Achieving Information Flow Security Through Precise 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

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 (45; 14; 44; 34; 29).

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 (25; 19). This modularity enables clear distinctions to be made in the type system that show exactly what facets of the global effect system a program fragment may impact. 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 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 (11). 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 (8), 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 (31) define concurrent execution (`||`) and, in some formulations, "reactive" programming features (25) such as message passing, synchronization, etc. Monad transformers (25; 19) are monad language "constructors" which add new features to a monad language with each monad transformer application while preserving the behavior of its existing features; such modularly-constructed monads are referred to as *layered* monads.

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

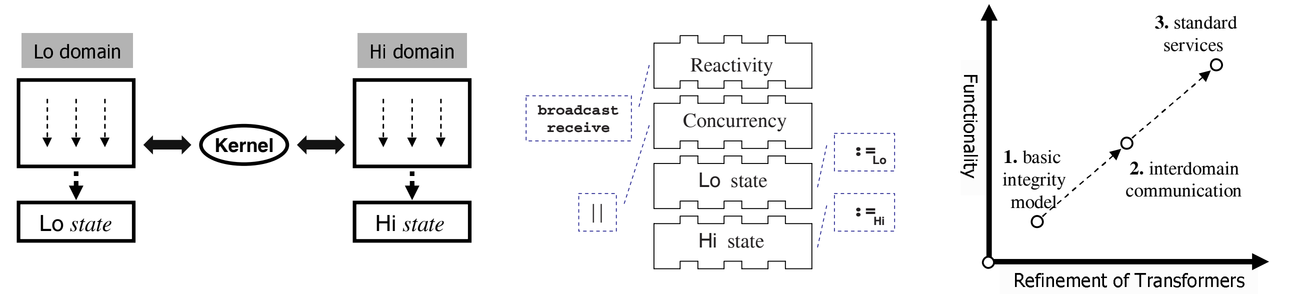

Figure 1. (Left) *Separation Kernel*: Threads within each domain can only access their own state, and all inter-domain communication is mediated by the kernel. The kernel enforces a "no write down" security policy. (Middle) *Layering Monads for Separation*: Combining fine control of stateful effects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

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 (8; 19; 13). 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. The fine-grained control of stateful effects achieved thereby is key to the present approach to secure systems. 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 (41; 40) 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, middle). However, layering effects is more than an implementation technique: properties arise from the underlying structure of layered monads which 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 1 (right) 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 fork system call. 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 (18; 12). Less frequently recognized is their value for formal specification and verification; the impact of the kernel refinements on the verification is minimal.

Section 2 summarizes the background on separation kernels and formulates three process languages corresponding to points (1) through (3) in Figure 1 (right). 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 and intradomain extensions to the basic model of integrity; the section begins with a description of reactivity in monadic form. Section 7 surveys related work. Finally, Section 8 summarizes the present work and outlines future directions.

2. Separation Kernels

A *separation kernel* enforces process isolation by partitioning the state into separate user spaces (Rushby 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 (41; 40), $M = (S, I, O, \mathit{next}, \mathit{inp}, \mathit{out})$, is an abstract machine formally characterizing a multi-user operating system. Here, S, I, and O are finite 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 the input visible to c. 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 $\mathit{res}(i)$ maps *out* onto each state in a computation starting from input i: $\mathit{res}(i) = \langle \mathit{out}(s_0), \mathit{out}(s_1), \ldots\rangle$. Function $\mathit{ext}(c, x)$ projects[1] all of the c-coloured objects from x, so $\mathit{ext}(c, \mathit{res}(i))$ is the trace of all c-outputs in $\mathit{res}(i)$. Function $\mathit{condense}(s)$ removes all "stutters" from s: $\mathit{condense}(\langle 1, 2, 2, 2, 3\rangle) = \langle 1, 2, 3\rangle$. Stuttering may occur because the scheduler represented in *next* is not completely fair, and allowing stuttering introduces the possibility of a timing channel (16) with which the separation property does not attempt to cope. The formal statement of separation security is:

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

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., $\mathit{ext}(c, i) = \mathit{ext}(c, j)$), c produces the same condensed output (i.e., $\mathit{condense}(\mathit{ext}(c, \mathit{res}(i))) = \mathit{condense}(\mathit{ext}(c, \mathit{res}(j)))$).

Separation (both in Rushby's formulation (41) and ours) confronts storage and legitimate (i.e., using system resources to transfer information) channels, but not covert or timing channels (16).

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.

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 1 (right) and is an extension of its predecessor:

— **1.** basic integrity:
Event $=$ *Loc* `:=` *Exp*
— **2.** interdomain communication:
Event $=$ *Loc* `:=` *Exp* | `bcast(`*Loc*`)` | `recv(`*Loc*`)`
— **3.** standard services:
Event $=$ *Loc* `:=` *Exp* | `bcast(`*Loc*`)` | `recv(`*Loc*`)` | `fork`
Exp $=$ *Int* | *Loc* | *Exp* $\oplus$ *Exp*

Each event language has a simple assignment statement, l`:=`e, which evaluates its right-hand side, $e \in \mathit{Exp}$, and stores it in the location, $l \in \mathit{Loc}$, on the left-hand side. Expressions are constants, the contents of a location, or a binary operation. The second and third event languages extend the first with broadcast and receive primitives: `bcast(`l`)` and `recv(`l`)`. The event `bcast(`l`)` broadcasts the contents of location l, while `recv(`l`)` receives an available message in location l. The third language extends the first two with a process forking primitive, `fork`, producing 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 `fork` primitive has no impact on security or integrity concerns at

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

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

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 (25; 18), and readers requiring more should consult the references for further background.

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* operator, $\eta\ :\ a \rightarrow M\ a$, and the *bind* operator, $\star\ :\ M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$. 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.

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 98 (32) although we would be equally justified 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 98 code for these constructions is available online (11). Haskell 98 reverses the standard use of (::) and (:) in that (::) stands for "has type" and (:) for list concatenation in Haskell 98. We will continue to use the standard interpretation of these symbols.

Defining a monad in Haskell typically consists of declaring a data type and an instance declaration of the built-in $Monad$ class (32); note, however, that Haskell does not guarantee that members of $Monad$ obey the monad laws. All of the constructions presented here, however, produce monads (25; 19; 31). The data type declaration defines the computational "raw materials" encapsulated by the monad. The identity monad I, containing no raw materials, and the state monad St, containing a single threaded state Sto, are declared:

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

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

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"):

$$
g\ =\ ST\ (\lambda s.\ (s, s)) \qquad u\ \delta\ =\ ST\ (\lambda\ s.\ ((), \delta\ s))
$$

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

The state monad transformer generalizes the state monad. It takes two type parameters as input—the type constructor m representing an existing monad and a store type s—and from these creates a monad adding single-threaded s-passing to the computational raw material of m. Using the bind and return of m, the new monad $(StateT\ s\ m)$ is defined:

$$
\begin{array}{ll}
\mathbf{data}\ StateT\ s\ m\ a & =\ ST\ (s\ \rightarrow\ m\ (a, s)) \\
deST\ (ST\ x) & = x \\
\eta\ v & = ST\ (\lambda\ s.\ \eta_m\ (v, s)) \\
(ST\ x) \star f & = ST\ (\lambda\ s.\ (x\ s) \star_m \lambda\ (y, s').\ deST\ (f\ y)\ s') \\
lift\ x & = ST\ (\lambda\ s.\ x \star_m \lambda\ v.\ \eta_m\ (v, s))
\end{array}
$$

The bind and return of the input 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, $lift :\ m\ a \rightarrow StateT\ s\ m\ a$, enriches computations in monad m as computations in $StateT\ s\ m$. The non-proper morphisms are redefined as:

$$
\begin{array}{ll}
g\ :\ StateT\ s\ m\ s & u\ :\ (s \rightarrow s) \rightarrow StateT\ s\ m\ () \\
g\ =\ ST\ (\lambda\ s.\ \eta_m\ (s, s)) & u\ \delta\ =\ ST\ (\lambda\ s.\ \eta_m\ ((), \delta\ s))
\end{array}
$$

The morphisms $\star$, η, and *lift* satisfy the *monad laws* (top three) and the *lifting laws* (bottom two) (18):

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

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,

$K \triangleq StateT\ Hi\ (StateT\ Lo\ I)$, where Hi and Lo are fixed types representing the high and low security states in Figure 1 (their exact structure need not concern us yet). The monad K has two states with corresponding update and get operations. The update and get operations corresponding to the Hi state, defined as u_H and g_H, are given by the application of the $(StateT\ Hi)$ transformer; the Hi operations are added to the one-state monad $m = (StateT\ Lo\ I)$ while the Lo operators are lifted from m:

$$
\begin{array}{ll}
u_H : (Hi \to Hi) \to K\,() & u_H\,\delta \triangleq ST\,(\lambda\, h.\, \eta_m\,((), \delta\, h)) \\
g_H : K\ Hi & g_H\,\delta \triangleq ST\,(\lambda\, h.\, \eta_m\,(h, h)) \\
u_L : (Lo \to Lo) \to K\,() & u_L \quad \triangleq \mathit{lift} \circ u_0 \\
g_L : K\ Lo & g_L \quad \triangleq \mathit{lift}\ g_0 \\
 & u_0 \ : \ (Lo \to Lo) \to m\,() \\
 & g_0 \ : \ m\ Lo
\end{array}
$$

The full definitions of the "unlifted" Lo operators, u_0 and g_0, are not given. A notational convention used throughout attaches a subscript H or L to any operator acting exclusively on the Hi or Lo domain, respectively.

3.1. Separability via Layered State Monads

Execution of threads in a separation kernel is ultimately reflected as a sequence of updates on the Hi and Lo domains; this section describes separation may be defined monadically and how layering supports separation verification. The Lo domain must be *separable* (41; 40) 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.

Intuitively, separating Lo from Hi means that Lo-events are unaffected the execution of Hi-events. 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, $mask$, which commutes with the Lo operations and cancels the Hi ones (i.e., $h_i ; \mathit{mask} = \mathit{mask}$), then we may extract the Lo effects by erasing the Hi ones:

$$
\begin{array}{rl}
 & h_0\ ;\ l_0\ ; \ldots ;\ h_n\ ;\ (l_n\ ;\ \mathit{mask}) \\
= & h_0\ ;\ l_0\ ; \ldots ;\ (h_n\ ;\ \mathit{mask})\ ;\ l_n \\
= & h_0\ ;\ l_0\ ; \ldots\ ;\ \mathit{mask}\ ;\ l_n \\
= & \ldots = l_0\ ; \ldots ;\ l_n\ ;\ \mathit{mask}
\end{array}
$$

The key insight is that *all* layered state monads have $mask$! 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.

Layered State Monads & Separation. 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 3.

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* $f, f' : s \to s$,

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

The (sequencing) axiom shows how updating by f and then updating by f' is the same as just 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) we use later is:

$$
u\,f >\!> \mathit{mask} \quad = \quad \mathit{mask} \qquad \text{(clobber)}
$$

where $mask$ is defined as: $u\ (\lambda\_.\sigma_0)$ for some state σ_0.

Definition 3 *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* $\varphi >\!> \gamma = \gamma >\!> \varphi$ *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 $(\mathit{StateT}\ s\ M)$, 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 $M' = \mathit{StateT}\, s'\, M$ with operations η', $\star'$, lift, g', and u' defined by $(\mathit{StateT}\, s')$. 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.*

Theorem 2 shows that, in layer state monads, the update operations are noninterfering; thus, the operations u_H and u_L in K do not interfere.

Theorem 2 *Let M be the state monad $\langle M, \eta, \star, u, g, s \rangle$. Let $M' = \langle \mathit{StateT}\, s'\, M, \eta', \star', u', g', s' \rangle$ be the state monad structure defined by $(\mathit{StateT}\, s')$ 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.*

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

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

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.

4. Addressing Integrity Concerns

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 (39). 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 (25; 31) support a similar notion of sequential concurrent computation:

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

Here, the bind operator for R is defined recursively in terms of the bind and unit for K. 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 atomic 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 $R\,()$:

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

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.

With this discussion in mind, the previous construction may be generalized as the monad transformer (30), $\mathbf{data}\ \mathit{ResT}\ m\ a\ =\ \mathit{Done}\ a \mid \mathit{Pause}\ (m\ (\mathit{ResT}\ m\ a))$, whose bind, unit, and step operations are defined as:

2 This is a slight simplification which suffices for our presentation.

$$
\begin{array}{lll}
(\mathit{Done}\ v) \star f & = f\ v \\
(\mathit{Pause}\ r) \star f & = \mathit{Pause}\ (r \star_m \lambda\kappa.\ \eta_m\ (\kappa \star f)) \\
\eta & = \mathit{Done} \\
\mathit{step} & : & m\ a \rightarrow \mathit{ResT}\ m\ a \\
\mathit{step}\ x & = \mathit{Pause}\ (x \star_m (\eta_m \circ \mathit{Done}))
\end{array}
$$

We refine this transformer to make it "security conscious" by reflecting the Hi and Lo security levels:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{ResT}\ m\ a\ =\ \mathit{Done}\ a \mid \mathit{Pause}_L\ (m\ (\mathit{ResT}\ m\ a)) \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{Pause}_H\ (m\ (\mathit{ResT}\ m\ a)) \\
\mathit{step}_d\ :\ m\ a\ \rightarrow\ \mathit{ResT}\ m\ a \\
\mathit{step}_d\ x\ =\ \mathit{Pause}_d\ (x \star_m (\eta_m\ \circ\ \mathit{Done}))\ \ \text{for}\ d \in \{H, L\}
\end{array}
$$

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 1, right); 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 $Event$, $Process$, and Exp languages, as well as specifying a scheduler. It is assumed the monad K is defined as in Section 3, that the Hi and Lo types model computer memory, and that R is defined from K using the resumption monad transformer:

$$
\begin{array}{ll}
\mathbf{type}\ \mathit{Loc} = \mathit{String} & \mathbf{type}\ \mathit{Lo} = \mathit{Loc} \rightarrow \mathit{Int} \\
\mathbf{type}\ R\ \ = \mathit{ResT}\ K & \mathbf{type}\ \mathit{Hi} = \mathit{Loc} \rightarrow \mathit{Int}
\end{array}
$$

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 $(\mathit{store}_d\ l\ v)$ stores v at location l on domain d:

$$
\begin{array}{lll}
\mathit{getloc}_d & : & \mathit{Loc}\ \rightarrow\ K\ \mathit{Int} \\
\mathit{getloc}_d\ l & = & (g_d \star_d \lambda\,\sigma.\ \eta\ (\sigma\ l)) \\
\mathit{store}_d & : & \mathit{Loc}\ \rightarrow\ \mathit{Int}\ \rightarrow\ K\ a \\
\mathit{store}_d\ l\ v & = & u_d\ [l \mapsto v] \\
[i \mapsto v] & = & \lambda\,\sigma.\ \lambda\,n.\ \mathit{if}\ i{=}n\ \mathit{then}\ v\ \mathit{else}\ \sigma\ n
\end{array}
$$

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

$$
\begin{array}{lll}
\mathcal{E}_d[\![-]\!] & : & \mathit{Event}\ \rightarrow\ R\ () \\
\mathcal{E}_d[\![l := e]\!] & = & \mathit{step}_d\ (\mathcal{V}_d[\![e]\!] \star \lambda\ v.\ \mathit{store}_d\ l\ v) \\
\mathcal{P}_d[\![-]\!] & : & \mathit{Process}\ \rightarrow\ R\ () \\
\mathcal{P}_d[\![e\ ;\mathit{es}]\!] & = & \mathcal{E}_d[\![e]\!] >> \mathcal{P}_d[\![\mathit{es}]\!]
\end{array}
$$

The process semantics, $\mathcal{P}_d[\![-]\!]$, gives a precise notion of thread: a *thread* is any R computation, φ, for which there is a $p \in \mathit{Process}$ such that $\mathcal{P}_d[\![p]\!] = \varphi$. 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 3):

$$
\begin{array}{lll}
\mathcal{V}_d[\![-]\!] & : & \mathit{Exp}\ \rightarrow\ K\ \mathit{Int} \\
\mathcal{V}_d[\![i]\!] & = & \eta\ i \\
\mathcal{V}_d[\![l]\!] & = & \mathit{getloc}_d\ l \\
\mathcal{V}_d[\![e_1 \oplus e_2]\!] & = & \mathcal{V}_d[\![e_1]\!] \star \lambda v_1.\ \mathcal{V}_d[\![e_2]\!] \star \lambda v_2.\ \eta\ (v_1 \oplus v_2)
\end{array}
$$

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

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 waiting thread list ts:

$$
\begin{array}{l}
\mathit{rr}\ :\ [R\ ()]\ \rightarrow\ R\ () \\
\mathit{rr}\ []\ \qquad\qquad\qquad = \mathit{Done}\ () \\
\mathit{rr}\ (\mathit{Pause}_d\ t\ ::\ ts) = \mathit{Pause}_d\ (t \star \lambda\kappa.\ \eta\ (\mathit{rr}\ (ts +\!\!+ [\kappa])))
\end{array}
$$

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.

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 synchronous receive events—which introduce the possibility of insecure information flow. Interdomain interactions are mediated entirely through the separation kernel as in Rushby's original conception (41; 40) and it is in the kernel that the "no write down" security policy is enforced. This extension follows the pattern of modular language definitions (18; 12) 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 1, right) 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. We now consider a refinement to the concurrency model presented in the Section 4 which allows computations to signal requests and receive responses to and from the kernel; we coin the term *reactive* resumption to distinguish this structure from the previous one. A *reactive* program (20) is one which interacts continually with its environment and may be designed to not terminate (e.g., an operating system). Reactive programs may be modeled with reactive resumption computations. The notion of concurrent computation associated with the reactive resumption monad resembles nothing so much as the interaction between an operating system and processes making system calls.

As with basic resumptions, reactive resumption monads may also be generalized as a monad transformer (25). The following monad transformer abstracts over the request and response data types (q and r, respectively) as well as over the input monad m:

$$
\begin{array}{lcl}
\mathbf{data}\, \mathit{ReactT}\ q\ r\ m\ a & = & D\ a \\
 & | & P\ (q,\ r \rightarrow (m\ (\mathit{ReactT}\ q\ r\ m\ a))) \\
\eta\ v & = & D\ v \\
(D\ v) \star f & = & f\ v \\
P\ (req, r) \star f & = & P\ (req,\ \lambda rsp.\ (r\ rsp) \star_m \lambda\kappa.\ \eta_m\ (\kappa \star f))
\end{array}
$$

In the last clause, the response rsp to request req is passed to the rest of the computation r.

Reactive resumption monads have two non-proper morphisms. The first of these, $step$, is defined as it was with $ResT$. 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 $signull$:

$$
\begin{array}{lcl}
step & : & m\ a \rightarrow Re\ a \\
step\ x & = & P\ (Cont,\ \lambda\, Ack.\ x \star_m (\eta_m \circ D)) \\
signal & : & Req \rightarrow Re\ Rsp \\
signal\ q & = & P\ (q,\ \eta_m \circ \eta) \\
signull & : & Req \rightarrow Re\ () \\
signull\ q & = & P\ (q,\ \lambda\_.\ \eta_m\ (\eta_{Re}\ ()))
\end{array}
$$

We make the monad transformer $(ReactT\ q\ r)$ security-conscious as before by including a high and low security pause:

$$
\begin{array}{lcl}
\mathbf{data}\, \mathit{ReactT}\ q\ r\ m\ a & = & D\ a \\
 & | & P_L\ (q,\ r \rightarrow (m\ (\mathit{ReactT}\ q\ r\ m\ a))) \\
 & | & P_H\ (q,\ r \rightarrow (m\ (\mathit{ReactT}\ q\ r\ m\ a)))
\end{array}
$$

The bind and unit operations are defined analogously to $ResT$ as are the high and low security versions of the $step$, $signal$, and $signull$ (11). 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 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, `bcast(`l`)` and `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 received response ($Rcvd\ Int$):

$$
\begin{array}{l}
\mathbf{type}\ Re\ \ = ReactT\ Req\ Rsp\ K \\
\mathbf{data}\ Req = Cont \mid Bcst\ Int \mid Rcv \\
\mathbf{data}\ Rsp = Ack \mid Rcvd\ Int
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathcal{E}_d[\![l := e]\!] & = & step_d\ (\mathcal{V}_d[\![e]\!] \star \lambda\, v.\ store_d\ l\ v) \\
\mathcal{E}_d[\![\mathtt{bcast}(x)]\!] & = & step_d\ (getloc_d\ x) \star (signull_d \circ Bcst) \\
\mathcal{E}_d[\![\mathtt{recv}(x)]\!] & = & (signal_d\ Rcv) \star \lambda(Rcvd\ m). \\
 & & step_d(store_d\ x\ m)
\end{array}
$$

The `bcast(`x`)` event reads the contents of x and requests its broadcast through a $Bcst$ signal. The `recv(`x`)` event signals a Rcv request, and, once message m is received, writes it to location x.

The kernel, $rr : ([Re()], [Int], [Int]) \rightarrow R()$, is defined in Figure 2. 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 2 introduces some shorthand useful in defining the scheduler rr. $(r \bullet_d s)$ passes the response signal s to the "continuation" r; that is, r is the second component in an Re computation $P_d(q, r)$. If a request may be handled by rr without affecting K, then $next_d$ is used.

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* (5)—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*

$$
\begin{array}{l}
rr : ([Re()], [Int], [Int]) \rightarrow R() \\
rr\ ([], \_, \_) \qquad = Done\ () \\
rr\ ((t{::}ts), l, h) = \\
\quad \textbf{case}\ (t, l, h)\ \textbf{of} \\
\qquad (P_d(Cont, r), \_, \_) \qquad \rightarrow\ Pause_d\ ((r\ \ Ack) \star \lambda k.\ \eta\ (rr\ (ts{+\!\!+}[k], l, h))) \\
\qquad (P_H(Bcst\ \ m, r), \_, \_) \quad \rightarrow\ next_H\ (ts{+\!\!+}[r \bullet_H Ack], l, h{+\!\!+}[m]) \\
\qquad (P_L(Bcst\ \ m, r), \_, \_) \quad \rightarrow\ next_L\ (ts{+\!\!+}[r \bullet_L Ack], l{+\!\!+}[m], h{+\!\!+}[m]) \\
\qquad (P_H(Rcv, r), \_, []) \qquad \rightarrow\ next_H\ (ts{+\!\!+}[P_H(Rcv,\ r)], l, []) \\
\qquad (P_H(Rcv, r), \_, (m{::}hs)) \rightarrow\ next_H\ (ts{+\!\!+}[r \bullet_H (Rcvd\ m)], l, hs) \\
\qquad (P_L(Rcv, r), [], \_) \qquad \rightarrow\ next_L\ (ts{+\!\!+}[P_L(Rcv,\ r)], [], h) \\
\qquad (P_L(Rcv, r), (m{::}ls), \_) \ \rightarrow\ next_L\ (ts{+\!\!+}[r \bullet_L (Rcvd\ \ m)], ls, h)
\end{array}
$$

$$
\begin{array}{l}
(\bullet_d)\ \ :\ \ (Rsp{\rightarrow}K(Re\ a)){\rightarrow}Rsp{\rightarrow}Re\ a \\
r \bullet_d s = P_d\ (Cont,\ \lambda Ack.\ r\ s) \\
next_d\ :\ ([Re\ ()], [Int], [Int]){\rightarrow}R\ () \\
next_d\ = Pause_d \circ \eta_K \circ rr
\end{array}
$$

Figure 2. ***Kernel for interdomain communication.***

equivalence is similar in that it quantifies over initial segments of R computations—two R computations are take equivalent if the "$mask$ing 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 $take_L$; they are used to capture and project the aforementioned initial sequences. The run morphism projects basic resumption computations to K. ($take_L\ n\ t$) partitions a thread t into two parts; the first part is the smallest initial segment or "prefix" of t containing n operations on Lo; the rest of t is returned as its value:

$$
\begin{array}{l}
run\ :\ R\ a\ \rightarrow\ K\ a \\
run\ (Done\ v) \qquad = \eta\ v \\
run\ (Pause_d\ \varphi) \quad = \varphi \star run \\
take_L\ :\ Int\ \rightarrow\ R\ a\ \rightarrow\ R\ (R\ a) \\
take_L\ 0\ x \qquad\quad = Done\ x \\
take_L\ n\ (Pause_L\ \varphi) = Pause_L\ (\varphi \star (\eta \circ (take_L\ (n{-}1)))) \\
take_L\ n\ (Pause_H\ \varphi) = Pause_H\ (\varphi \star (\eta \circ (take_L\ n)))
\end{array}
$$

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. 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:

$$
\begin{aligned}
run(x \star_R f) &= (run\ x) \star_K (run \circ f) && (1) \\
take_L\ (n{+}1)\ \varphi &= (take_L\ 1\ \varphi) \star_R (take_L\ n) && (2)
\end{aligned}
$$

where φ is an infinite resumption. 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 as in that case both sides of (1) denote $\bot$. Note also that ($take_L\ n\ t$) is always finite. Property (2) follows by induction on n.

Definition 4 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 4 (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*

$$run\ (take_L\ n\ \varphi) >> mask\ =\ run\ (take_L\ n\ \gamma) >> mask$$

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

Definition 5 (Take Separation) *Domain separation holds for the kernels* (*i.e., one of points 1-3*) *if, and only if,* $rr\ [(ts, l, h)] \equiv_{te} rr\ [(ts{\downarrow_{\texttt{Lo}}}, l, h)]$ *for every finite thread list* ts *and message buffers* h *and* l. ($ts{\downarrow_{\texttt{Lo}}}$) *restricts* ts *to* Lo *threads without changing their order.*

Definition 5 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.

Rather than prove the take separation of the kernels in this paper, we instead prove the salient issue with respect to information security; namely, that Hi broadcasts have no effect on Lo receives:

Theorem 4 (no write down) *For* $x, y : Loc$ *and* $l, h : [Int]$,

$$
\begin{aligned}
&run\ (rr\ ([\mathcal{E}_H[\![\texttt{bcast(}x\texttt{)}]\!]\ >>\ \mathcal{E}_L[\![\texttt{recv(}y\texttt{)}]\!]], l, h))\ >>\ mask \\
&\qquad = run(rr\ ([\mathcal{E}_L[\![\texttt{recv(}y\texttt{)}]\!]], l, h)) >> mask
\end{aligned}
$$

Proof. Below are three properties used in this proof:

$$
\begin{array}{lll}
rr([\mathcal{E}_L[\![\texttt{recv(}y\texttt{)}]\!]], l, h) & = rr([\mathcal{E}_L[\![\texttt{recv(}y\texttt{)}]\!]], l, h') & (i)\\
run\ (Pause_d(x \star \lambda v.\ \eta\ y)) & = x \star \lambda v.\ run\ y & (ii)\\
run\ (Pause_d(\eta\ x)) & = run\ x & (iii)
\end{array}
$$

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.

$$
\begin{array}{l}
run\ (rr\ ([\mathcal{E}_H[\![\texttt{bcast(}x\texttt{)}]\!] \texttt{>>} \mathcal{E}_L[\![\texttt{recv(}y\texttt{)}]\!]], l, h)) \texttt{>>} mask\\
\{\text{def. } \mathcal{E}_H[\![-]\!], r_y{=}\mathcal{E}_L[\![\texttt{recv(}y\texttt{)}]\!]\}\\
\quad =\ run\ (rr\ ([step_H(getloc_H(x)) \star \lambda v.\ signal_H(Bcst(v)) \texttt{>>} r_y\ ], l, h))\\
\qquad\qquad \texttt{>>} mask\\
\{\text{def. } rr\}\\
\quad =\ run\ (Pause_H(getloc_H(x) \star_K \lambda v.\\
\qquad\qquad \eta\ (rr\ ([signal_H(Bcst(v)) \texttt{>>} r_y], l, h)))) \texttt{>>} mask\\
\{\text{def. } rr\}\\
\quad =\ run\ (Pause_H(getloc_H(x) \star_K \lambda v.\\
\qquad\qquad \eta\ (Pause_H(\eta\ (rr\ ([r_y], l, h{+\!\!+}[v])))))) \texttt{>>} mask\\
\{i\}\\
\quad =\ run\ (Pause_H(getloc_H(x) \texttt{>>}\\
\qquad\qquad \eta\ (Pause_H(\eta\ (rr\ ([r_y], l, h)))))) \texttt{>>} mask\\
\{ii\}\\
\quad =\ getloc_H(x) \texttt{>>} run\ (Pause_H(\eta\ (rr\ ([r_y], l, h))) \texttt{>>} mask\\
\{iii\}\\
\quad =\ getloc_H(x) \texttt{>>} run\ (rr\ ([r_y], l, h))) \texttt{>>} mask\\
\{\text{atomic n.i., def. } r_y\}\\
\quad =\ getloc_H(x) \texttt{>>} mask \texttt{>>} run\ (rr\ ([\mathcal{E}_L[\![\texttt{recv(}y\texttt{)}]\!]], l, h)))\\
\{\text{cancell.,atomic n.i.}\}\\
\quad =\ mask \texttt{>>} run\ (rr\ ([\mathcal{E}_L[\![\texttt{recv(}y\texttt{)}]\!]], l, h)))\\
\quad =\ run\ (rr\ ([\mathcal{E}_L[\![\texttt{recv(}y\texttt{)}]\!]], l, h))) \texttt{>>} mask
\end{array}
$$

□

6. Achieving Scalability

How are typical operating system behaviors (e.g., process forking, 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 forking primitive called `fork`—to the interdomain communication kernel of the previous section. As it turns out, this functionality requires no change to the existing resumption monadic framework and has little impact on the security verification.

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 forking primitive. The changes are quite minimal. First, add an additional request Frk to Req; the Rsp type remains unchanged as the response to a fork will be Ack.

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

Implicitly, this change to Req is actually a refinement to the reactive resumption monad transformer, but we assume now that $Re\ a\ =\ ReactT\ Req'\ Rsp\ K\ a$. Then, define the `fork` event and add a clause to the kernel scheduler rr:

$$
\begin{array}{l}
\mathcal{E}_d[\![\texttt{fork}]\!] \qquad\quad = signull_d\ Frk\\
rr\ ((t{::}ts), l, h) = \mathbf{case}\ \ (t, l, h)\ \ \mathbf{of}\\
\qquad\qquad\qquad \vdots\\
\qquad (P_d(Frk, r), \_, \_) \rightarrow next_d(ts{+\!\!+}[r', r'], l, h)\\
\qquad\qquad \mathbf{where}\ r' = r \bullet_d Ack
\end{array}
$$

The meaning of `fork` is simply to signal the kernel with a Frk request, and the kernel action taken simply duplicates the signaling thread within the thread list. The only change to the kernel code is an additional branch within the **case** expression of rr (11). It is easily shown that $run\ (rr\ ([\mathcal{E}_d[\![\ \texttt{fork}]\!], l, h))\ =\ \eta_K\ ()$, or, in other words, process forking has no "footprint" on K or interdomain message buffers and thus change has little impact on security verification.

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 (3), 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. Related Work

Many techniques in language-based security (45; 14; 44; 34; 29) are *proscriptive*, meaning that they rely on sophisticated type systems to reject programs with security flaws. Other models (9; 22; 48; 23; 41) 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 applies relational semantics to the control of information flow (15). That approach also involves partitioning the state variables of a concurrent, guarded command language (similar to *Beh*, Point 1) 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* in that it nullifies the effects of s on the high state. A drawback of their approach (according to the authors (15)) 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 (26). 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 (45)) is characterized via a translation into DCC. An encoding of DCC into system F is presented in Tse, et al. (46). In (1), the Smith-Volpano fragment has a conventional store passing semantics of state, except that the denotational model of DCC (like those in (24; 14)) uses parametricity (38) to restrict the store transformers to those respecting the security discipline. The state monad transformer provides a canonical means of constructing such store transformer functions (namely, g and u) as evidenced by Theorems 1-3; this is central our approach.

There has been a growing emphasis on such *language-based* techniques for information flow security (45; 14; 44; 33; 34); please see Sabelfeld and Myers (42) 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 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 which are secure but which will be rejected by the type system due to lack of precision, and second, there will be programs that have information flow leaks which we want to allow (e.g., a declassification program (49)) which would be rejected by the type system. An interesting open problem is how well the current approach accommodates declassification and other relaxed formulations of non-interference (17).

Separation logic (28; 36) incorporates the notion of disjoint regions of state into the specification logic of Reynolds (37); 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 (28) 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 (25). The formulation of basic resumptions in terms of monad transformers used here is that of Papaspyrou (30); the reactive resumption monad transformer originates with Moggi (25). Concurrency may also modeled by the continuation-passing monad (7); 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 or reasoning, as evidenced by the security verification outlined in the previous section.

There have been many previous attempts to develop secure OS kernels: PSOS (27), KSOS (21), UCLA Secure Unix (47), KIT (4), and EROS (43) among many others. There has also been work using functional languages to develop high confidence system software: the Fox project at CMU (10) 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 (6) 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 OGI (35) is working to develop and formally verify *OSKer* (Oregon Separation Kernel), a kernel for MLS applications. To formally verify security properties of such a system is a formidable task, and the current work arose as an exemplary design for *OSKer*.

8. 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 mech-

(a) Broadcast & Receiver Threads

```
brc = x:=100;
      loop { x:=x+1 ; bcast(x); }

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 3. *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.

anisms based in type checking. However, such approaches are domain-specific 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 (41)) 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 proven their usefulness in the construction of modular interpreters and compilers (18; 12), and the kernel refinements in Figure 1 (right) 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 3).

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 proven their usefulness for modularizing interpreters and compilers (18; 12), resulting in modular components from which systems can be created; the three separation kernels (i.e., Points 1-3 in Figure 1, right) are modular in precisely the same sense.

Acknowledgements

The authors wish to thank René Rydhof Hansen, Tom Harke, 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] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corp., Bedford MA, May 1973.

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

[5] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall Series in Computer Science. Prentice-Hall Europe, London, UK, second edition, 1998.

[6] 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.

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

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

[9] 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, Apr. 1990. IEEE Computer Society Press.

[10] 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.

[11] W. Harrison. Haskell implementations of the monadic separation kernels for CSFW 2005. Available from `www.cs.missouri.edu/~harrison/CSFW05`.

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

[13] 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.

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

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

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

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

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

[19] 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, New York, NY, USA, 1995. ACM Press.

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

[21] 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.

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

[23] 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.

[24] 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.

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

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

[27] 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.

[28] P. W. O'Hearn, H. Yang, and J. C. 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.

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

[30] N. Papaspyrou. A resumption monad transformer and its applications in the semantics of concurrency. In *Proceedings of the 3rd Panhellenic Logic Symposium*, Anogia, Crete, 2001.

[31] N. S. Papaspyrou. A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. In *Proceedings of the 3rd Panhellenic Logic Symposium*, Anogia, Crete, July 17–22, 2001.

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

[33] 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.

[34] 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, Jan. 2002.

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

[36] 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.

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

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

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

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

[41] 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.

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

[43] 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, 1999.

[44] 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, 2001.

[45] 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, Jan. 1998.

[46] 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.

[47] 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.

[48] 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.

[49] 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 $M' = \mathit{StateT}\, s'\, M$ with operations η', $\star'$, lift, g', and u' defined by $(\mathit{StateT}\, s')$. 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.

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

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

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

□

Theorem 2 *Let M be the state monad $\langle M, \eta, \star, u, g, s \rangle$. Let $M' = \langle \mathit{StateT}\, s'\, M, \eta', \star', u', g', s' \rangle$ be the state monad structure defined by $(\mathit{StateT}\, s')$ 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.

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

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

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

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

□