Domain Separation by Construction

William Harrison Mark Tullsen James Hook

Pacific Software Research Center
OGI School of Science & Engineering
Oregon Health & Science University

Abstract. Language-based approaches to security typically use static type systems to control information flow, relying on type inference to distinguish secure programs from insecure ones. This paper advocates a novel approach to language-based security: by structuring software with monads (a form of abstract data type for effects), we are able to maintain separation of effects by construction. The thesis of this work is that well-understood properties of monads and monad transformers aid in the construction and verification of secure software. We introduce a formulation of non-interference based on monads (rather than the typical trace-based formulation). Using this formulation, we prove a non-interference style property for a simple instance of our abstract system model. Because monads may be easily and safely represented within any higher-order, typed functional language, monadic event systems may be directly realized within such a language.

1 Introduction

This paper advocates a novel approach to language-based security: security by construction. Starting from a mathematical model of shared-state concurrency, we develop, through a sequence of refinements, the derivation of an exemplary operating system kernel supporting both standard Unix-like system calls (e.g., fork, sleep, etc.) and a formally verified security policy (domain separation).

The research reported here presents a formal, language-based model of security combining three approaches to system security and language semantics:

- **Security "By Design."** Some approaches advocate implementation strategies for secure system construction, with the idea that such disciplined strategies are more likely to remain secure. One such strategy used in Java implementations, *sandboxing*, limits the scope of stateful effects by executing threads in disjoint regions of memory as illustrated in Figure 1. Good engineering, however, does not constitute a guarantee of any security policy.
- **Trace-based Formal Security Models.** There are a number of formal security models [10, 40, 20, 41, 21] which characterize permissible interactions between concurrent threads in terms of traces of abstract events. These models make precise the intuition that low-security operations should be "oblivious" to the execution of high-level operations. One drawback of such models is that their precise relationship to actual systems remains unclear.

– **Monadic Language Semantics.** The approach advocated here combines "by design" security with trace-based models into a common framework based on *monads*. Monads provide a mathematically sophisticated theory of effects which has proven useful in denotational semantics [23, 17, 25], functional programming [35], and software verification [15]. Structuring our system specifications with monads yields many benefits, not the least of which are a number of useful properties obtained "by construction" which simplify the verification of our security property.

Language-based Security via Monadic Constructions. We present a formal model of security in which the model itself may be refined to an implementation of a system with secure shared-state concurrency. Essential to our approach is the use of monads and monad transformers to structure our specifications. It is our thesis that systems constructed monadically are more easily verified because of the monadic encapsulation of effects. Monads and monad transformers allow us to reason about our system definitions at the level of denotational semantics. Because monads may be easily realized within any higher-order functional programming language, system specifications are readily executable.

Many formal security models are formulated in terms of sequences of abstract events. For the sake of convenience, we will refer to such models as *event systems*. The intended interpretation of events is that they are imperative operations on a shared state, but this is not made explicit—that is, the events themselves are uninterpreted. Our approach makes this interpretation explicit by considering languages of system behaviors and their denotational semantics. According to our view of language-based security, an arbitrary, interleaved sequence of Lo and Hi operations in a traditional traced-based model:

$$h_0, l_0, \ldots, h_n, l_n$$

is viewed as the imperative *program* describing a particular (partial) system behavior:

$$h_0 \; ; \; l_0 \; ; \ldots ; \; h_n \; ; \; l_n$$

We then give such programs a (monadic) denotational semantics:

$$[\![-]\!] : (\mathit{Event}_{\mathsf{Lo}} + \mathit{Event}_{\mathsf{Hi}}) \to \mathsf{R}()$$

where R is a monad encapsulating imperative effects and a notion of interleaving concurrency called resumptions[26, 23, 25]. The monad R is constructed especially to isolate Hi, Lo, and kernel effects from one another. These denotational semantics are developed in Section 5.

Partitioning Effects with Monads. Domain separation is supported by partitioning the state into disjoint pieces, with each piece corresponding to a separate domain. Stateful operations are then given a security level and can only manipulate the storage partition corresponding to its security level. This partitioning

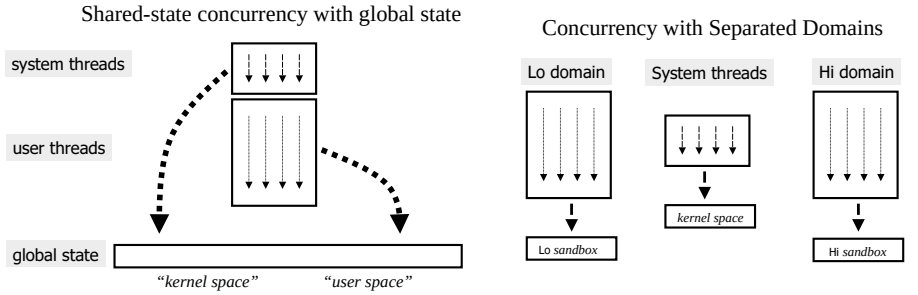

Fig. 1. *Shared-state concurrency with global state (left).* In this system architecture, all threads access the same global state, and so may interfere with one another in ways difficult to control. *Concurrency with separated domains (right).* Secure computation is promoted by partitioning Hi, Lo, and kernel threads into separate domains, each of which has its own storage (or *sandbox*) which only it may affect. Domain separation limits the scope of effects occuring in one domain from others. While this does not guarantee secure computation, it does simplify the implementation, specification, and verification of security properties.

is sometimes referred to as *sandboxing*. With monads, it is a simple matter to partition storage into sandboxes, and this process is particularly straightforward when the monads are constructed with monad transformers.

How this partitioning works is illustrated in Figure 2, where, for the sake of simplicity[1], we assume that there are two security levels Hi and Lo. Corresponding to these levels are separated domains **(c)** that maintain distinct stores H and L, respectively. Hi and Lo stateful operations are then encapsulated within monads of the same name, created with the monad transformers (StateT H) and (StateT L), respectively. Hi and Lo stateful operations h and l may be executed by lifting them to the kernel level **(b)** with liftH and liftL, respectively, and these lift mappings are created by the application of the state monad transformers. Separation of effects is maintained by these lift mappings—precisely how is described in detail in Section 4 below.

Separation By Construction. The approach advocated here achieves secure shared-state concurrency by construction, where "by construction" is used in two different, yet complementary, senses. The process is illustrated in Figure 3, which illuminates the meanings of "by construction":

– **Stepwise-refinement of System.** The vertical axis of Figure 3 measures the richness of observable system events, and each step along that axis marks an addition to those observables. At point 0, only a single, monolithic process domain exists and all thread scheduling is static. At point 1, threads are

[1] All of our results generalize easily to n separate domains.

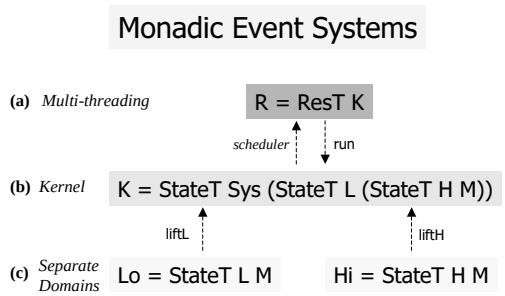

Fig. 2. *Monadic Event Systems*: Processes created in isolated security domains Lo and Hi are lifted to the runtime platform K. Laws about monad transformers (StateT and ResT above) aid in proving domain separation.

executable on separate domains. Point 2 allows statically-scheduled multitasking, while point 3 allows the scheduling of threads to occur dynamically. Dynamic scheduling is necessary for threads to affect their own execution behavior (as with the Unix system call `sleep` and intradomain synchronisation mechanisms such as semaphores) or to affect the system "wait list" (as with the Unix system call `fork`). Point 4 adds such thread-level control operations, and point 5 allows for secure, interdomain communication (such as message-passing which obeying a "no-write-down" security policy).

– **Properties of Monads & Monad Transformers.** Many of the above enhancements to system functionality are reflected in refinements to the underlying monads and monad transformers of Figure 2. Monads and monad transformers allow the effects of threads of differing security levels to be controlled in a mathematically rigorous manner, and this scoping of effects tames insecure interference between threads. The "by construction" properties of monads and monad transformers are extremely useful in formally demonstrating domain separation.

Security Property. The security property we prove may be intuitively described as the execution of low security events being *oblivious* to the execution of high security events. For any initial sequence of interleaved Hi and Lo events

$$h_0 \; ; \; l_0 \; ; \ldots ; \; h_n \; ; \; l_n$$

the effect of its execution on the Lo state should be identical to that of executing the Lo events in isolation:

$$l_0 \; ; \ldots ; \; l_n$$

Using the aforementioned denotational semantics, we make this notion precise in Section 5.3.

Overview. This paper is structured as follows. Section 2 surveys related work and Section 3 summarizes the background on monads, monad transformers, and resumption-based concurrency necessary for understanding this work. Section 4 describes the useful properties that are obtained *by construction* with monad transformers. Section 5 presents the main results of this research—the stepwise development of an exemplary operating system kernel for secure computation. First, a language of system behaviors is defined and given a resumption-monadic semantics in Section 5.1. Second, a system for shared-state concurrency with global store is transformed into a system with separated domains in Section 5.2. Third, security in this setting—*take separation*—is specified and verified in Section 5.3. Finally, Section 6 summarizes the present work and outlines future directions.

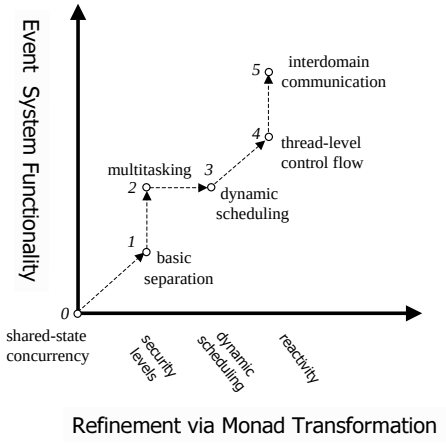

Fig. 3. *Separation by construction*: System functionality is enhanced by refining monad transformers.

2 Related Work

Many formal security models—including non-interference and separability—are formulated in terms of *event systems* [10, 40, 20, 41, 21]. While there are a variety of similar formulations of event systems, they all include the following: an event system S is comprised of a set of abstract *events* E and a set of *traces* T of (potentially infinite) sequences of events in E. Furthermore, each event is assigned a security level, which, for the sake of the present discussion, we will assume to be Lo and Hi[2].

Security policies may then be formulated in terms of events systems as relationships between event traces which specify permitted system executions. These

[2] Event systems also distinguish subsets of E as *input* and *output* events.

typically are of the form: if $\tau_1, \ldots, \tau_n \in T$, then $f(\tau_1, \ldots, \tau_n) \in T$, where function f combines the traces τ_i in some manner specific to the security policy. For example, *separability* [21, 41] can be defined formally as: for every pair of traces $\tau_1, \tau_2 \in T$, then any trace τ such that $(\tau \downarrow_{\mathsf{Lo}}) = (\tau_1 \downarrow_{\mathsf{Lo}})$ and $(\tau \downarrow_{\mathsf{Hi}}) = (\tau_2 \downarrow_{\mathsf{Hi}})$ is also in T. Here $(\tau \downarrow_{\mathsf{Lo}})$ and $(\tau \downarrow_{\mathsf{Hi}})$ filter out all but the Lo and Hi events from τ, respectively. (As formulated these techniques are easily generalized to any total order of security levels known statically at compile time.)

Previous work has focused on languages with implicit imperative features, such as Java or ML. This paper instead assumes a language in which all imperative features are captured by a monad. As a result all impure effects (references, exceptions, I/O) are distinguished from pure computations by their types, and thus side-effects are allowed while preserving the semantics of the purely functional subset of the language.

Moggi was the first to observe that the simple structure known as a monad was appropriate for the development of modular semantic theories for programming languages [22]. In his initial development, Moggi showed that most known semantic effects could be naturally expressed monadically. He also showed how a sequential theory of concurrency could be expressed in the Resumption monad [23]. That model of concurrency is used extensively below. Wadler and colleagues at Glasgow University observed that using monads internally in the pure, higher-order, typed language Haskell gave a natural and safe embedding of effectful computation in a pure language.

We are working to develop and formally verify a security kernel as part of the Programatica project at OGI. To formally verify security properties of such a system is a formidable task. One approach to reducing the enormity of this task has been using type-systems for information-flow. There has been a growing emphasis on such *language-based* techniques for information flow security [34, 33, 16, 32, 27] (see [29] for a 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: 1) there will be programs which are secure but which will be rejected by the type system due to lack of precision and 2) there will be programs that have information flow leaks which we want to allow (e.g., a declassification program) but which would be rejected by the type system.

The Programatica team has investigated a more general—but potentially more expensive—approach to building large systems guaranteed to satisfy security properties. In our approach we derive proofs of security properties by hand, and yet proving security properties of large systems is kept tractable. This is due primarily to the following techniques: 1) Using a purely functional, statically-typed, polymorphic language, we are able to reason about most of the system components using only their types. And, 2) We structure the system so as to minimize the amount of code that must be reasoned about explicitly. We do this using monads, a mechanism previously used for modularizing interpreters. Different monadic types are assigned different security levels, and thereby indi-

vidual applications using effects (such as accessing low security memory) may be validated automatically and statically according to a security policy.

There have been a number of attempts to develop secure OS kernels: PSOS [24], KSOS [19], UCLA Secure Unix [37], SAT [5], KIT [2], EROS [31], and MASK [39] among others. There has also been some work using functional languages to develop high confidence system software: the Fox project at CMU [14] is a case in point of how typed functional languages can be used to build reliable systems software (e.g., network protocols, active networks); the Ensemble project at Cornell [4] uses a functional language to build high performance networking software; and the Switchware project [1] 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.

While our approach does not provide full system assurance automatically, as with the type-based approach, many large components of the system are verified automatically (by the type system); this aids verification of the global system security property. This approach to guaranteeing security underlies the construction of the OSKer[3] operating system kernel.

Formulating security policies in terms of non-interference goes back to the work of Goguen and Meseguer [10, 11]. Our notion of separation is basically the non-interference of Goguen and Meseguer; although for simplicity of the current presentation our approach only allows for an "interferes" relation that is the identity relation (i.e., no domain can interfere with another). Haigh and Young [13] and Rushby [28] have extended this work to the intransitive case (where the "interferes" relation is not required to be transitive). Non-interference has been extended to concurrent [33, 6] and probabilistic models [12].

Modeling concurrency by resumptions was introduced by Plotkin [26, 30]. Moggi showed how resumptions could be modeled with monads [23] and our formulation of resumptions in terms of monad transformers is that of Papaspyrou [25]. Concurrency can also be handled by continuations [38, 7, 8]. Resumptions can be viewed as a disciplined use of continuations. Using resumptions rather than arbitrary continuations makes reasoning about our system easier.

3 Background

Monads [23] can be understood as abstract data types for defining languages and programs [4]. A monad ADT can encapsulate such language features as state, exceptions, multi-threading, environments, and CPS. Although combinations of such features can be encapsulated in a single monad, a more modular approach, in which each feature can be treated separately, is achieved with *monad transformers* [23, 18, 17]. Monad transformers allow us to easily combine and extend monads. (A monad is extended similarly to how a class is extended using inheritance in object oriented languages: what makes sense in a monad "class" makes sense in the "subclass" created by inheritance.) In this section we will introduce

[3] *OSKer* stands for "Oregon Security Kernel."

[4] An introduction to monads can be found in Wadler [36].

monads and monad transformers, we will then describe the resumption monad transformer: a monad for concurrency.

3.1 Monads and Monad Transformers

A monad is a triple $\langle \mathsf{M}, \eta, \star \rangle$ consisting of a type constructor M and two operations:

$$\begin{aligned} \eta &: a \to \mathsf{M}\, a && \text{(unit)} \\ (\star) &: \mathsf{M}\, a \to (a \to \mathsf{M}\, b) \to \mathsf{M}\, b && \text{(bind)} \end{aligned}$$

The η operator is the monadic equivalent of the identity function, it brings an element into the monad. The $\star$ operator gives a form of sequential composition. These operators must satisfy the monad laws:

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

In what follows we often use the $\gg$ operator, defined in terms of $\star$ thus:

$$v \gg k = v \star \lambda\_.k$$

The identity monad $\langle \mathsf{Id}, \eta_{\mathsf{Id}}, \star_{\mathsf{Id}} \rangle$ is defined as follows:

$$\begin{aligned} \mathsf{Id}\ a &= a \\ x \star_{\mathsf{Id}} k &= k\ x \\ \eta_{\mathsf{Id}}\ x &= x \end{aligned}$$

Another example is the state monad $\langle \mathsf{St}\ s, \eta_{\mathsf{St}}, \star_{\mathsf{St}} \rangle$ defined as follows:

$$\begin{aligned} \mathsf{St}\ s\ a &= s \to (a \times s) \\ \eta_{(\mathsf{St}\ s)}\ x &= \lambda\sigma : s.(x, \sigma) \\ x \star_{(\mathsf{St}\ s)}\ f &= \lambda\sigma_0.\ \text{let}\ (a, \sigma_1) = x\ \sigma_0\ \text{in}\ f\ a\ \sigma_1 \\ \mathsf{u}(\Delta) &= \lambda\sigma.((), \Delta\ \sigma) \\ \mathsf{g} &= \lambda\sigma.(\sigma, \sigma) \end{aligned}$$

Here, $()$ signifies both the unit type and the single element of that type. The operators u and g, for updating and getting the state respectively, are defined only for the $\mathsf{St}\, s$ monad. Such monad specific operators are referred to as improper morphisms.

There are various formulations of monad transformers, we follow that given in [18] where a monad transformer consists of a type constructor T, a mapping from a given monad $\langle \mathsf{M}, \eta_{\mathsf{M}}, \star_{\mathsf{M}} \rangle$ to a new monad $\mathsf{T}\,\mathsf{M} = \langle \mathsf{T}\,\mathsf{M}, \eta_{(\mathsf{T}\,\mathsf{M})}, \star_{(\mathsf{T}\,\mathsf{M})} \rangle$, and an associated function $\mathrm{lift}_{\mathsf{T}}$. The function $\mathrm{lift}_{\mathsf{T}} : \mathsf{M}\, a \to \mathsf{T}\,\mathsf{M}\, a$ performs a "lifting" of computations in M to computations in $(\mathsf{T}\,\mathsf{M})$; and will generally satisfy the *Lifting Laws* [17]:

$$\begin{aligned} \mathrm{lift} \circ \eta_{\mathsf{M}} &= \eta_{(\mathsf{T}\,\mathsf{M})} \\ \mathrm{lift}(x \star_{\mathsf{M}} f) &= (\mathrm{lift}\ x) \star_{(\mathsf{T}\,\mathsf{M})} (\mathrm{lift} \circ f) \end{aligned}$$

These laws ensure that a monad transformer adds features without changing features of the base monad M.

As an example, the state monad $\mathsf{St}\, s$ can be written more generally as the monad transformer $\mathsf{StateT}\, s$ which transforms M. The definition is in Figure 4. Note that the monad transformer $\mathsf{StateT}\ s$ applied to the identity monad Id gives the original $\mathsf{St}\ s$ monad.

$$
\begin{array}{ll}
\mathsf{StateT}\, s\, \mathsf{M}\, a = s \to \mathsf{M}(a \times s) & \mathsf{ResT}\, \mathsf{M}\, a = \mu R.\mathit{Done}\, a + \mathit{Pause}\, (\mathsf{M}(\mathsf{R}\, a)) \\
\eta\, x = \lambda\sigma.\, \eta_{\mathsf{M}}(x, \sigma) & \eta\, x = \mathit{Done}\, x \\
x \star f = \lambda\sigma_0.\, (x\, \sigma_0) \star_{\mathsf{M}} (\lambda(a, \sigma_1).f\, a\, \sigma_1) & (\mathit{Done}\, v) \star f = f\, v \\
\mathrm{lift}\, x = \lambda\sigma.\, x \star_{\mathsf{M}} \lambda y.\, \eta_{\mathsf{M}}(y, \sigma) & (\mathit{Pause}\, m) \star f = \mathit{Pause}(m \star_{\mathsf{M}} \lambda r.\eta(r \star_{\mathsf{M}} f)) \\
\mathsf{u}(\Delta : s \to s) = \lambda\sigma.\, \eta_{\mathsf{M}}((), \Delta\, \sigma) & \mathrm{step}\, \phi = \mathit{Pause}(\phi \star_{\mathsf{M}} \lambda v.\eta_{\mathsf{M}}(\mathit{Done}\, v)) \\
\mathsf{g} = \lambda\sigma.\, \eta_{\mathsf{M}}(\sigma, \sigma) &
\end{array}
$$

Fig. 4. The State and Resumption Monad Transformers ($\mathsf{StateT}\, s$) and ResT. The unit and bind of the new monads are η and $\star$, respectively, while those of the transformed monad M are subscripted.

3.2 Resumption-based Concurrency and ResT

This section introduces concurrency based on the resumption monad transformer, the definition of which can be found in Figure 4. How resumption based concurrency works is best explained by an example. We define *thread* to be a (possibly infinite) sequence of "atomic operations." We make this notion precise below, but for the moment, assume that an atomic operation is a single machine instruction and that a thread is a stream of such instructions characterizing a program execution. Consider first that we have two simple threads $a = [a_0; a_1]$ and $b = [b_0]$. According to the "concurrency as interleaving" model of concurrency, concurrent execution of threads a and b means the set of all their possible interleavings: $\{[a_0; a_1; b_0], [a_0; b_0; a_1], [b_0; a_0; a_1]\}$.

But how do computations in a resumption monad correspond to threads? If the atomic operations of a and b are computations of type $\mathsf{M}\,()$, then the computations of type $\mathsf{ResT}\, \mathsf{M}\, ()$ are the set of possible interleavings:

$$
\begin{array}{l}
\mathit{Pause}\, (a_0 \gg_{\mathsf{M}} \eta_{\mathsf{M}}(\mathit{Pause}\, (a_1 \gg_{\mathsf{M}} \eta_{\mathsf{M}}(\mathit{Pause}\, b_0 \gg_{\mathsf{M}} \eta_{\mathsf{M}}(\mathit{Done}\, ())))))) \\
\mathit{Pause}\, (a_0 \gg_{\mathsf{M}} \eta_{\mathsf{M}}(\mathit{Pause}\, (b_0 \gg_{\mathsf{M}} \eta_{\mathsf{M}}(\mathit{Pause}\, a_1 \gg_{\mathsf{M}} \eta_{\mathsf{M}}(\mathit{Done}\, ())))))) \\
\mathit{Pause}\, (b_0 \gg_{\mathsf{M}} \eta_{\mathsf{M}}(\mathit{Pause}\, (a_0 \gg_{\mathsf{M}} \eta_{\mathsf{M}}(\mathit{Pause}\, a_1 \gg_{\mathsf{M}} \eta_{\mathsf{M}}(\mathit{Done}\, ()))))))
\end{array}
$$

Closer comparison of these two reveals that, instead of using the lazy "cons" operation $(h : t)$ as in the stream definition of concurrency above, the monadic version uses something similar: $\mathit{Pause}\, (h \gg_{\mathsf{M}} \eta_{\mathsf{M}} t)$. This is important because threads may be infinite, and the laziness of *Pause* allows infinite *computations* to be constructed in $(\mathsf{ResT}\, \mathsf{M})$ just as the laziness of $(h : t)$ allows infinite *streams* to be constructed.

Finally, we note that the resumption semantics of concurrency involves the elaboration of all possible thread interleavings, and that, while such a semantics may be expressed monadically via the non-determinism monad [23, 18], it is not computationally tractable. We choose instead to pick out a single particular interleaving via a scheduler.

4 "By Construction" Properties of Monad Transformers

We get a number of useful properties by construction through the use of monad transformers. The state monad transformer's lift mappings have two principal uses here. First, lifting preserves stateful behavior. In Figure 2 for example, this means that Hi and Lo operations behave the same when lifted to the kernel monad K as at their respective base monads. This is formally captured in Section 4.1 below. Furthermore, liftings also delimit the effects of stateful operations on separate domains, and this phenomenon—which we call *atomic noninterference*—is described in detail in Section 4.2.

4.1 State Monads and Their Axiomatization

This section presents an algebraic characterization of state monads. Intuitively, a state monad is a monad with non-proper morphisms to manipulate state. The behavior of these non-proper morphisms is captured by axioms below. Not surprisingly, it is then demonstrated that the state monad transformer creates new state monads, and preserves existing state monads.

State Monad Structure. The quintuple $\langle \mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, s\rangle$ is a *state monad structure* when:

1. $\langle \mathsf{M}, \eta, \star\rangle$ is a monad with operations: unit $\eta : \alpha \to \mathsf{M}\,\alpha$ and bind $\star : \mathsf{M}\,\alpha \to (\alpha \to \mathsf{M}\,\beta) \to \mathsf{M}\,\beta$
2. update operation on s is $\mathsf{u} : (s \to s) \to \mathsf{M}()$
3. get operation on s is $\mathsf{g} : \mathsf{M}\,s$

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

State Monad Axioms. The following axiomitization of the state monad is not meant to be complete. Rather, it reflects the properties of state monads required later in the proofs.

Let $\mathsf{M} = \langle \mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, s\rangle$ be a state monad structure. M is a *state monad* if the following equations hold for any $f, g : s \to s$,

$$\mathsf{u}\, f \star \lambda\_.\mathsf{u}\, g = \mathsf{u}\,(g \circ f) \qquad \text{(sequencing)}$$

$$\mathsf{g} \star \lambda\sigma_0.\mathsf{u}\,(\lambda\_.\sigma_0) = \eta() \qquad \text{(cancellation)}$$

The (sequencing) axiom shows how updating by f and then updating by g is the same as just updating by their composition $(g \circ f)$. The (cancellation) axiom requires that overwriting the state with the result of an immediately-preceding g operation has no effect.

For state monad $\langle \mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, \sigma \rangle$, a consequence of (sequencing) we use later is:

$$\mathsf{u}\ f \gg \text{init} = \text{init} \qquad \text{(clobber)}$$

where init is defined as: $\mathsf{u}\ (\lambda\_.\sigma_0)$ for some constant state σ_0.

Theorem 1 (StateT creates a state monad). *For any monad M, let monad $\mathsf{M}' = \mathsf{StateT}\ sto\ \mathsf{M}$, $u : (sto \to sto) \to \mathsf{M}'()$ and $g : \mathsf{M}'\ sto$ be the non-proper morphisms added by $(\mathsf{StateT}\ sto)$. Then $\langle \mathsf{M}', \eta_{\mathsf{M}'}, \star_{\mathsf{M}'}, u, g, sto \rangle$ is a state monad.*

Theorem 2 (StateT preserves stateful behavior). *For any state monad $\mathsf{M} = \langle \mathsf{M}, \eta, \star, u, g, sto \rangle$, the following state monad structure is a state monad:*

$$\langle \mathsf{StateT}\ s\ \mathsf{M}, \eta', \star', \text{lift} \circ u, \text{lift} g, s \rangle$$

where η', $\star'$, and lift *are the monadic unit, bind, and lifting operations, respectively, defined by $(\mathsf{StateT}\ s)$.*

4.2 Formalizing Atomic Non-interference

The second "by construction" property of monad transformers relates to how their associated lift mappings delimit stateful effects in monads created from multiple applications of the state monad transformers (e.g., the kernel monad K in Figure 2). This property—*atomic non-interference*—is useful for proving the security property below in Section 5.3.

Atomic Non-interference Relation #. For any monad M with bind operation $\star$, define the non-interference relation $\# \subseteq \mathsf{M}\,() \times \mathsf{M}\,()$ so that, for $\varphi, \gamma : \mathsf{M}\,()$, $\varphi \# \gamma$ holds if and only if:

$$\varphi \gg \gamma = \gamma \gg \varphi$$

Theorem 3 (Atomic Non-interference of updates). *Let M be the state monad $\langle \mathsf{M}, \eta_{\mathsf{M}}, \star_{\mathsf{M}}, \mathsf{u}_A, \mathsf{g}_A, A \rangle$. By Theorem 1, the following structure is also a state monad: $\mathsf{M}' = \langle \mathsf{StateT}\ B\ \mathsf{M}, \eta_{\mathsf{M}'}, \star_{\mathsf{M}'}, \mathsf{u}_B, \mathsf{g}_B, B \rangle$. Then, for all $\delta_A : A \to A$ and $\delta_B : B \to B$,*

$$(\mathsf{u}_B\ \delta_B) \#_{\mathsf{M}'} \text{lift}(\mathsf{u}_A\ \delta_A)$$

Theorem 4 (Non-interference Preserved by Monad Transformers). *Let M be a monad with two operations, $a : \mathsf{M}\,()$ and $b : \mathsf{M}\,()$ such that $a \#_{\mathsf{M}} b$. Then,*

$$(\text{lift}\, a) \#_{(\mathsf{T}\,\mathsf{M})} (\text{lift}\, b)$$

where T is a monad transformer and $\text{lift} : \mathsf{M}\, a \to (\mathsf{T}\,\mathsf{M})\, a$.

5 Stepwise Development of a Secure OS Kernel

In monadic event systems, events are programs in the *Event* language, and traces of events are the denotations of these programs according to a resumption-monadic semantics. The abstract syntax and resumption semantics of the language of events are presented below. The *Event* language contains sufficient expressiveness to allow for potentially infinite streams of operations because of the inclusion of loops. More importantly, it allows for expression of potentially interfering programs as well.

Abstract Syntax for the Event Language. Below is an abstract syntax for *Event*:

$$\begin{array}{lll} \mathit{Event} & ::= & \mathit{Var}\texttt{:=}\mathit{Exp} \mid \texttt{skip} \mid \mathit{Event}\texttt{;}\mathit{Event} \mid \\ & & \quad \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Event}\ \texttt{else}\ \mathit{Event} \mid \texttt{while}\ \mathit{Exp}\ \texttt{do}\ \mathit{Event} \\ \mathit{Exp} & ::= & \mathit{Var} \mid \mathit{Integer} \end{array}$$

5.1 Shared-State Concurrency with Global State (system 0)

In this section, we introduce the most basic model for shared-state concurrency is constructed. The monadic event system described in this section corresponds to the point labelled with "0" in Figure 3.

Monad Hierarchy Using the monad transformers ResT and StateT defined in Section 3, we define the monad hierarchy for shared-state concurrency with global state. This monadic event system is degenerate in the sense that the For any monad M, this hierarchy is:

$$\mathsf{GI} = \mathsf{StateT}\ G\ \mathsf{M} \qquad \mathsf{K} = \mathsf{GI} \qquad \mathsf{R} = \mathsf{ResT}\ \mathsf{K}$$

Here, state type G is $\mathit{Name} \to \mathit{Integer}$.

Associated with these monad constructions are a number of non-proper morphisms (i.e., monadic operators other than η and $\star$). These are:

$$\begin{array}{ll} \mathsf{u}_G \ : \ (G \to G) \to \mathsf{GI}() & \mathit{stepG} \ : \ \mathsf{K}\,a \to \mathsf{R}\,a \\ \mathsf{g}_G \ : \ \mathsf{GI}\,G & \mathit{stepG}\ \varphi = \mathit{PauseLo}\ (\varphi \star_{\mathsf{K}} \lambda v.\eta_{\mathsf{K}}(\mathit{Done}\ v)) \\ \mathit{liftG} = \mathit{id} & \end{array}$$

The morphism u_G applies a state-to-state map to the current G state in the GI monad, while the morphism g_G reads and returns the current G state. The lifting liftG reinterprets GI computations in the kernel monad K. The aforementioned morphisms are all defined by applications of the StateT monad transformer. The morphisms *stepH* and *stepL* create a "paused" K computation in either the Hi or Lo security levels. The "step" functions result from the application of the ResT transformer.

Semantics for the Event Language. The resumption-monadic denotational semantics for system behaviors is listed below. It is just what one would expect given recent work on the resumption-monadic semantics of concurrency [25].

$$
\begin{array}{ll}
ev\ :\ Event \to \mathsf{R}\ () & \\
ev\ (x\texttt{:=}e) & = (exp\ e) \star_{\mathsf{R}} \lambda v.\ (load \circ \mathsf{u}_G)[x \mapsto v] \\
ev\ (e_1\texttt{;}e_2) & = (ev\ e_1)\ \gg_{\mathsf{R}}\ (ev\ e_2) \\
ev\ \texttt{skip} & = (load \circ \mathsf{u}_G)\ (\lambda i.i) \\
ev\ (\texttt{if}\ \ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2) & = exp\ b \star_{\mathsf{R}} \lambda v.\ if\ v = 0\ then\ (ev\ e_1)\ else\ (ev\ e_2) \\
ev\ (\texttt{while}\ \ b\ \texttt{do}\ c) & = mwhile\ (exp\ b)\ (ev\ c) \\
mwhile\ b\ \varphi & = \ \ b \star_{\mathsf{R}} \lambda v.\ if\ v = 0\ then\ \varphi\ \gg_{\mathsf{R}}\ (mwhile\ b\ \varphi) \\
 & \qquad\qquad\qquad\qquad\qquad else\ \ (load \circ \mathsf{u}_G)\ (\lambda i.i)
\end{array}
$$

$$
\begin{array}{ll}
exp : Exp \to \mathsf{R}\ Integer & load : \mathsf{Gl}\ a \to \mathsf{R}\ a \\
exp\ x = (load\ \mathsf{g}_G) \star_{\mathsf{R}} \lambda\sigma.\eta_{\mathsf{R}}(\sigma\ x) & load = step \circ \mathrm{lift} \\
exp\ i\ = \eta_{\mathsf{R}}\ i &
\end{array}
$$

Please observe that because each observable event (namely u or g) is wrapped by a *load*, each such event is *Pause*'d.

5.2 Basic Separation (system 1)

Event systems contain trace projections based on security level—we write these as "$\downarrow_{\mathsf{Hi}}$" and "$\downarrow_{\mathsf{Lo}}$." Monadic event systems need a similar capability, which is achieved by refining the resumption monad transformer of Papaspyrou [25] to reflect the Hi and Lo security levels. The refined resumption monad transformer is:

$$\mathsf{ResT}\ \mathsf{M}\ a = \mu R.Done\ a + PauseLo\,(\mathsf{M}(\mathsf{R}\,a)) + PauseHi\,(\mathsf{M}(\mathsf{R}\,a))$$

The unit (η) is *Done*, and the bind ($\star$) of the transformed monad is defined just as one would expect:

$$
\begin{array}{ll}
(Done\ v) \star f & = f\ v \\
(PauseLo\ \varphi) \star f & = PauseLo\ (\varphi \star_{\mathsf{M}} \lambda r.\ \eta_{\mathsf{M}}(r \star f)) \\
(PauseHi\ \varphi) \star f & = PauseHi\ (\varphi \star_{\mathsf{M}} \lambda r.\ \eta_{\mathsf{M}}(r \star f))
\end{array}
$$

Using this resumption transformer and StateT, we define the monad hierarchy from Figure 3 for any monad M as:

$$
\begin{array}{ll}
\mathsf{Hi} = \mathsf{StateT}\ H\ \mathsf{M} & \mathsf{K} = \mathsf{StateT}\ H\ (\mathsf{StateT}\ L\ \mathsf{M}) \\
\mathsf{Lo} = \mathsf{StateT}\ L\ \mathsf{M} & \mathsf{R} = \mathsf{ResT}\ \mathsf{K}
\end{array}
$$

Here, L and H are state types equal to $Name \to Integer$.

Associated with these monad constructions are a number of non-proper morphisms (i.e., monadic operators other than η and $\star$). These are:

$$
\begin{array}{ll}
\mathsf{u}_L & : (L \to L) \to \mathsf{Lo}() \\
\mathsf{g}_L & : \mathsf{Lo}\, L \\
\mathit{liftL} & : \mathsf{Lo}\, a \to \mathsf{K}\, a \\
\mathit{stepL} & : \mathsf{K}\, a \to \mathsf{R}\, a \\
\mathit{stepL}\ \varphi = \mathit{PauseLo}\ (\varphi \star_{\mathsf{K}} \lambda v.\eta_{\mathsf{K}}(\mathit{Done}\ v))
\end{array}
\qquad
\begin{array}{ll}
\mathsf{u}_H & : (H \to H) \to \mathsf{Hi}() \\
\mathsf{g}_H & : \mathsf{Hi}\, H \\
\mathit{liftH} & : \mathsf{Hi}\, a \to \mathsf{K}\, a \\
\mathit{stepH} & : \mathsf{K}\, a \to \mathsf{R}\, a \\
\mathit{stepH}\ \varphi = \mathit{PauseHi}\ (\varphi \star_{\mathsf{K}} \lambda v.\eta_{\mathsf{K}}(\mathit{Done}\ v))
\end{array}
$$

The morphisms u_L and u_H apply a state-to-state map to the current state in their respective monads, while the morphisms g_L and g_H read and return the current state. The liftings *liftL* and *liftH* reinterpret Lo and Hi computations, resp., in the kernel monad K. The aforementioned morphisms are all defined by applications of the StateT monad transformer. The morphisms *stepH* and *stepL* create a "paused" K computation in either the Hi or Lo security levels. The "step" functions result from the application of the ResT transformer.

By Theorems 1 and 2, we know that the monad K is a state monad with the operations lifted from the Hi and Lo monads. That is, the following are state monads:

$$\langle \mathsf{K}, \eta_{\mathsf{K}}, \star_{\mathsf{K}}, (\mathit{liftL} \circ \mathsf{u}_L), (\mathit{liftL}\ \mathsf{g}_L), L \rangle \qquad \langle \mathsf{K}, \eta_{\mathsf{K}}, \star_{\mathsf{K}}, (\mathit{liftH} \circ \mathsf{u}_H), (\mathit{liftH}\ \mathsf{g}_H), H \rangle$$

Event Semantics (system 1) There are two semantics for *Event* corresponding to the Hi and Lo security levels—these are *evH* and *evL*, respectively. The low-security resumption semantics *evL* and *expL*. The high-security semantics (not shown), *evH* and *expH*, is analogous[5].

$$
\begin{array}{ll}
\mathit{evL} :: \mathit{Event} \to \mathsf{R}\ () & \\
\mathit{evL}\ (x \texttt{:=} e) & = (\mathit{expL}\ e) \star_{\mathsf{R}} \lambda v.\ (\mathit{loadL} \circ \mathsf{u}_L)[x \mapsto v] \\
\mathit{evL}\ (e_1 \texttt{;} e_2) & = (\mathit{evL}\ e_1)\ \gg_{\mathsf{R}}\ (\mathit{evL}\ e_2) \\
\mathit{evL}\ \texttt{skip} & = (\mathit{loadL} \circ \mathsf{u}_L)\ (\lambda i.i) \\
\mathit{evL}\ (\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2) & = \mathit{expL}\ b \star_{\mathsf{R}} \lambda v.\ \mathit{if}\ v = 0\ \mathit{then}\ (\mathit{evL}\ e_1)\ \mathit{else}\ (\mathit{evL}\ e_2) \\
\mathit{evL}\ (\texttt{while}\ b\ \texttt{do}\ c) & = \mathit{mwhile}\ (\mathit{expL}\ b)\ (\mathit{evL}\ c) \\
\mathit{mwhile}\ b\ \varphi & = \ b \star_{\mathsf{R}} \lambda v.\ \mathit{if}\ v = 0\ \mathit{then}\ \varphi \gg_{\mathsf{R}} (\mathit{mwhile}\ b\ \varphi) \\
 & \qquad\qquad\qquad\qquad \mathit{else}\ (\mathit{loadL} \circ \mathsf{u}_L)\ (\lambda i.i)
\end{array}
$$

$$
\begin{array}{ll}
\mathit{expL} : \mathit{Exp} \to \mathsf{R}\ \mathit{Integer} & \mathit{loadL} : \mathsf{Lo}\ a \to \mathsf{R}\ a \\
\mathit{expL}\ x = (\mathit{loadL}\ \mathsf{g}_L) \star_{\mathsf{R}} \lambda\sigma.\eta_{\mathsf{R}}(\sigma\ x) & \mathit{loadL} = \mathit{stepL} \circ \mathit{liftL} \\
\mathit{expL}\ i\ = \eta_{\mathsf{R}}\ i &
\end{array}
$$

The semantics *evL* (*evH*) creates traces by injecting the Lo (Hi) operations into the *PauseLo* (*PauseHi*) side of R. Let e be $(x\texttt{:=}1)$ in the following:

$$(\mathit{evL}\ e) = \mathit{PauseLo}\ (\mathsf{u}_L[x \mapsto 1] \gg_{\mathsf{K}} \eta_{\mathsf{K}}(\mathit{Done}\ ())) \tag{1}$$

$$(\mathit{evH}\ e) = \mathit{PauseHi}\ (\mathsf{u}_H[x \mapsto 1] \gg_{\mathsf{K}} \eta_{\mathsf{K}}(\mathit{Done}\ ())) \tag{2}$$

Note that the assignment in e is mapped by *evL* and *evH* into lifted operations on *different* monads (i.e., Lo and Hi in (1) and (2), resp.). Then, this security

[5] The *evH* semantics is obtained from *evL* by replacing all "L"-suffixed operations (e.g., "u_L") by their corresponding "H"-suffixed operations (e.g., "u_H").

assignment is maintained in the resumption-trace by the *PauseLo* and *PauseHi* tags. Because u_L and u_H operate on different states, these assignments can not interfere (in the sense of Section 4.2). The monadic structure is the key to making this approach work.

Below are two schedulers for the basic separation model, *withHi* and *withoutHi*, for a simple monadic event system. The schedule ($withHi\ lo\ hi$) creates a Lo and Hi threads from event behaviors lo and hi, resp., in a round-robin fashion. Schedule ($withoutHi\ lo$) creates a single Lo thread.

$$
\begin{array}{ll}
withHi\ :\ Event \to Event \to \mathsf{R}\,() & withoutHi\ :\ Event \to \mathsf{R}\,() \\
withHi\ lo\ hi\ =\ weave\ (evL\ lo)\ (evH\ hi) & withoutHi\ lo\ =\ evL\ lo \\
weave\ :\ \mathsf{R}\,() \to \mathsf{R}\,() \to \mathsf{R}\,() & \\
weave\ (PauseLo\ \varphi)\ (PauseHi\ \gamma) = & \\
\quad PauseLo\ (\varphi \star_{\mathsf{K}} \lambda r_{lo}.\ \eta_{\mathsf{K}}(PauseHi\ (v \star_{\mathsf{K}} \lambda r_{hi}.\eta_{\mathsf{K}}(weave\ r_{lo}\ r_{hi})))) &
\end{array}
$$

5.3 Security for this setting: *take separation*

This section develops a non-interference style specification of process separation for monadic event systems. The question we answer is: given a resumption computation representing a schedule of threads on separated domains, how do we specify that those processes do not interfere? The answer we provide in this section adapts a technique for proving properties of streams to the resumption-monadic setting.

A common technique for proving a property of infinite lists is to show that the property holds of all finite approximations (i.e., finite initial prefixes) of the list. A well-known version of this technique is the *take lemma* [3, 9]:

$$s_1 = s_2 \ \Leftrightarrow\ \forall\ (n < \omega).\ take\ n\ s_1 = take\ n\ s_2$$

where $(take\ n\ [v_1, \ldots, v_n, \ldots]) = [v_1, \ldots, v_n]$. To show two streams s_1 and s_2 are equal using the take lemma, one shows that, for any non-negative integer n, each length n prefix of s_1 and s_2 are equal.

The security property proved here is analogous to the take lemma—hence its name. If, for any initial sequence of interleaved Hi and Lo events with n Lo events obtained from a system execution:

$$h_0\ ;\ l_0\ ;\ldots;\ h_n\ ;\ l_n$$

the effect of its execution on the Lo state should be identical to that of executing the Lo events in isolation:

$$l_0\ ;\ldots;\ l_n$$

Using the denotational semantics, we make this notion precise:

$$[\![h_0\ ;\ l_0\ ;\ldots;\ h_n\ ;\ l_n]\!] \gg_{\mathsf{K}} maskNonLo = [\![l_0\ ;\ldots;\ l_n]\!] \gg_{\mathsf{K}} maskNonLo$$

Here, *maskNonLo* is a stateful operation on the K monad which overwrites all non-Lo states in the K monad. The particular definition of *maskNonLo* differs as the monadic event system is refined, but for our case, it is merely

liftHi($\mathsf{u}_H\ (\lambda\_.h_0)$) The operator *maskNonLo* plays the rôle of $\downarrow_{\mathsf{Lo}}$ in a trace-based event system. We define two additional helper functions, *takeLo* and *run*. The helper function *takeLo* picks out the initial sequences containing n Lo events, and the *run* function projects schedulings in the resumption monad R back into the kernel monad K for execution:

$$
\begin{array}{l}
takeLo : Integer \to \mathsf{R}() \to \mathsf{R}() \\
takeLo\ 0\ x \qquad\qquad\quad = Done() \\
takeLo\ n\ (PauseLo\ \varphi) = \\
\qquad PauseLo\ (\varphi \star_{\mathsf{K}} (\eta_{\mathsf{K}} \circ (takeLo\ (n-1)))) \\
takeLo\ n\ (PauseHi\ \varphi) = \\
\qquad PauseHi\ (\varphi \star_{\mathsf{K}} (\eta_{\mathsf{K}} \circ (takeLo\ n)))
\end{array}
\qquad
\begin{array}{l}
run\ ::\ \mathsf{R}\,a \to \mathsf{K}\,a \\
run\ (Done\ v) \quad = \eta_{\mathsf{K}} v \\
run\ (PauseLo\ \varphi) = \varphi \star_{\mathsf{K}} run \\
run\ (PauseHi\ \varphi) = \varphi \star_{\mathsf{K}} run
\end{array}
$$

We may now formulate and prove take separation for the Basic Separation system described in Section 5.2:

Theorem 5 (Take Separation). *Let $lo, hi \in Event$, then for all natural numbers n,*

$$
\begin{aligned}
&run\ (takeLo\ n\ (withoutHi\ (evL\ lo)))\ \gg\ initH \\
&\qquad = run\ (takeLo\ n\ (withHi\ (evL\ lo)\ (evH\ hi)))\ \gg\ initH
\end{aligned}
$$

where $initH = liftHi(\mathsf{u}_H\ (\lambda\_.h_0))$ for any arbitrary fixed $h_0 \in H$.

Proof of Theorem 5 by induction on n. All binds ($\star$) and units (η) are in the K monad.

Case $n = 0$.

$$
\begin{aligned}
&run\ (takeLo\ 0\ (withoutHi\ (evL\ lo))) \gg initH \\
&\quad_{\{\text{defn takeLo}\}} = run(Done\ ()) \gg initH \\
&\qquad\qquad = run\ (takeLo\ 0\ (withHi\ (evL\ lo)\ (evH\ hi)))\ \gg\ initH
\end{aligned}
$$

Case $n = k + 1$.

$$
\begin{aligned}
&run\ (takeLo\ (k+1)\ (withHi\ (evL\ lo)\ (evH\ hi)))\ \gg\ initH \\
&\qquad\qquad = (l_1 \gg h_1 \gg \ldots \gg l_{(k+1)} \gg h_{(k+1)} \gg \eta())\ \gg\ initH \\
&\quad_{\{\text{right unit}\}} = l_1 \gg h_1 \gg \ldots \gg l_{(k+1)} \gg h_{(k+1)} \gg initH \\
&\quad_{\{\text{clobber}\}} = l_1 \gg h_1 \gg \ldots \gg l_{(k+1)} \gg initH \\
&\quad_{\{l_{(k+1)} \# initH\}} = l_1 \gg h_1 \gg \ldots \gg initH \gg l_{(k+1)} \\
&\quad_{\{\text{ind. hyp.}\}} = \underbrace{l_1 \gg \ldots}_{h_i \text{ excised}} \gg initH \gg l_{(k+1)} \\
&\quad_{\{l_i \# initH\}} = l_1 \gg \ldots \gg l_{(k+1)} \gg initH \\
&\quad_{\{\text{right unit}\}} = l_1 \gg \ldots \gg l_{(k+1)} \gg \eta() \\
&\qquad\qquad = run\ (takeLo\ (k+1)\ (withoutHi\ (evL\ lo)))
\end{aligned}
$$

6 Conclusion

Monads and monad transformers provide a powerful framework for the structuring of high-assurance software systems because they permit an explicit yet flexible partitioning of the domains-of-effect within a computation. Pure, higher-order, typed languages that support this style of programming, such as Haskell, must be considered as modeling and implementation languages for complex, security-critical software systems.

Monads were originally introduced into pure, higher-order functional programming languages to allow principled, effectful programming. Monads permitted functional programmers to program imperatively without imperative languages. This work demonstrates that the monadic approach to effects has benefits far beyond the mere imitation of imperative-style; the precise scoping of effects provided by monads allows information-flow to be managed in a straightforward yet mathematically rigorous manner. How such scoping of effects within imperative languages might be achieved remains an open question.

This paper presents an overview of an approach for constructing and formally verifying domain separation for systems developed in pure, higher-order, typed languages such as Haskell. The approach relies fundamentally on the ability of the language to express all internal effects using monads and to express monad transformers as a functional construction. Though we use the pure functional language Haskell as our implementation language, we do not use laziness or type classes in any essential way; thus our techniques could be applied using the *pure* subset of ML. Note that the purity of the language *is* essential to our approach.

In this paper we have taken a basic model of concurrency, reflected it inside a programming language type system, and exploited it to build a rich model of provably non-interfering concurrent execution. We have refined this to the point where basic mechanisms for process communication are provided.

This work is being done as part of the Programatica project at OGI, where we are developing tools and techniques to support high-assurance software development in Haskell. These techniques are being applied in the validation of the OSKer security kernel—a novel high-assurance operating system.

The models presented in this paper have been developed in the context of the development of a multi-level secure operating system written in Haskell called OSKer. Although the verification of OSKer's security properties are not complete, the challenges of that verification have inspired this formalism. In particular, this formalism is rich enough to be extended to realistic system architectures in which trusted processes (such as cryptographic servers) are allowed to reclassify information. When such extensions are introduced the verification of system assurance becomes more complex, and involves some local application of traditional methods, but the overall system validation is still modular and scalable.

Because monad transformers give a notion of composable formal specification [17, 15], monad structuring of the event system allows modification without complete re-verification. Scalable techniques for the development of complex systems with high confidence in their security properties remains a grand challenge

of computer science. We believe that the ultimate solution to this challenge will draw heavily from the theory and practice of pure, higher-order, typed languages.

References

1. D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunder, S. Nettles, and J. Smith. The switchware active network architecture, 1998.
2. W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
3. Richard J. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall Series in Computer Science. Prentice-Hall Europe, London, UK, second edition, 1998.
4. 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)*, Hilton Head, South Carolina USA, January 2000.
5. W.E. Boebert, W.D. Young, R.Y. Kain, and S.A. Hansohn. Secure ada target: Issues, system design, and verification. In *Proc. IEEE Symposium on Security and Privacy*, pages 176–183, 1985.
6. Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs. *Lecture Notes in Computer Science*, 2076:382+, 2001.
7. Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.
8. Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (or revenge of the son of the lisp machine). In *International Conference on Functional Programming*, pages 138–147, 1999.
9. Jeremy Gibbons and Graham Hutton. Proof methods for structured corecursive programs. In *1st Scottish Functional Programming Workshop, Stirling, Scotland*, August 1999.
10. 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.
11. J.A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symposium on Security and Privacy*, pages 75–86, 1984.
12. J.W. Gray III. Probabilistic interference. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 170–179, 1990.
13. J.T. Haigh and W.D. Young. Extending the non-interference version of MLS for SAT. In *Proc. IEEE Symposium on Security and Privacy*, pages 232–239, 1986.
14. Robert Harper, Peter Lee, and Frank 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. (Also published as Fox Memorandum CMU-CS-FOX-98-02).
15. William Harrison. *Modular Compilers and Their Correctness Proofs*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.
16. Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998*, pages 365–377, New York, NY, USA, 1998. ACM Press.

17. Sheng Liang. *Modular Monadic Semantics and Compilation*. PhD thesis, Yale University, 1998.
18. Sheng Liang, Paul Hudak, and Mark 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.
19. E. J. McCauley and P. J. Drongowski. KSOS—the design of a secure operating system. In *Proc. AFIPS National Computer Conference*, volume 48, pages 345–353, 1979.
20. D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symposium on Security and Privacy*, pages 177–187, 1988.
21. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.
22. Eugenio 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.
23. Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ., 1990.
24. P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proof. Technical Report CSL-116, SRI, May 1980.
25. Nikos S. Papaspyrou. A resumption monad transformer and its applications in the semantics of concurrency. In *Proceedings of the 3rd Panhellenic Logic Symposium*, Anogia, Crete, 2001.
26. G. D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3), 1976.
27. François Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 319–330, Portland, Oregon, January 2002.
28. John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI, dec 1992.
29. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.
30. David A. Schmidt. *Denotational Semantics*. Allyn and Bacon, Boston, 1986.
31. Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.
32. G. Smith. A new type system for secure information flow, 2001.
33. Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 355–364, New York, January 1998. ACM.
34. Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.
35. Philip Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.
36. 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.

37. Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the ucla unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.
38. Mitchell Wand. Continuation-based multiprocessing. In J. Allen, editor, *Conference Record of the 1980 LISP Conference*, pages 19–28, Palo Alto, CA, 1980. The Lisp Company.
39. P. White, W. Martin, A. Goldberg, and F.S. Taylor. Formal construction of the mathematically analyzed separation kernel. In *The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 133–141, September 2000.
40. Aris Zakinthinos and E. S. Lee. The composability of non-interference. In *Proceedings of the Eighth Computer Security Foundations Workshop (CSFW '95)*, pages 2–8, Washington - Brussels - Tokyo, June 1995. IEEE.
41. Aris Zakinthinos and E. Stewart Lee. A general theory of security properties. In *Proceedings of the 18th IEEE Computer Society Symposium on Research in Security and Privacy*, 1997.

A Theorems and Proofs

Theorem 1 (StateT creates a state monad) *For any monad* M*, let monad* $\mathsf{M}' = \mathsf{StateT}\ sto\ \mathsf{M}$ *and also* $u : (sto \to sto) \to \mathsf{M}'()$ *and* $g : \mathsf{M}'\ sto$ *be the non-proper morphisms added by* ($\mathsf{StateT}\ sto$)*. Then* $\langle \mathsf{M}', \eta_{\mathsf{M}'}, \star_{\mathsf{M}'}, u, g, sto \rangle$ *is a state monad.*

Proof.
Case: Sequencing.

$$
\begin{array}{ll}
\mathsf{u}f \star_{\mathsf{M}'} \lambda\_.\mathsf{u}g & \\
= \lambda\sigma_0.(\mathsf{u}f)\sigma_0 \star_{\mathsf{M}} (\lambda\langle v, \sigma_1\rangle.(\lambda\_.\mathsf{u}g)\, v\, \sigma_1) & \{\text{def. } \star_{\mathsf{M}'}\} \\
= \lambda\sigma_0.(\mathsf{u}f)\sigma_0 \star_{\mathsf{M}} (\lambda\langle v, \sigma_1\rangle.(\mathsf{u}g)\, \sigma_1) & \{\beta\} \\
= \lambda\sigma_0.(\lambda\sigma.\eta_{\mathsf{M}}\langle(), f\sigma\rangle)\sigma_0 \star_{\mathsf{M}} (\lambda\langle v, \sigma_1\rangle.(\lambda\sigma.\eta_{\mathsf{M}}\langle(), g\sigma\rangle)\, \sigma_1) & \{\text{def. } \mathsf{u}(\times 2)\} \\
= \lambda\sigma_0.(\eta_{\mathsf{M}}\langle(), f\sigma_0\rangle) \star_{\mathsf{M}} (\lambda\langle v, \sigma_1\rangle.(\eta_{\mathsf{M}}\langle(), g\sigma_1\rangle)) & \{\beta(\times 2)\} \\
= \lambda\sigma_0.(\eta_{\mathsf{M}}\langle(), g(f\sigma_0)\rangle)) & \{\text{left unit}\} \\
= \mathsf{u}(f; g) & \{\text{def. } \mathsf{u}(\times 2)\}
\end{array}
$$

Case: Cancellation.

$$
\begin{array}{ll}
\mathsf{g} \star_{\mathsf{M}'} \lambda s.\mathsf{u}(\lambda\_.s_0) & \\
= \lambda\sigma.\eta_{\mathsf{M}}\langle\sigma, \sigma\rangle \star_{\mathsf{M}} \lambda\langle v, \sigma_1\rangle.\ (\lambda s.\lambda\sigma'.\eta_{\mathsf{M}}\langle(), s_0\rangle)v\ \sigma_1 & \{\text{def. } \mathsf{g}, \mathsf{u}, \star_{\mathsf{M}}\} \\
= \lambda\sigma.\eta_{\mathsf{M}}\langle\sigma, \sigma\rangle \star_{\mathsf{M}} \lambda\langle v, \sigma_1\rangle.\ (\eta_{\mathsf{M}}\langle(), s_0\rangle) & \{\beta\} \\
= \lambda\sigma.\ (\eta_{\mathsf{M}}\langle(), s_0\rangle) & \{\text{left unit}\} \\
= \mathsf{u}(\lambda\_.s_0) & \{\text{def.}\mathsf{u}\}
\end{array}
$$

□

Theorem 3 (StateT preserves stateful behavior) *For any state monad* $\mathsf{M} = \langle \mathsf{M}, \eta, \star, u, g, sto \rangle$*, the following state monad structure is a state monad:*

$$\langle \mathsf{StateT}\ s\ \mathsf{M}, \eta', \star', (u; \mathrm{lift}), \mathrm{lift}(g) \rangle$$

where η'*,* $\star'$*, and* lift *are the monadic unit, bind, and lifting operations, respectively, defined by* ($\mathsf{StateT}\ s$) *in Section 3.1.*

Proof. Let $\mathsf{M}' = \mathsf{T}\,\mathsf{M}$ below. Each step follows from the *Lifting Laws* listed in Section 3.1.
Case: Sequencing.

$$\mathrm{lift}(\mathsf{u}f) \star_{\mathsf{M}'} \lambda\_.\mathrm{lift}(\mathsf{u}g) = \mathrm{lift}(\mathsf{u}f \star_{\mathsf{M}} \lambda\_.\mathsf{u}g) = \mathrm{lift}(\mathsf{u}(f;g))$$

Case: Cancellation.

$$\mathrm{lift}(\mathsf{g}) \star_{\mathsf{M}'} \lambda s.\mathrm{lift}(\mathsf{u}(\lambda\_.s_0)) = \mathrm{lift}(\mathsf{g} \star_{\mathsf{M}} \lambda s.\mathsf{u}(\lambda\_.s_0)) = \mathrm{lift}(\mathsf{u}(\lambda\_.s_0))$$

□

Theorem 4 (Non-interference of updates) *Define state monads* M *and* M' *as the following:*

$$\mathsf{M} = \langle \mathsf{M}, \eta_{\mathsf{M}}, \star_{\mathsf{M}}, \mathsf{u}_A, \mathsf{g}_A, A \rangle$$
$$\mathsf{M}' = \langle \mathsf{StateT}\ B\ \mathsf{M}, \eta_{\mathsf{M}'}, \star_{\mathsf{M}'}, \mathsf{u}_B, \mathsf{g}_B, B \rangle$$

Then, for all $\delta_A : A \to A$ *and* $\delta_B : B \to B$,

$$(\mathsf{u}_B\ \delta_B) \#_{\mathsf{M}'\,()} \mathrm{lift}(\mathsf{u}_A\ \delta_A)$$

Proof.

$$\begin{aligned}
&(\mathsf{u}_B\ \delta_B) \gg_{\mathsf{M}'} \mathrm{lift}(\mathsf{u}_A\ \delta_A) \\
&= \lambda\sigma_0.((\mathsf{u}_B\ \delta_B)\ \sigma_0) \star_{\mathsf{M}} \lambda\langle (), \sigma_1\rangle.(\lambda_.\mathrm{lift}(\mathsf{u}_A\ \delta_A))\ ()\ \sigma_1 \\
&= \lambda\sigma_0.((\mathsf{u}_B\ \delta_B)\ \sigma_0) \star_{\mathsf{M}} \lambda\langle (), \sigma_1\rangle.(\mathrm{lift}(\mathsf{u}_A\ \delta_A))\ \sigma_1 \\
&= \lambda\sigma_0.((\lambda\sigma.\eta_{\mathsf{M}}\langle (), \delta_B\ \sigma\rangle)\ \sigma_0) \star_{\mathsf{M}} \lambda\langle (), \sigma_1\rangle.(\mathrm{lift}(\mathsf{u}_A\ \delta_A))\ \sigma_1 \\
&= \lambda\sigma_0.(\eta_{\mathsf{M}}\langle (), \delta_B\ \sigma_0\rangle) \star_{\mathsf{M}} \lambda\langle (), \sigma_1\rangle.(\mathrm{lift}(\mathsf{u}_A\ \delta_A))\ \sigma_1 \\
&= \lambda\sigma_0.(\mathrm{lift}(\mathsf{u}_A\ \delta_A))\ (\delta_B\ \sigma_0) \\
&= \lambda\sigma_0.(\lambda\sigma.(\mathsf{u}_A\ \delta_A) \star_{\mathsf{M}} \lambda v.\eta_{\mathsf{M}}\langle v, \sigma\rangle)\ (\delta_B\ \sigma_0) \\
&= \lambda\sigma_0.(\mathsf{u}_A\ \delta_A) \star_{\mathsf{M}} \lambda v.\eta_{\mathsf{M}}\langle v, \delta_B\ \sigma_0\rangle) \\
&= \lambda\sigma_0.(\mathsf{u}_A\ \delta_A) \star_{\mathsf{M}} \lambda v.(\lambda\sigma.\eta_{\mathsf{M}}\langle v, \delta_B\ \sigma\rangle)\ \sigma_0 \\
&= \lambda\sigma_0.(\mathsf{u}_A\ \delta_A) \star_{\mathsf{M}} \lambda v.(\mathsf{u}_B\ \delta_B)\ \sigma_0 \\
&= \lambda\sigma_0.(\lambda\sigma.(\mathsf{u}_A\ \delta_A \star_{\mathsf{M}} \lambda_.\eta_{\mathsf{M}}\langle (), \sigma\rangle)\ \sigma_0 \star_{\mathsf{M}} \lambda\langle v, \sigma\rangle.(\mathsf{u}_B\ \delta_B)\ \sigma \\
&= \lambda\sigma_0.(\mathrm{lift}(\mathsf{u}_A\ \delta_A))\ \sigma_0 \star_{\mathsf{M}} \lambda\langle v, \sigma\rangle.(\mathsf{u}_B\ \delta_B)\ \sigma \\
&= (\mathrm{lift}(\mathsf{u}_A\ \delta_A)) \star_{\mathsf{M}} \lambda_.(\mathsf{u}_B\ \delta_B)
\end{aligned}$$

□

Theorem 5 (Non-interference Preserved by Monad Transformers) *Let* M *be a monad with two operations,* $a : \mathsf{M}\,()$ *and* $b : \mathsf{M}\,()$ *such that* $a \#_{\mathsf{M}} b$. *Then,*

$$(\mathrm{lift}\ a) \#_{(\mathsf{T}\,\mathsf{M})} (\mathrm{lift}\ b)$$

where T *is a monad transformer and* $\mathrm{lift} : \mathsf{M}\,a \to (\mathsf{T}\,\mathsf{M})\,a$.

Proof.

$$
\begin{aligned}
&\mathrm{lift}\, a \star_{(\mathsf{TM})} \lambda\_.\mathrm{lift}\, b \\
&= \mathrm{lift}(a \star_{\mathsf{M}} \lambda\_.b) \\
&= \mathrm{lift}(b \star_{\mathsf{M}} \lambda\_.a) \\
&= \mathrm{lift}\, b \star_{(\mathsf{TM})} \lambda\_.\mathrm{lift}\, a
\end{aligned}
$$

□