Electronic Notes in Theoretical Computer Science 90 No. 1 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume90.html 21 pages

Domain Separation by Construction

William Harrison [1] Mark Tullsen [2] James Hook [3]

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

Abstract

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 system model. 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 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 outline the development by stepwise refinements of an operating system kernel supporting both standard Unix-like system calls (e.g., fork, sleep, etc.) and a formally verified security policy (domain separation). Previous work has focused on languages with explicit or implicit imperative features, such as Java or ML; our approach assumes a purely functional language in which all imperative features are captured by monads. 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.

[1] Email: wlh@cse.ogi.edu
[2] Email: mtullsen@cse.ogi.edu
[3] Email: hook@cse.ogi.edu

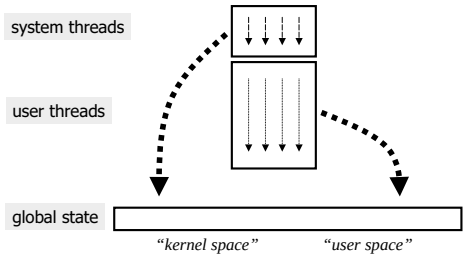

Fig. 1. *Shared-state concurrency with global state*: Threads access the same global state, potentially interfering in ways difficult to control.

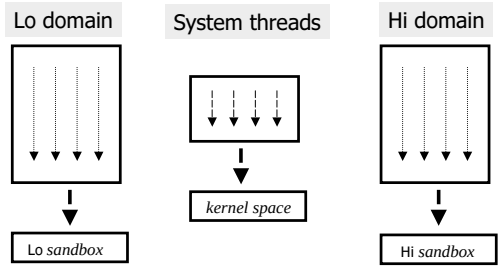

Fig. 2. *Concurrency with Domain Separation*: Scoping of effects is achieved by partitioning the global state into separate sandboxes.

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 produce secure systems. 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 2. 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 [10,19,40,20,28] 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 feature 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 [21,16,23], functional programming [36], and software verification [14]. 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.

1.1 A Monadic Model for Security

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 (written *Beh*) and their denotational semantics. For the sake of simplicity, we assume there are only two separated domains, Hi and Lo, corresponding to two security levels of the same name. However, all of our results generalize easily to n separate domains.

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 *Beh*-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:

$$[\![-]\!] : (Beh_{\mathsf{Lo}} + Beh_{\mathsf{Hi}}) \to \mathsf{R}() \tag{1}$$

where R is a monad encapsulating imperative effects and a notion of interleaving concurrency called resumptions[25,21,23]. The monad R is constructed especially to isolate Hi, Lo, and kernel effects from one another.

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:

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

The precise definition of the security policy, using the above denotational semantics, is developed in Section 5.

1.2 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. Achieving process separation via partitioning, sometimes referred to as *sandboxing*, seems to have originated in the work of Rushby [28,27]. 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 3. Corresponding to the security levels Hi and Lo are separate domains (marked **(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 (marked **(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—and precisely how is described in detail in Section 4 below.

1.3 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 depicted in Figure 4, which illustrates both meanings of "by construction":

- **Stepwise-refinement of System.** The vertical axis of Figure 4 measures the richness of system behaviors, and each step along that axis marks an addition to those behaviors. At point 0, only a single, monolithic process domain exists and all thread scheduling is static. At point 1, threads are 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 intra-domain synchronization 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 asynchronous message-passing 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 3. 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 of differing security levels. The "by construction" properties of monads and monad transformers are extremely useful in formally demonstrating domain separation.

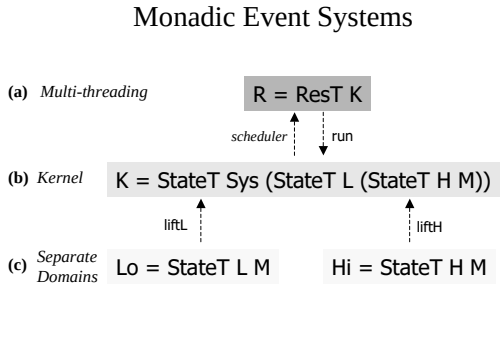

Fig. 3. Observable events created in isolated security domains Lo and Hi are lifted to the kernel K. Laws about monad transformers (StateT and ResT) aid in proving domain separation.

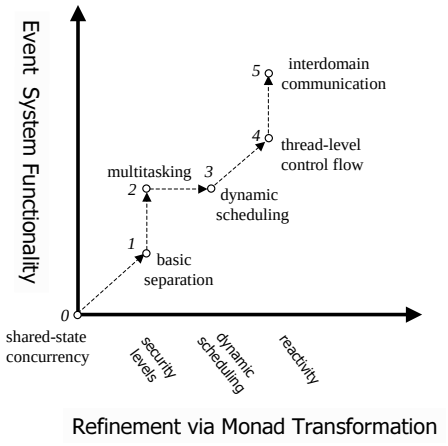

Fig. 4. *Separation by construction*: System is enhanced through refinement of monad transformers ; "by construction" properties of monad transformers aid in verifying separation.

1.4 Overview

Section 2 surveys related work and Section 3 summarizes the necessary background on monads, monad transformers, and resumption based concurrency 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 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. Section 6 gives a brief overview of the development path from system 1 to system 5 in Figure 4. Finally, Section 7 summarizes the present work and outlines future directions.

2 Related Work

Formulating security policies in terms of non-interference goes back to the work of Goguen and Meseguer [10,9] and our notion of domain separation is based on theirs. Haigh and Young [12] and Rushby [29] 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 [34,6] and probabilistic models [11].

Many formal security models—including non-interference and separability—are formulated in terms of *event systems* [10,19,40,20]. While there are a vari-

ety 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 [4].

Security policies may then be formulated as relationships between event traces which specify permitted system executions. These 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* [20,40] 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.

The research reported here is inspired by Rushby's work [28,27] on abstract operating systems called *separation kernels*. A separation kernel enforces process isolation by partitioning the state into separate user spaces (called "colours"), allowing reasoning about the processes as if they were physically distributed. The security levels Hi and Lo used in the present work correspond to colours. Separation kernels are then specified using finite-state machines, and separability (i.e., that differently-colored processes do not interfere) is characterized in terms of event traces arising from executions of these machines.

In the context of the Programatica project at OGI we are 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. One approach to reducing the enormity of this task has been to use type-systems for information-flow. There has been a growing emphasis on such *language-based* techniques for information flow security [35,34,15,33,26] (see [30] 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: 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) but which would be rejected by the type system.

Moggi was the first to observe that the categorical structure known as a monad was appropriate for the development of modular semantic theories for programming languages [21]. 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

[4] Some formulations also distinguish subsets of E as *input* and *output* events.

resumption monad [21]. That model of concurrency is used extensively in the present work. 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 [24].

Modeling concurrency by resumptions was introduced by Plotkin [25,31]. Moggi showed how resumptions could be modeled with monads [21] and our formulation of resumptions in terms of monad transformers is that of Papaspyrou [23]. Although continuations can also model concurrency [38,7,8], resumptions can be viewed as a disciplined use of continuations which allows for simpler reasoning about our system.

There have been many previous attempts to develop secure OS kernels: PSOS [22], KSOS [18], UCLA Secure Unix [37], SAT [5], KIT [2], EROS [32], and MASK [39] among others. There has also been work using functional languages to develop high confidence system software: the Fox project at CMU [13] 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 [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.

3 Background

Some familiarity with monads and their uses in the denotational semantics of languages with effects is essential to understanding the current work. We assume of necessity that the reader possesses such familiarity. This section serves as a quick review of the basic concepts of monads and monadic semantics, and readers requiring more should consult the references for further background. An informal introduction to monads can be found in Wadler [36]. A formal treatment of monads and their rôle in denotational semantics is in Moggi [21].

Monads [21] can be understood as abstract data types (ADTs) for defining languages and programs. 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* [21,17,16]. 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 monads and monad transformers, we will then describe the resumption monad transformer.

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 analogue of the identity function, injecting a value into the monad. The $\star$ operator gives a form of sequential application. 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:

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

The lambda null binding (i.e., "λ_") acts as a dummy variable, ignoring the value produced by x. The identity monad $\langle \mathsf{Id}, \eta_{\mathsf{Id}}, \star_{\mathsf{Id}} \rangle$ is defined as follows:

$$\mathsf{Id}\ a = a \qquad x \star_{\mathsf{Id}} k = k\ x \qquad \eta_{\mathsf{Id}}\ x = x$$

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

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

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 non-proper morphisms.

There are various formulations of monad transformers, we follow that given in [17] 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 lift_{T}. The function $\text{lift}_{\mathsf{T}} : \mathsf{M}\, a \to \mathsf{T}\,\mathsf{M}\, a$ performs a "lifting" of computations in M to computations in $(\mathsf{T}\,\mathsf{M})$; and will generally satisfy the *Lifting Laws* [16]:

$$\text{lift} \circ \eta_{\mathsf{M}} = \eta_{(\mathsf{T}\,\mathsf{M})} \qquad \text{lift}(x \star_{\mathsf{M}} f) = (\text{lift}\ x) \star_{(\mathsf{T}\,\mathsf{M})} (\text{lift} \circ f)$$

These laws ensure that a monad transformer adds features without changing existing 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 (see Definition 3.1). Note that the monad transformer $\mathsf{StateT}\ s$ applied to the identity monad Id gives the original $\mathsf{St}\ s$ monad.

In Definitions 3.1 and 3.2, the unit and bind of the new monads are η and $\star$, respectively, while those of the transformed monad M are subscripted.

Definition 3.1 (State Monad Transformer) *For monad* M *and type* s*,*

$$\begin{array}{lll}
\mathsf{M}'\,a = \mathsf{StateT}\;s\;\mathsf{M}\,a = s \to \mathsf{M}(a \times s) & & u \;\; : \; (s \to s) \to \mathsf{M}'() \\
\mathrm{lift}\,x \;\; = \lambda\sigma.\, x \star_{\mathsf{M}} \lambda y.\, \eta_{\mathsf{M}}(y, \sigma) & & u\;\Delta \;= \lambda\sigma.\, \eta_{\mathsf{M}}((), \Delta\,\sigma) \\
\eta\,x \;\; = \lambda\sigma.\, \eta_{\mathsf{M}}(x, \sigma) & & g \;\; : \; \mathsf{M}'s \\
x \star f \; = \lambda\sigma_0.\, (x\,\sigma_0) \star_{\mathsf{M}} (\lambda(a, \sigma_1).f\;a\,\sigma_1) & & g \;\; = \lambda\sigma.\, \eta_{\mathsf{M}}(\sigma, \sigma)
\end{array}$$

The following resumption monad transformer provides the foundation for the basic concurrency model here [23], corresponding to point 0 in Figure 4. To avoid confusion, we label resumption monad transformers by their corresponding point number from Figure 4. Section 3.2 explains the intuitions behind this monad transformer.

Definition 3.2 (Resumption Monad Transformer) *For monad* M*,*

$$\begin{array}{lll}
\mathsf{ResT}_0\;\mathsf{M}\,a = \mu\mathrm{R.D}\,a + \mathrm{P}\,(\mathsf{M}(\mathrm{R}\,a)) & & \\
step\,\phi \; = \mathrm{P}(\phi \star_{\mathsf{M}} \lambda v.\eta_{\mathsf{M}}(\mathrm{D}\,v)) & & (\mathrm{D}\,v) \star f \;\; = f\,v \\
\eta\,x \;\; = \mathrm{D}\,x & & (\mathrm{P}\;m) \star f \; = \mathrm{P}(m \star_{\mathsf{M}} \lambda r.\eta(r \star_{\mathsf{M}} f))
\end{array}$$

3.2 Resumption Based Concurrency

This section introduces concurrency based on the resumption monad transformer, the definition of which can be found in Definition 3.2. How resumption based concurrency works is best explained by an example. We define a *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 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, 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 $\mathsf{ResT}_0\;\mathsf{M}\,()$ are the set of possible interleavings:

$$\begin{array}{l}
P\,(a_0 \gg \eta(P\,(a_1 \gg \eta(P\,(b_0 \gg \eta(D\,()))))))) \\
P\,(a_0 \gg \eta(P\,(b_0 \gg \eta(P\,(a_1 \gg \eta(D\,()))))))) \\
P\,(b_0 \gg \eta(P\,(a_0 \gg \eta(P\,(a_1 \gg \eta(D\,())))))))
\end{array}$$

Closer comparison of both versions of these interleaving semantics reveals a strong similarity. While the stream version implicitly uses a lazy "cons" operation $(h : t)$, the monadic version[5] uses something similar: $P(h \gg \eta t)$. This is important because threads may be infinite, and the laziness of P allows

[5] Read P as "pause" and D as "done."

infinite *computations* to be constructed in $(\mathsf{ResT}_0\,\mathsf{M})$ just as the laziness of cons in $(h : t)$ 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 [21,17], it is not computationally tractable. We choose instead to pick out a single interleaving via a scheduler.

4 "By Construction" Properties

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 3 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 delimit the effects of stateful operations on separate domains; this phenomenon—which we call *atomic non-interference*—is described in detail in Section 4.2.

Atomic non-interference is not only useful for proving the security property in Section 5.3, but it captures *precisely* what we mean by *monadic scoping of effects*. The monadic structure of the system semantics is the foundation on which domain separation is built.

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 state monads and preserves existing state monads.

Definition 4.1 (State Monad Structure) *The quintuple* $\langle \mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, s\rangle$ *is a* state monad structure *when* $\langle \mathsf{M}, \eta, \star\rangle$ *is a monad, and the* update *and* get *operations on* s *are:* $\mathsf{u} : (s \to s) \to \mathsf{M}()$ *and* $\mathsf{g} : \mathsf{M}\,s$.

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.

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

Definition 4.2 (State Monad Axioms) *Let* M *be the state monad structure* $\langle \mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, s\rangle$. M *is a* state monad *if the following equations hold for any* $f, f' : s \to s$,

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

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

Monad Hierarchy

$$
\begin{array}{lll}
\mathsf{G_L} = \mathsf{StateT}\ G\ \mathsf{M}, & \mathsf{u}_G & : (G \to G) \to \mathsf{G_L}() \\
\mathit{liftG} = \mathit{id} & \mathsf{g}_G & : \mathsf{G_L}\, G \\
\mathsf{K} = \mathsf{G_L}, & \mathit{stepG} & : \mathsf{K}\, a \to \mathsf{R}\, a \\
\mathsf{R} = \mathsf{ResT}_0\ \mathsf{K} & \mathit{stepG}\ \varphi = P_{\mathsf{Lo}}\ (\varphi \star_{\mathsf{K}} \lambda v.\eta_{\mathsf{K}}(D\ v)) &
\end{array}
$$

Semantics of $\mathsf{G_L}$ Language

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

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

Fig. 5. Shared-state Concurrency with Global State (system 0).

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.

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

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

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

Theorem 4.3 (StateT creates a state monad) *For monad* M*, let monad* $\mathsf{M}' = \mathsf{StateT}\, s\, \mathsf{M}$*, with non-proper morphisms* u *and* g *added by* $(\mathsf{StateT}\ s)$*. Then* $\langle \mathsf{M}', \eta_{\mathsf{M}'}, \star_{\mathsf{M}'}, \mathsf{u}, \mathsf{g}, s \rangle$ *is a state monad.*

Theorem 4.4 (StateT preserves stateful behavior) *For any state monad* $\mathsf{M} = \langle \mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, s \rangle$*, the structure* $\langle \mathsf{StateT}\, s\, \mathsf{M}, \eta', \star', \mathrm{lift} \circ u, \mathrm{lift}\, g, s \rangle$ *is also a state monad, 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 3).

Definition 4.5 (Atomic Non-interference) *For monad* M *with bind operation* $\star$*, define the atomic non-interference relation* $\# \subseteq \mathsf{M}\,() \times \mathsf{M}\,()$ *so that, for* $\varphi, \gamma : \mathsf{M}\,()$*,* $\varphi \# \gamma$ *holds if and only if the equation* $\varphi \gg \gamma = \gamma \gg \varphi$ *holds.*

Theorem 4.6 *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 4.3,* $\mathsf{M}' = \langle \mathsf{StateT}\, B\, \mathsf{M}, \eta_{\mathsf{M}'}, \star_{\mathsf{M}'}, \mathsf{u}_B, \mathsf{g}_B, B \rangle$ *is also a state monad. Then, for all* $\delta_A : A \to A$ *and* $\delta_B : B \to B$*,* $(\mathsf{u}_B\, \delta_B) \#_{\mathsf{M}'} \mathrm{lift}(\mathsf{u}_A\, \delta_A)$ *holds.*

Theorem 4.7 *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* $\mathrm{lift} : \mathsf{M}\, a \to (\mathsf{T}\,\mathsf{M})\, a$*.*

5 Basis for a Secure OS Kernel

In monadic event systems, behaviors are programs in the *Beh* 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 behaviors, *Beh*, are presented below. The *Beh* language contains sufficient expressiveness to allow for potentially infinite streams of operations because it includes loops. Importantly, it allows for expression of potentially interfering programs as well.

Definition 5.1 (Abstract Syntax for *Beh***)** *Below is a BNF syntax for Beh:*

$$\begin{aligned} \mathit{Beh} &::= \mathit{Var} \texttt{:=} \mathit{Exp} \mid \texttt{skip} \mid \mathit{Beh}\texttt{;}\mathit{Beh} \mid \\ &\qquad \texttt{ite}\ \mathit{Exp}\ \mathit{Beh}\ \mathit{Beh} \mid \texttt{while}\ \mathit{Exp}\ \texttt{do}\ \mathit{Beh} \\ \mathit{Exp} &::= \mathit{Var} \mid \mathit{Int} \end{aligned}$$

5.1 Global Shared-State Concurrency

In this section, 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 4.

Using the monad transformers ResT_0 and StateT (defined in Section 3), we define, for any monad M, the monad hierarchy for shared-state concurrency with global state. This construction is summarized in Figure 5. Unlike the system pictured in Figure 3, this monadic event system does not provide separation, and so it has only one global domain (called G_L for "global"). In Figure 5, state type G is $\mathit{Name} \to \mathit{Int}$.

Associated with these monad constructions are a number of non-proper morphisms. The morphism u_G applies a state-to-state map to the current G state in the G_L monad, while the morphism g_G reads and returns the current G state. The u_G and g_G morphisms are defined by an application of the StateT monad transformer. The lifting *liftG* reinterprets G_L computations in

Monad Hierarchy & Lo Morphisms

$$
\begin{array}{ll}
\mathsf{Hi} = \mathsf{StateT}\ H\ \mathsf{M} & \mathsf{u}_L \quad : (L \to L) \to \mathsf{Lo}() \\
\mathsf{Lo} = \mathsf{StateT}\ L\ \mathsf{M} & \mathsf{g}_L \quad : \mathsf{Lo}\, L \\
\mathsf{K} = \mathsf{StateT}\ H\ (\mathsf{StateT}\ L\ \mathsf{M}) & \mathit{liftL} \quad : \mathsf{Lo}\, a \to \mathsf{K}\, a \\
\mathsf{R} = \mathsf{ResT}_1\ \mathsf{K} & \mathit{stepL} \quad : \mathsf{K}\, a \to \mathsf{R}\, a \\
 & \mathit{stepL}\ \varphi = P_{\mathsf{Lo}}\ (\varphi \star_{\mathsf{K}} \lambda v.\eta_{\mathsf{K}}(D\ v))
\end{array}
$$

Semantics of Lo Language

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

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

Fig. 6. Shared-State Concurrency with Basic Separation (system 1). The *evH* semantics (not shown) is obtained from *evL* by replacing all "L"-suffixed operations (e.g., "u_L") by their corresponding "H"-suffixed operations (e.g., "u_H"). There are also morphisms u_H, g_H, *liftH*, *stepH*, and *loadH* as well.

the kernel monad K, and because the kernel monad K is just G_L in this example, *liftG* is merely identity. The morphism *step* creates a "paused" K computation and results from the application of the ResT_0 transformer.

The resumption monadic denotational semantics for system behaviors, *ev*, is presented in Figure 5. It is just what one would expect given recent work on the resumption monadic semantics of concurrency [23]. Please note that because each observable event (namely u or g) is wrapped by a *load*, each such event is paused with *P*. The *run* morphism for this monadic event system, which projects resumption computations in R to the runtime platform K (see Figure 3), is defined as:

$$
\begin{array}{l}
\mathit{run} :: \mathsf{R}\, a \to \mathsf{K}\, a \\
\mathit{run}\ (D\ v) = \eta_{\mathsf{K}} v \\
\mathit{run}\ (P\ \varphi) = \varphi \star_{\mathsf{K}} \mathit{run}
\end{array}
$$

5.2 Basic Separation System

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 [23] to reflect the Hi and Lo security levels. The refined resumption monad transformer is:

$$\mathsf{ResT}_1\,\mathsf{M}\,a = \mu\mathsf{R}.D\,a + P_{\mathsf{Lo}}\,(\mathsf{M}(\mathsf{R}\,a)) + P_{\mathsf{Hi}}\,(\mathsf{M}(\mathsf{R}\,a))$$

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

$$\begin{aligned}(D\ v) \star f &= f\ v\\ (P_{\mathsf{Lo}}\ \varphi) \star f &= P_{\mathsf{Lo}}\ (\varphi \star_{\mathsf{M}} \lambda r.\ \eta_{\mathsf{M}}(r \star f))\\ (P_{\mathsf{Hi}}\ \varphi) \star f &= P_{\mathsf{Hi}}\ (\varphi \star_{\mathsf{M}} \lambda r.\ \eta_{\mathsf{M}}(r \star f))\end{aligned}$$

Using this refined resumption transformer and StateT, we define the new monad hierarchy and monadic event system in Figure 6 for any monad M. Here, L and H are state types equal to $Name \to Int$. Associated with these monad constructions are a number of non-proper morphisms also shown in Figure 6. 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_1 transformer.

By Theorems 4.3 and 4.4, 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:

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

There are two semantics for *Beh* corresponding to the Hi and Lo security levels—these are *evH* and *evL*, respectively.

The following example illustrates how the monadic structure is key to making this approach work. The semantics *evL* (*evH*) creates traces by injecting the Lo (Hi) operations into the P_{Lo} (P_{Hi}) side of R. Let c be (x:=1), then the low- and high-security meanings of c are:

$$\begin{aligned}\mathit{evL}\ c &= P_{\mathsf{Lo}}\ (\mathsf{u}_L[x \mapsto 1] \gg_{\mathsf{K}} \eta_{\mathsf{K}}(D\ ())) \qquad \text{(low)}\\ \mathit{evH}\ c &= P_{\mathsf{Hi}}\ (\mathsf{u}_H[x \mapsto 1] \gg_{\mathsf{K}} \eta_{\mathsf{K}}(D\ ())) \qquad \text{(high)}\end{aligned}$$

Note that the assignment in c is mapped by *evL* and *evH* into lifted operations on *different* monads (i.e., Lo and Hi in (low) and (high), resp.). Then, this security assignment is maintained in the resumption trace by the P_{Lo} and P_{Hi} tags. Because u_L and u_H operate on different states, these assignments can not interfere (in the sense of Section 4.2).

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

Schedule ($withoutHi\ lo$) creates a single Lo thread.

$$\begin{array}{ll} withHi\ :\ Beh \to Beh \to \mathsf{R}\,() & withoutHi\ :\ Beh \to \mathsf{R}\,() \\ withHi\ lo\ hi\ =\ weave\ (evL\ lo)\ (evH\ hi) & withoutHi\ lo\ =\ evL\ lo \end{array}$$

$$\begin{array}{l} weave\ :\ \mathsf{R}\,() \to \mathsf{R}\,() \to \mathsf{R}\,() \\ weave\ (P_{\mathsf{Lo}}\ \varphi)\ (P_{\mathsf{Hi}}\ \gamma) = P_{\mathsf{Lo}}\ (\varphi \star_{\mathsf{K}} \lambda r_{lo}.\ \eta_{\mathsf{K}}(P_{\mathsf{Hi}}\ (\gamma \star_{\mathsf{K}} \lambda r_{hi}.\eta_{\mathsf{K}}(weave\ r_{lo}\ r_{hi})))) \end{array}$$

The *run* morphism from Figure 3, which projects resumption computations in R to the runtime platform K, is defined as:

$$\begin{array}{l} run\ ::\ \mathsf{R}\,a \to \mathsf{K}\,a \\ run\ (D\ v) \quad = \eta_{\mathsf{K}} v \\ run\ (P_{\mathsf{Lo}}\ \varphi) = \varphi \star_{\mathsf{K}} run \\ run\ (P_{\mathsf{Hi}}\ \varphi) = \varphi \star_{\mathsf{K}} run \end{array}$$

5.3 Security for this setting: take separation

This section develops a non-interference style specification of domain 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]:

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

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 precise:

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

Here, $[\![-]\!]$ is defined as in Equation 1 with *evL* and *evH* from Figure 6, and *mask* is a stateful operation on the K monad which overwrites all non-Lo states. The particular definition of *mask* differs as the monadic event system

Dynamic Scheduling (3)

$$\begin{aligned}
\mathsf{R} &= \mathsf{ResT}_3\ \mathsf{K}\\
\mathsf{K} &= \mathsf{StateT}\ Sys\ (\mathsf{StateT}\ H\ (\mathsf{StateT}\ L\ \mathsf{Id}))\\
Sys &= [\mathsf{R}()] \times [\mathsf{R}()]
\end{aligned}$$

Thread-level Control (4)

$$\begin{aligned}
\mathsf{R} &= \mathsf{ResT}_3\ \mathsf{K}\\
\mathsf{K} &= \ldots\\
Sys &= [\mathsf{Re}\,Rsp] \times [\mathsf{Re}\,Rsp]\\
\mathsf{Re} &= \mathsf{ResT}_4\ \mathsf{K}\\
Req_4 &= Continue + Sleep\\
Rsp &= ()
\end{aligned}$$

Interdomain Communication (5)

$$\begin{aligned}
\mathsf{R} &= \mathsf{ResT}_3\ \mathsf{K}\\
\mathsf{K} &= \ldots\\
Sys &= [\mathsf{Re}\,Rsp] \times [\mathsf{Re}\,Rsp] \times [Int] \times [Int]\\
\mathsf{Re} &= \mathsf{ResT}_4\ \mathsf{K}\\
Req_5 &= Req_4 + Snd(Dom,Int) + Rcv(Dom,Var)\\
Rsp &= ()\\
Dom &= Hi + Lo
\end{aligned}$$

Summary of Refinements to ResT

$$\begin{aligned}
\mathsf{ResT}_0\ \mathsf{M}\,a &= \mu\mathsf{R}.\,D\,a + P\,(\mathsf{M}(\mathsf{R}\,a))\\
\mathsf{ResT}_1\ \mathsf{M}\,a &= \mu\mathsf{R}.\,D\,a + P_{\mathsf{Lo}}\,(\mathsf{M}(\mathsf{R}\,a)) + P_{\mathsf{Hi}}\,(\mathsf{M}(\mathsf{R}\,a))\\
\mathsf{ResT}_3\ \mathsf{M}\,a &= \mu\mathsf{R}.\,D\,a + P_{\mathsf{Lo}}\,(\mathsf{M}(\mathsf{R}\,a)) + P_{\mathsf{Hi}}\,(\mathsf{M}(\mathsf{R}\,a)) + P_{\mathsf{K}}\,(\mathsf{M}(\mathsf{R}\,a))\\
\mathsf{ResT}_4\ \mathsf{M}\,a &= \mu\mathsf{R}.\,Done\ a + PauseLo\,(Req \times (Rsp \to \mathsf{M}(\mathsf{R}\,a))) + PauseHi\,(Req \times (Rsp \to \mathsf{M}(\mathsf{R}\,a))) + PauseK\,(\mathsf{M}(\mathsf{R}\,a))
\end{aligned}$$

Fig. 7. The rest of the story (Points 3-5 in Figure 4). This briefly summarizes the refinements to the monad hierarchies underlying these systems. We omit point 2 (multitasking) as it requires no refinement to the monad transformers.

is refined, but here, it is merely $liftHi(\mathsf{u}_H\ (\lambda\_.h_0))$. The operator *mask* plays the rôle of $\downarrow_{\mathsf{Lo}}$ in a trace-based event system. We define an additional helper function, *takeLo*. The function $(takeLo\ n)$ picks out the initial sequences containing n Lo events:

$$\begin{aligned}
&takeLo : Int \to \mathsf{R}() \to \mathsf{R}()\\
&takeLo\ 0\ x &&= D()\\
&takeLo\ n\ (P_{\mathsf{Lo}}\ \varphi) &&= P_{\mathsf{Lo}}\ (\varphi \star_{\mathsf{K}} (\eta_{\mathsf{K}} \circ (takeLo\ (n-1))))\\
&takeLo\ n\ (P_{\mathsf{Hi}}\ \varphi) &&= P_{\mathsf{Hi}}\ (\varphi \star_{\mathsf{K}} (\eta_{\mathsf{K}} \circ (takeLo\ n)))
\end{aligned}$$

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

Theorem 5.2 (Take Separation) *Let lo and hi be any Beh programs, then for all natural numbers n,*

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

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

Proof (sketch) of Theorem 5.2 by induction on n. All binds $\star$ and units η below are in the K monad. For any natural number n, the computation:

$$run\ (takeLo\ n\ (withHi\ (evL\ lo)\ (evH\ hi)))$$

will have the form: $(l_1 \gg h_1 \gg \ldots \gg l_n \gg h_n)$ for K-computations l_i and h_i. Note that l_i and h_i are not arbitrary K-computations, but rather are lifted update and get events from the Lo and Hi domains, respectively. We know this from the definitions of the semantics *evL* and *evH*. This permits reasoning about system behaviors using atomic noninterference, the state monad axioms,

and the clobber rule in the following inductive proof.

Case $n = 0$.

$run\ (takeLo\ 0\ (withoutHi\ (evL\ lo))) \gg mask$

$= run(Done\ ()) \gg mask$ {defn takeLo}

$= run\ (takeLo\ 0\ (withHi\ (evL\ lo)\ (evH\ hi))) \gg mask$

Case $n = k + 1$.

$run\ (takeLo\ (k+1)\ (withHi\ (evL\ lo)\ (evH\ hi))) \gg mask$

$= (l_1 \gg h_1 \gg \ldots \gg l_{(k+1)} \gg h_{(k+1)} \gg \eta()) \gg mask$ {right unit & assoc.}

$= l_1 \gg h_1 \gg \ldots \gg l_{(k+1)} \gg h_{(k+1)} \gg mask$ {clobber/cancellation}

$= l_1 \gg h_1 \gg \ldots \gg l_{(k+1)} \gg mask$ $\{l_{(k+1)} \# mask\}$

$= l_1 \gg h_1 \gg \ldots \gg mask \gg l_{(k+1)}$ {ind. hyp.}

$= \underbrace{l_1 \gg \ldots}_{h_i \text{ excised}} \gg mask \gg l_{(k+1)}$ $\{l_i \# mask\}$

$= l_1 \gg \ldots \gg l_{(k+1)} \gg mask$ $\{l_{(k+1)} \# mask\}$

$= run\ (takeLo\ (k+1)\ (withoutHi\ (evL\ lo))) \gg mask$

6 The rest of the spectrum

Figure 7 summarizes the refinements to the monadic hierarchy along the development path in Figure 4—this includes points 3-5. While lack of space allows only a very high-level description of these systems, it is astonishing how easily system behavior may be enriched with only very small changes to the underlying monad hierarchy.

Point (3) extends system behavior with dynamic dispatch of threads. The *Sys* state now maintains wait lists for Hi and Lo threads. Kernel-level events (i.e., scheduling decisions) now are included in the ResT_3 transformer in the form of the P_K constructor. Point (4) extends system behavior with thread-level control flow in the form of a Unix-like "sleep" signal. Monad transformer ResT_4 is a refinement to ResT_3 which adds signals. ResT_4 provides a non-proper morphism, $usersignal : Req \to \mathsf{Re}\ Rsp$ allowing user threads to contact the kernel with requests (e.g., *Sleep*), and, due to the dynamic dispatch capability, the kernel may service such requests. Point (5) implements asynchronous send and receive by enriching the *Sys* state with Hi and Lo message buffers and the *Req* type with send and receive signals. A thread contacts the kernel with *usersignal* as before, and it is the responsibility of the kernel to ignore insecure, "write-down" requests (e.g., $usersignal(Snd(\mathsf{Lo}, msg))$) from Hi threads.

7 Conclusion

Although the verification of OSKer's security properties is not complete, the challenges of this verification have inspired the work here. Our approach is

based on the assumption that fully-automated methods—such as information-flow type systems—are not currently powerful and general enough to verify a large, complex system such as OSKer. The formidable task of verifying OSKer can be kept tractable by these techniques: (1) the system is built using a purely functional, statically-typed, polymorphic language, and thus the majority of the system components can be reasoned about using only their types; and (2) the system is structured to minimize the code that must be reasoned about explicitly.

Our system structure is based on monads, originally introduced to allow principled, effectful programming in pure, higher-order functional programming languages. Using monads, imperative features can be added to a pure language while preserving the semantics of the purely functional subset of the language. This work demonstrates that the monadic approach to effects has benefits far beyond the mere imitation of imperative style: using monads we can precisely manage and control information-flow and scope of effects. How such control might be achieved in an imperative language remains an open question.

The implementation language must be typed and purely functional to ensure that the monad abstraction is not compromised. Though we use Haskell as our implementation language, we do not use its laziness or type classes in any essential way; thus our techniques could be applied using the pure subset of ML. We have implemented systems 0-5 from Figure 4 in Haskell, and these implementations are available by request from the first author.

Monad transformers have proven their usefulness for modularizing interpreters and compilers [16,14], resulting in modular components from which systems can be created. In this work we have shown their usefulness for modularizing a very different type of system. As seen in Section 5, this modularity enables us to straightforwardly extend and refine our system to have greater functionality. Our formulation also allows for extending our notion of interference to the intransitive case [12,29], thus allowing for trusted processes (such as cryptographic servers) that are allowed to reclassify information.

A modular system not only allows for easier extension but also results in more modular verification. As we extend the system, we extend and modify the verification proofs; we need not perform verification from scratch. Although this is a work in progress, we expect our techniques to scale as our system grows in functionality. Scalable techniques for the modeling and implementation of complex systems requiring 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 purely-functional, higher-order, typed languages.

References

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

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

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

[4] Birman, K., 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, 2000.

[5] Boebert, W., W. Young, R. Kain and S. Hansohn, *Secure ada target: Issues, system design, and verification*, in: *Proc. IEEE Symposium on Security and Privacy*, 1985, pp. 176–183.

[6] Boudol, G. and I. Castellani, *Noninterference for concurrent programs*, Lecture Notes in Computer Science **2076** (2001), pp. 382+.

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

[8] Flatt, M., R. B. Findler, S. Krishnamurthi and M. Felleisen, *Programming languages as operating systems (or revenge of the son of the lisp machine)*, in: *International Conference on Functional Programming*, 1999, pp. 138–147.

[9] Goguen, J. and J. Meseguer, *Unwinding and inference control*, in: *Proc. IEEE Symposium on Security and Privacy*, 1984, pp. 75–86.

[10] Goguen, J. A. and J. Meseguer, *Security policies and security models*, in: *Proceedings of the 1982 Symposium on Security and Privacy (SSP '82)* (1990), pp. 11–20.

[11] Gray III, J., *Probabilistic interference*, in: *Proc. IEEE Symposium on Research in Security and Privacy*, 1990, pp. 170–179.

[12] Haigh, J. and W. Young, *Extending the non-interference version of MLS for SAT*, in: *Proc. IEEE Symposium on Security and Privacy*, 1986, pp. 232–239.

[13] Harper, R., 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 (1998), (Also published as Fox Memorandum CMU-CS-FOX-98-02).

[14] Harrison, W., "Modular Compilers and Their Correctness Proofs," Ph.D. thesis, University of Illinois at Urbana-Champaign (2001).

[15] Heintze, N. and J. 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* (1998), pp. 365–377.

[16] Liang, S., "Modular Monadic Semantics and Compilation," Ph.D. thesis, Yale University (1998).

[17] Liang, S., P. Hudak and M. Jones, *Monad transformers and modular interpreters*, in: *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995* (1995), pp. 333–343.

[18] McCauley, E. J. and P. J. Drongowski, *KSOS—the design of a secure operating system*, , **48**, 1979, pp. 345–353.

[19] McCullough, D., *Noninterference and the composability of security properties*, in: *Proc. IEEE Symposium on Security and Privacy*, 1988, pp. 177–187.

[20] McLean, J., *A general theory of composition for trace sets closed under selective interleaving functions*, in: *Proc. IEEE Symposium on Research in Security and Privacy*, 1994, pp. 79–93.

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

[22] Neumann, P. G., 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 (1980).

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

[24] Peyton Jones, S. and W. P., *Imperative functional programming*, in: *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*, 1993, pp. 71–84.

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

[26] Pottier, F. and V. Simonet, *Information flow inference for ML*, in: *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, Portland, Oregon, 2002, pp. 319–330.

[27] Rushby, J., *Design and verification of secure systems*, in: *Proceedings* 8^{th} *Symposium on Operating System Principles*, Pacific Grove, CA, 1981, pp. 12–21.

[28] Rushby, J., *Proof of separability: A verification technique for a class of security kernels*, in: *Proc.* 5^{th} *Int. Symp. on Programming* (1982), pp. 352–362.

[29] Rushby, J., *Noninterference, transitivity, and channel-control security policies*, Technical report, SRI (1992).

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

[31] Schmidt, D. A., "Denotational Semantics," Allyn and Bacon, Boston, 1986, xiii+331 pp.

[32] Shapiro, J. S., J. M. Smith and D. J. Farber, *EROS: a fast capability system*, in: *Symposium on Operating Systems Principles*, 1999, pp. 170–185.

[33] Smith, G., *A new type system for secure information flow*, in: *CSFW14* (2001), pp. 115–125.

[34] Smith, G. and D. 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*, ACM, New York, 1998, pp. 355–364.

[35] Volpano, D. M. and G. Smith, *A type-based approach to program security*, in: *TAPSOFT*, 1997, pp. 607–621.

[36] Wadler, P., *The Essence of Functional Programming*, in: *Proceedings of the 19th Symposium on Principles of Programming Languages* (1992), pp. 1–14.

[37] Walker, B. J., R. A. Kemmerer and G. J. Popek, *Specification and verification of the ucla unix security kernel*, Communications of the ACM **23** (1980), pp. 118–131.

[38] Wand, M., *Continuation-based multiprocessing*, in: J. Allen, editor, *Conference Record of the 1980 LISP Conference* (1980), pp. 19–28.

[39] White, P., W. Martin, A. Goldberg and F. Taylor, *Formal construction of the mathematically analyzed separation kernel*, in: *The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, 2000, pp. 133–141.

[40] Zakinthinos, A. and E. S. Lee, *A general theory of security properties*, in: *Proceedings of the 18th IEEE Computer Society Symposium on Research in Security and Privacy*, 1997.