any monad and $M' = StateT\ s'\ M$ with operations η', $\star'$, lift, g', and u' defined by $(StateT\ s')$. Then:*

1. $\langle M', \eta', \star', u', g', s'\rangle$ *is a state monad.*
2. $\langle M, \eta, \star, u, g, s\rangle$ *is a state monad* $\Rightarrow \langle M', \eta', \star', \mathit{lift} \circ u, \mathit{lift}\, g, s\rangle$ *is also.*

Proof. In each of these proofs, we suppress the use of the ST constructor for the sake of readability.

Part *1*. Case: Sequencing.

$$\begin{array}{ll}
\multicolumn{2}{l}{uf >> uf'} \\
\multicolumn{2}{l}{\{\text{def. } \star'\}} \\
= & \lambda\sigma_0.(uf)\sigma_0 \star (\lambda(v,\sigma_1).(\lambda\_.uf')\, v\, \sigma_1) \\
\multicolumn{2}{l}{\{\beta\}} \\
= & \lambda\sigma_0.(uf)\sigma_0 \star (\lambda(v,\sigma_1).(uf')\, \sigma_1) \\
\multicolumn{2}{l}{\{\text{def. } u(\times 2)\}} \\
= & \lambda\sigma_0.(\lambda\sigma.\eta_M((), f\sigma))\sigma_0 \star (\lambda(v,\sigma_1).(\lambda\sigma.\eta_M((), f'\sigma))\, \sigma_1) \\
\multicolumn{2}{l}{\{\beta(\times 2)\}} \\
= & \lambda\sigma_0.(\eta_M((), f\sigma_0)) \star (\lambda(v,\sigma_1).(\eta_M((), f'\sigma_1))) \\
\multicolumn{2}{l}{\{\text{left unit}\}} \\
= & \lambda\sigma_0.\eta_M((), f'(f\sigma_0)) \\
= & \lambda\sigma_0.\eta_M((), (f' \circ f)\, \sigma_0) \\
\multicolumn{2}{l}{\{\text{def. } u\}} \\
= & u(f' \circ f)
\end{array}$$

Case: Cancellation.

$$\begin{array}{ll}
\multicolumn{2}{l}{g >> \varphi} \\
\multicolumn{2}{l}{\{\text{def. } g, \star_M\}} \\
= & \lambda\sigma_0.\eta_M(\sigma_0,\sigma_0) \star_M (\lambda(v,\sigma_1).\, (\lambda\_.\varphi)\, v\, \sigma_1) \\
\multicolumn{2}{l}{\{\beta\}} \\
= & \lambda\sigma_0.\eta_M(\sigma_0,\sigma_0) \star_M (\lambda(v,\sigma_1).\, \varphi\, \sigma_1) \\
\multicolumn{2}{l}{\{\text{left unit}\}} \\
= & \lambda\sigma_0.\, \varphi\, \sigma_0 \\
\multicolumn{2}{l}{\{\textit{eta} \text{ reduction}\}} \\
= & \varphi
\end{array}$$

Part *2*. To show that $\mathit{lift} \circ u$ and $\mathit{lift}\ g$ obey sequencing and cancellation. These follow directly from the lifting laws of Section 3 and from the fact that M is a state monad.

$$\begin{array}{lcl}
\mathit{lift}(uf) >> \mathit{lift}(uf') & = & \mathit{lift}(uf >>_M uf') \\
 & = & \mathit{lift}(u(f' \circ f)) \\
 & & \\
\mathit{lift}(g) >> \mathit{lift}\ \varphi & = & \mathit{lift}(g >>_M \varphi) \\
 & = & \mathit{lift}\ \varphi
\end{array}$$

□

Theorem 2. *Let M be the state monad $\langle M, \eta, \star, u, g, s\rangle$. Let M' be the state monad structure, $\langle StateT\; s'\; M, \eta', \star', u', g', s'\rangle$, defined by $(StateT\; s')$ with operations η', $\star'$, lift, g', and u'. By Theorem 1, M' is also a state monad. Then, for all $f : s \to s$ and $f' : s' \to s'$, $\mathit{lift}(u\,f)\,\#_{M'}\,(u'\,f')$ holds.*

Proof. Below, β^{-1} refers to β-expansion.

$$
\begin{aligned}
& (u'\,f') >>_{M'} \mathit{lift}(u\,f)\\
& \{\text{def. } \star'\}\\
& \quad = \lambda\sigma_0.((u'\,f')\,\sigma_0) \star_M \lambda((), \sigma_1).(\lambda\_.\mathit{lift}(u\,f))\,()\,\sigma_1\\
& \{\beta\}\\
& \quad = \lambda\sigma_0.((u'\,f')\,\sigma_0) \star_M \lambda((), \sigma_1).(\mathit{lift}(u\,f))\,\sigma_1\\
& \{\text{def. } u'\}\\
& \quad = \lambda\sigma_0.((\lambda\sigma.\eta_M((), f'\,\sigma))\,\sigma_0) \star_M \lambda((), \sigma_1).(\mathit{lift}(u\,f))\,\sigma_1\\
& \{\beta\}\\
& \quad = \lambda\sigma_0.(\eta_M((), f'\,\sigma_0)) \star_M \lambda((), \sigma_1).(\mathit{lift}(u\,f))\,\sigma_1\\
& \{\text{left unit}\}\\
& \quad = \lambda\sigma_0.(\mathit{lift}(u\,f))\,(f'\,\sigma_0)\\
& \{\text{def. } \mathit{lift}\}\\
& \quad = \lambda\sigma_0.(\lambda\sigma.(u\,f) \star_M \lambda v.\eta_M(v, \sigma))\,(f'\,\sigma_0)\\
& \{\beta\}\\
& \quad = \lambda\sigma_0.(u\,f) \star_M \lambda v.\eta_M(v, f'\,\sigma_0))\\
& \{\beta^{-1}\}\\
& \quad = \lambda\sigma_0.(u\,f) \star_M \lambda v.(\lambda\sigma.\eta_M(v, f'\,\sigma))\,\sigma_0\\
& \{\text{def. } u'\}\\
& \quad = \lambda\sigma_0.(u\,f) \star_M \lambda v.(u'\,f')\,\sigma_0\\
& \{\beta^{-1}\}\\
& \quad = \lambda\sigma_0.(u\,f) \star_M \lambda v.(\lambda\_.u'\,f')\,v\,\sigma_0\\
& \{\text{right unit}\}\\
& \quad = \lambda\sigma_0.(u\,f \star_M \eta_M) \star_M \lambda v.(\lambda\_.u'\,f')\,v\,\sigma_0\\
& \{\text{ calculation }\}\\
& \quad = \lambda\sigma_0.(\lambda\sigma.(u\,f \star_M \lambda w.\eta_M(w, \sigma))\,\sigma_0 \star_M \lambda(v, \sigma).(\lambda\_.u'\,f')\,v\,\sigma\\
& \{\text{defn. } \mathit{lift}\}\\
& \quad = \lambda\sigma_0.(\mathit{lift}(u\,f))\,\sigma_0 \star_M \lambda(v, \sigma).(\lambda\_.u'\,f')\,v\,\sigma\\
& \{\text{defn. } \star'\}\\
& \quad = \mathit{lift}(u\,f) >>_{M'} u'\,f'
\end{aligned}
$$

□

Theorem 3. *Let M be a monad with operations* $o, p : M\,()$ *such that* $o \#_M p$*. Then,* $(\mathit{lift}\ o)\ \#_{TM}\ (\mathit{lift}\ p)$ *where* T *is a monad transformer and* $(\mathit{lift} : M\ a \rightarrow T\ M\ a)$ *obeys the lifting laws* (*see Section 3*).

Proof. This follows from the lifting laws of Section 3.

$$\begin{array}{rcl} \mathit{lift}\, o >>_{TM} \mathit{lift}\, p & = & \mathit{lift}(o >>_M p) \\ & = & \mathit{lift}(p >>_M o) \\ & = & \mathit{lift}\, p >>_{TM} \mathit{lift}\, o \end{array}$$

□

B Proof of Approximation Lemma for Basic Resumption Computations

In this section, we prove an approximation lemma for resumption monads analogous to the approximation lemma for lists [16]. The statement and proof of the resumption approximation lemma are almost identical to those of the list case; this is perhaps not too surprising because of the analogy between lists and resumptions remarked upon earlier. Assume that, for a given monad M, that R is declared in Haskell as:

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

The Haskell function approx approximates R computations:

$$\begin{array}{l} \mathit{approx}\ :\ \mathit{Int} \rightarrow R\, a \rightarrow R\, a \\ \mathit{approx}\ (n{+}1)\ (\mathit{Done}\ v)\ \ = \mathit{Done}\ v \\ \mathit{approx}\ (n{+}1)\ (\mathit{Pause}\ \varphi) = \mathit{Pause}\ (\varphi \star (\eta\ \circ\ \mathit{approx}\ n)) \end{array}$$

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

Theorem 2. For any $\varphi, \gamma : Ra$, $\varphi = \gamma \Leftrightarrow$ for all $n \in \omega,\ \mathit{approx}\ n\ \varphi = \mathit{approx}\ n\ \gamma$.

To prove this theorem requires a denotational model of R—that is, we need to know precisely what φ, γ, approx, etc., are—and for this we turn to the denotational semantics of resumption computations as developed by Papaspyrou [53]. His semantics for resumption monads applies a categorical technique for constructing denotational models of lazy data types by calculating fixed points of functors [20, 4]. Using this semantics, we show that, for every simple type τ (i.e., a variable-free type such as Int), the approximation lemma holds for $R\,\tau$.

Assume that D and M are a fixed domain and monad[3], respectively. Assume for the sake of this proof that domain D is the model of the simple type τ. Let O denote the trivial domain $\{\bot_{\mathrm{O}}\}$. Then the following defines the functor F:

[3]The functor component of the monad M is required to be *locally continuous* [53], although we make no explicit use of this fact here.

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

F is indeed an endofunctor on the category of domains Dom [53].

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

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

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

Between these approximations of $R\,\tau$, there are useful functions that extend an approximation in $F^i\text{O}$ to an approximation in $F^{i+1}\text{O}$ and truncate an approximation in $F^{i+1}\text{O}$ to one in $F^i\text{O}$; these are defined in terms of the *embedding-projection* functions ι^e and ι^p [20, 66]:

$$\begin{array}{ll} \iota^e : \text{O} \to F\text{O} & \iota^p : F\text{O} \to \text{O} \\ \iota^e\ \bot_{\text{O}} \ = \ \bot_{F\text{O}} & \iota^p x \ = \ \bot_{\text{O}} \end{array}$$

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

The domain for type $R\,\tau$ is formed from the collection of all infinite sequences $(\varphi_n)_{n\in\omega}$ such that $\varphi_i \in F^i\text{O}$. Each component of the domain element $(\varphi_n)_{n\in\omega}$ is an approximation of its successor; this condition is expressed formally using truncation: $\varphi_i = F^i(\iota^p)\ \varphi_{i+1}$. This collection, when ordered pointwise, forms a domain [53].

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

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

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

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

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

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

Given these facts, the proof of the approximation lemma for resumption computations proceeds exactly as that of the list version in Gibbons and Hutton [16]; we repeat this proof for the convenience of the reader.

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

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

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

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

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