# The Confinement Problem in the Presence of Faults*

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

[1] University of Missouri, MO 65211 USA
[2] US Naval Research Laboratory, Code 5543, Washington, DC, USA

**Abstract.** In this paper, we establish a semantic foundation for the safe execution of untrusted code. Our approach extends Moggi's computational $\lambda$-calculus in two dimensions with operations for asynchronous concurrency, shared state and software faults and with an effect type system *à* la Wadler providing fine-grained control of effects. An equational system for fault isolation is exhibited and its soundness demonstrated with a semantics based on monad transformers. Our formalization of the equational system in the Coq theorem prover is discussed. We argue that the approach may be generalized to capture other safety properties, including information flow security.

## 1 Introduction

Suppose that you possess an executable of unknown provenance and you wish to run it safely. The cost of analyzing the binary is prohibitive, and so, ultimately, you have little choice but to explore its effects by trial and error. That is, you run it and hope that nothing irreversibly damaging is done to your system. There are two alternatives proposed in the literature to the trial and error strategy. You can attempt to detect safety and security flaws in the untrusted code with automated static analyses. This is the approach being explored by much of the literature from the language-based security [27] community. The other approach is to isolate the untrusted code so that any destructive side effects (malicious or otherwise) resulting from its execution are rendered inert.

This paper introduces the *confinement calculus* (CC) and uses it as a vehicle for exploring the design and verification of isolation kernels (defined below). CC extends Moggi's computational $\lambda$-calculus [22] with constructs for state, faults and concurrency. Furthermore, the type system for the CC also incorporates an effect system *à* la Wadler [29] to distinguish computations occurring on different domains. The CC concurrency metalanguage is closely related to recent work of Goncharov and Schröder [12].

Lampson coined the term *confinement problem* [17] for the challenge of confining arbitrary programs—i.e., executing arbitrary code in a manner that prevents the illegitimate leakage of information through what Lampson termed

* This research was supported by NSF CAREER Award 00017806, US Naval Research Laboratory Contract 1302-08-015S, and by the U.S. Department of Education GAANN grant no. P200A100053.

*covert channels*. Legitimate channels transfer information via a system's resources used as intended by its architects. Covert channels transfer information by using or misusing system resources in ways unintended by the system's architects. The isolation property we define—called *domain isolation*—is similar to, albeit more restrictive than, the security property from our previous work [14]. Delimiting the scope of effects for arbitrary programs is the essence of confinement and the combination of effect types with monads is the scoping mechanism we use to confine effects.

A simple isolation kernel written in CC is presented in Figure 1. We assume there are two confinement domains, named Athens ($\mathsf{A}$) and Sparta ($\mathsf{S}$). The kernel is a function `k` which is parameterized by *domain handler functions* for each of the input domains, with types $\Delta_{\mathsf{A}}$ and $\Delta_{\mathsf{S}}$ respectively. These domain handlers are applied by the kernel to produce a single effectful computation step; the effect system guarantees that the effects of the $\Delta_{\mathsf{A}}$ (resp. $\Delta_{\mathsf{S}}$)-typed handler are restricted to $\mathsf{A}(\mathsf{S})$. The kernel also takes as input an internal kernel state value (here just a domain tag of type `D` which serves a similar function to a process id), and domain state values of types $Dom_{\mathsf{A}}$ and $Dom_{\mathsf{S}}$. Execution of the respective threads is interleaved according to a round robin policy. The `unfold` operator encapsulates guarded recursion.

The proof that `k` is, in fact, an *isolation* kernel rests on two important features of the CC. The effect system guarantees that the domain handlers do not themselves induce state effects outside of their respective domains. The equational logic allows us to prove, using simple monadic equational reasoning, that the interleaving of the threads by `k` does not introduce new interactions between the domains: failure in Athens will not propagate to Sparta (nor vice versa).

```
D = {A, S}
k : (Δ_A × Δ_S) → (D × Dom_A × Dom_S) → R^D()
k (ha,hs) (s_0,α_0,σ_0) =
 unfold (s_0,α_0,σ_0)
        (λ(s,α,σ). case s of
              A -> case α of
                       (Just x) -> ha x >>= λα'. return (Left (S,α',σ))
                       Nothing  -> return (Left (S,α,σ))
              S -> case σ of
                       (Just x) -> hs x >>= λσ'. return (Left (A,α,σ'))
                       Nothing  -> return (Left (A,α,σ)))
```

**Fig. 1:** A Simple Isolation Kernel in CC

The structure of the remainder of this article is as follows. The rest of this section introduces the safety property *fault isolation* and motivates our approach to it. Section 2 presents an overview of the literature on effect systems and monads. The Confinement Calculus is defined in Section 3. Section 4 demonstrates how to use the CC to construct and verify an isolating kernel. Formalization of the Confinement Calculus in the Coq theorem prover is discussed in Section 5. Related work is discussed in Section 6, and Section 7 concludes.

### A Monadic Analysis of Fault Isolation

Fault isolation is a safety property which prescribes boundaries on the extent of a fault effect. Here, we take a *fault* to mean a failure within a thread that causes it to terminate abnormally. The causes of a fault can be many and system-dependent, but some typical causes include activities such as division by zero, the corruption of a runtime stack, etc. Under some circumstances, one thread's failure can "crash" other threads. Fault isolation in this context means that the failure of one thread can only effect a subset of all threads running on a system.

We assume that the threads running on a system are partitioned into *domains* where the term is adapted from the terminology of hypervisors [3] and separation kernels [20] rather than denotational semantics. Fault isolation, as we use the term, means that a thread effect may only operate on its own domain. We refer to a *fault-isolating kernel* as a multitasking, multi-domain kernel in which the imperative and fault operations on one domain have no impact on other thread domains. Our fault model applies equally as well to software traps and programmable exceptions, although we do not provide the details here.

From the perspective of an individual thread, the scope of a fault should be global. Let the thread $t$ be a sequence of atoms, $a_0; a_1; a_2; \cdots$, then, if $a_0$ causes a fault, then the execution of $a_1; a_2; \cdots$ should be cancelled, thereby satisfying the (pseudo-)equation, $a_0; a_1; a_2; \cdots = a_0$. From the point of view of a concurrent system (e.g., a multitasking kernel, etc.), the scope of a fault within an individual thread must remain isolated. The execution of $t$ is really interwoven with other actions, including potentially those of other threads (e.g., $b_0; a_0; b_1; a_1; b_2; a_2; \cdots$), and a fault within $t$ must not effect the execution of the other actions. In other words, should $a_0$ cause a fault, then the following (pseudo-)equation should hold, $b_0; a_0; b_1; a_1; b_2; a_2; \cdots = b_0; a_0; b_1; b_2; \cdots$, specifying that the subsequent actions of $t$ should be filtered from the global system execution. The pseudo- prefix on the aforementioned equations signifies that the equations capture intuitions rather than rigorous mathematical truth. The confinement calculus will allow us to make these statements rigorous.

## 2 Effect Systems and Monads

Effect systems [24] and monads [22, 18] are means of representing the potential side effects of a program explicitly within its type. This section provides a brief overview of effect systems and monads and motivates our use of their combination.

**Effect Types.** Effect systems are commonly associated with impure, strongly typed functional languages (e.g., ML [21]) because the effect type annotations make explicit the side effects already present implicitly in the language itself. In an impure, strongly-typed functional language, the type of a function specifies its input and output behavior only. An ML function, $f : \texttt{int} \to \texttt{int}$, takes and returns integer values, but, because ML is impure, it may also have side effects (e.g., destructive update or programmable exceptions) which are not reflected

in its type. An effect system would indicate the potential side effects in the type itself. Annotating the arrow in $f$'s type with $\rho$ (i.e., $f : \texttt{int} \xrightarrow{\rho} \texttt{int}$) could be used to indicate that $f$ may destructively update region $\rho$. Effect annotations are introduced via side effecting language constructs (e.g., ML's assignment and dereference operations, `:=` and `!`, respectively). An effect type system tracks the effects within a program to indicate its potential side effects. For an excellent account of effect systems, the reader is referred to Nielson, et al. [24].

**Monads.** Pure, strongly-typed functional languages (e.g., Haskell [25]) do not allow side effects, so there are no implicit side effects to make explicit. Monads are used to mimic side effecting computations within a pure language. Monads in Haskell are type constructors with additional operations, bind (`>>=`) and unit (`return`), obeying the "monad laws" (defined in Figure 2). What makes monads useful is that programmers can tailor the desired effects to the application being constructed, effectively configuring a domain-specific language for each application. Rewriting it in Haskell, $f$ now has type $\texttt{Int} \rightarrow \texttt{M Int}$ where `M` is the monad type constructor that encapsulates desired effects. Monads are also algebraic constructions with properties useful to formal verification (more will be said about this below). Figure 2 presents Moggi's well-known computational $\lambda$-calculus [22]. The computational $\lambda$-calculus is the core of any equational logic for monadic specifications, including the logic presented in Section 3. An equational judgment has the form, $\Sigma \vdash \Gamma \rhd\ e_1 = e_2\ :\ t$, where $\Sigma$, $\Gamma$ and $t$ are a set of hypotheses, a typing environment, and a type, respectively.

$$\frac{\Sigma \vdash \Gamma, x : B \rhd\ e_1 = e_2\ :\ A}{\Sigma \vdash \Gamma, x : B \rhd\ \texttt{return}(e_1) = \texttt{return}(e_2)\ :\ MA} \quad (\mathsf{cong1})$$

$$\frac{\Sigma \vdash \Gamma, x : C \rhd\ e_1 = e_2\ :\ MA \quad \Sigma \vdash \Gamma, x' : A \rhd\ e_1' = e_2'\ :\ MB}{\Sigma \vdash \Gamma, x : C \rhd\ e_1 \texttt{>>=} \lambda x'.e_1' = e_2 \texttt{>>=} \lambda x'.e_2'\ :\ MB} \quad (\mathsf{cong2})$$

$$\frac{\Gamma, x : A \rhd e_1 : MA \quad \Gamma, x_1 : B \rhd e_2 : MB \quad \Gamma, x_2 : C \rhd e_3 : MC}{\Sigma \vdash \Gamma, x : A \rhd\ (e_1 \texttt{>>=} \lambda x_1.e_2) \texttt{>>=} \lambda x_2.e_3 = e_1 \texttt{>>=} \lambda x_1.(e_2 \texttt{>>=} \lambda x_2.e_3)\ :\ MC} \quad (\mathsf{assoc})$$

$$\frac{\Gamma, x : A \rhd e_1 : B \quad \Gamma, x_1 : B \rhd e_2 : MC}{\Sigma \vdash \Gamma, x : A \rhd\ (\texttt{return}(e_1) \texttt{>>=} \lambda x_1.e_2) = [e_1/x_1]e_2\ :\ MC} \quad (\mathsf{l\text{-}unit})$$

$$\frac{\Gamma, x : A \rhd e_1 : MB}{\Sigma \vdash \Gamma, x : A \rhd\ e_1 \texttt{>>=} \lambda x_1.\texttt{return}(x_1) = e_1\ :\ MB} \quad (\mathsf{r\text{-}unit})$$

**Fig. 2:** The Computational $\lambda$-Calculus. $M$ stands for any monad. The "monad laws" are assoc (associativity), l-unit (left unit), and r-unit (right unit).

**Effect Systems + Monads.** Combining effect systems with monadic semantics (as in Wadler [29]) provides fine-grained tracking of effects with a semantic model of those effects. Monads give rise to an integrated theory of effects and effect propagation. The integration of multiple effects within a single monad $M$ has consequences for formal verification. Because all of the effects are typed in $M$, those effects are not distinguished syntactically within the type system of a

specification language. More positively, a rich equational theory governing their interaction follows by construction.

Effect systems can reflect this semantic information in the syntax of the specification language itself, thereby making monadic specifications more amenable to logical analysis. In the setting of this research, the combination of effects systems with monads is used to abstract over computations that occur on a particular domain. Given a particular domain $d$ and monad $K$, for example, any term, $\Gamma \triangleright e : K^{\{d\}} A$, is arbitrary code on domain $d$, i.e., its effects occur only in domain $d$. Combining effects systems and monads delimits the scope of effects for arbitrary programs and is the principal mechanism for designing and verifying confinement systems.

**The Identity and State Monads.** The identity (left) and state (right) monads are defined below (where *Sto* can be any type). The *return* operator is the monadic analogue of the identity function, injecting a value into the monad. The $\gg\!=$ operator is a form of sequential application. Monadic operators other than $\gg\!=$ and *return* are key to the formulation of a particular notion of computation. The state monad $S$ encapsulates an imperative notion of computation with operators for updating and reading the state, $u$ and $g$, resp.

$$
\begin{array}{ll}
\textbf{data}\ Id\ a \quad = Id\ a & g\ :\ S\ Sto \\
return\ v \quad = Id\ v & g\ =\ S\ (\lambda\sigma.(\sigma,\sigma)) \\
(Id\ x)\ \gg\!= f = f\ x & return\ v\ =\ S\ (\lambda\sigma.\ (v,\sigma)) \\
\textbf{data}\ Sa = S(Sto \to (a, Sto)) & (S\ x)\ \gg\!= f \\
deS\ (S\ x) \quad = x & \quad =\ S\ (\lambda\sigma_0.\ \textbf{let}\ (v,\sigma_1)\ =\ x\ \sigma_0 \\
u : (Sto \to Sto) \to S() & \qquad\qquad \textbf{in}\ deS\ (f\ v)\ \sigma_1) \\
u\ f \quad = S\ (\lambda\sigma.((), f\ \sigma)) &
\end{array}
$$

**The Maybe Monad & Errors.** The usual formulation of an error monad is called *Maybe* in Haskell (see below). An error (i.e., *Nothing*) has the effect of canceling the rest of the computation (i.e., `f`). The scope of *Nothing* is global in the sense that each of the following expressions evaluates to *Nothing*: $(Just\ 1\ \gg\!=\ \lambda v.\ Nothing)$, $(Nothing\ \gg\!=\ \lambda d.\ Just\ 1)$, $(Just\ 1\ \gg\!=\ \lambda v.\ Nothing\ \gg\!=\ \lambda d.\ Just\ 2)$.

$$
\begin{array}{ll}
\textbf{data}\ Maybe\ a = Just\ a \mid Nothing & Just\ v\ \gg\!= f \quad = f\ v \\
return \quad = Just & Nothing\ \gg\!= f = Nothing
\end{array}
$$

Observe that this behavior precludes the possibility of *fault isolation* within domains: if the *Nothing* occurs on one domain and the $(Just\ 1)$ or $(Just\ 2)$ occur on another, the entire multi-domain computation will be canceled. From a security point of view, this is clearly undesirable: allowing a high-security computation to terminate a low-security computation introduces information flow via a termination channel, and allowing a low-security computation to terminate a high-security computation exposes the system to a denial of service attack.

**Monad transformers.** Monad transformers allow monads to be combined and extended. Monad transformers corresponding to the state monad are defined in

Haskell below. Formulations of the state monad equivalent to those above are produced by the applications of this transformer to the identity monad, *StateT Sto Id*. In the following, type variable $m$ abstracts over monads.

$$
\begin{array}{l}
\textbf{data}\ \mathit{StateT}\ s\ m\ a\ =\ \mathit{ST}\ (s\ \rightarrow\ m\ (a, s)) \\
\mathit{deST}\ (\mathit{ST}\ x)\ =\ x \\
\mathit{return}\ v\ =\ \mathit{ST}\ (\lambda s.\ \mathit{return}_m\ (v, s)) \\
(\mathit{ST}\ x)\ \gg\!=\ f\ =\ \mathit{ST}\ (\lambda s_0.\ (x\ s_0)\ \gg\!=_m\ \lambda(y, s1).\ \mathit{deST}\ (f\ y)\ s1) \\
\mathit{lift}\ :\ m\ a\ \rightarrow\ \mathit{StateT}\ s\ m\ a \\
\mathit{lift}\ \varphi\ =\ \mathit{ST}\ (\lambda s.\ \varphi\ \gg\!=_m\ \lambda v.\ \mathit{return}_m\ (v, s))
\end{array}
$$

For a monad $m$ and type $s$, $(\mathit{StateT}\ s\ m)$ "extends" $m$ with an updateable store $s$. The *lift* morphism is used to redefine any existing operations on $m$ for the monad $(\mathit{StateT}\ s\ m)$. The process of lifting operations is analogous to inheritance in object-oriented languages. The layer $(\mathit{StateT}\ s\ m)$ also generalizes the definitions of the update and get operators:

$$
\begin{array}{ll}
u\ :\ (s\ \rightarrow\ s)\ \rightarrow\ \mathit{StateT}\ s\ m\ () & g\ :\ \mathit{StateT}\ s\ m\ s \\
u\ f\ =\ \mathit{ST}\ (\lambda s.\ \mathit{return}_m\ ((), f\ s)) & g\ =\ \mathit{ST}\ (\lambda s.\ \mathit{return}_m\ (s, s))
\end{array}
$$

**Layered State Monads.** A layered state monad is a monad constructed from multiple applications of the state monad transformer to an existing monad.

$$
\begin{array}{l}
\mathit{type}\ K\ =\ \mathit{StateT}\ \mathit{Sto}\ (\mathit{StateT}\ \mathit{Sto}\ (\mathit{StateT}\ \mathit{Sto}\ \mathit{Id})) \\
u_1,\ u_2,\ u_3\ :\ (\mathit{Sto}\ \rightarrow\ \mathit{Sto})\ \rightarrow\ K\ () \\
u_1\ f\ =\ u\ f \\
u_2\ f\ =\ \mathit{lift}\ (u\ f) \\
u_3\ f\ =\ \mathit{lift}\ (\mathit{lift}\ (u\ f))
\end{array}
$$

Each application of $(\mathit{StateT}\ \mathit{Sto})$ creates a layer with its own instances of the update ($u_1$-$u_3$) and get operations (not shown). These imperative operators come with useful properties by construction [14] and some of these are included as the equational rules (clobber) and (atomic n.i.) (atomic non-interference) in Section 3.

**Resumption-monadic Concurrency.** Two varieties of resumption monad are utilized here, the *basic* and *reactive* resumption monads [13]. Basic resumptions encapsulate a concurrency-as-interleaving notion of computation, while reactive resumptions refine this notion to include a failure signal. The basic and reactive monad transformers are defined in Haskell in terms of monad $m$ as:

$$
\begin{array}{l}
\textbf{data}\ \mathit{ResT}\ m\ a\ \ =\ \mathit{Done}\ a\ \mid\ \mathit{Pause}\ (m\ (\mathit{ResT}\ m\ a)) \\
\mathit{return}\ \ \ \ \ \ \ \ \ \ \ \ =\ \mathit{Done} \\
(\mathit{Done}\ v)\ \gg\!=\ f\ \ =\ f\ v \\
(\mathit{Pause}\ \varphi)\ \gg\!=\ f\ =\ \mathit{Pause}\ (\varphi\ \gg\!=_m\ \lambda\kappa.\ \mathit{return}_m\ (\kappa\ \gg\!=\ f))
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathit{ReactT}\ m\ a\ =\ \mathit{Dn}\ a\ \mid\ \mathit{Ps}\ (m\ (\mathit{ReactT}\ m\ a))\ \mid\ \mathit{Fail} \\
\mathit{return}\ \ \ \ \ \ \ \ \ \ \ \ \ =\ \mathit{Dn} \\
(\mathit{Dn}\ v)\ \gg\!=\ f\ \ \ \ \ =\ f\ v \\
(\mathit{Ps}\ \varphi)\ \gg\!=\ f\ \ \ \ \ =\ \mathit{Ps}\ (\varphi\ \gg\!=_m\ \lambda\kappa.\ \mathit{return}_m\ (\kappa\ \gg\!=\ f)) \\
\mathit{Fail}\ \gg\!=\ f\ \ \ \ \ \ \ =\ \mathit{Fail}
\end{array}
$$

We chose to formulate the reactive resumption transformer along the lines of Swierstra and Altenkirch [28] rather than that of our previous work [13] because it is simpler. We define the following monads: $Re = ReactTK$, $R = ResTK$, and $K = StateT^n\ Sto\ Id$ where $n$ is the the number of domains and $Sto$ is the type of stores (left unspecified).

Figure 3 presents the concurrency and co-recursion operations for $R$ and $Re$. The *step* operation lifts an $m$-computation into the $R$ (resp. $Re$) monad, thereby creating an atomic (w.r.t. $R$ ($Re$)) computation. A resumption computation may be viewed as a (possibly infinite) sequence of such steps; a finite $R$-computation will have the form, $(step_R\ m_1) \gg\!=_R \lambda v_1. \cdots \gg\!=_R \lambda v_n.\ (step_R\ m_n)$. The definition of $step_R$ is below ($step_{Re}$ is analogous). The $unfold_R$ operator is used to define kernels while the $unfold_{Re}$ operator is used to define threads. The co-recursion provided by these operators is the only form of co-recursion supported by the confinement calculus. An important consequence of this limitation on recursion is that it guarantees productivity [8]. It should be noted that the presence of *Maybe* in the type of $unfold_{Re}$ means that threads may fail, while its absence from the type of $unfold_R$ means that kernels cannot fail. The unfold operators are defined below; note that *Either a b* is simply Haskell-inspired notation for the sum type $a + b$.

```
step_R : K A → R A
step_R φ = Pause (φ >>= (return ∘ Done))

unfold_R : (Monad t) ⇒ a → (a → t (Either a b)) → ResT t b
unfold_R a f = step_R (f a) >>= λκ.
                 case κ of
                   (Left a')  → unfold_R a' f
                   (Right b)  → return b
unfold_Re : (Monad t) ⇒ a → (a → t (Maybe (Either a b))) → ReactT t b
unfold_Re a f = step_Re (f a) >>= λκ.
                  case κ of
                    (Just (Left a'))  → unfold_Re a' f
                    (Just (Right b))  → return b
                    Nothing           → Fail
```

**Fig. 3:** Monadic Concurrency and Co-recursion Operations

## 3 The Confinement Calculus

This section introduces the confinement calculus and defines its syntax, type system and semantics. The CC proceeds from Moggi's computational $\lambda$-calculus [22] and Wadler's marriage of type systems for effects with monads [29].

Types in the CC are directly reflective of semantic domains introduced in the previous section, and as a result are named similarly. As a notational convention,

$$
\begin{array}{lll}
e, e' \in \mathit{Exp} & ::= & x \mid \lambda x.e \mid e\, e' \mid \texttt{return}\ e \mid e\ \texttt{>>=}\ \lambda x.e' \mid \texttt{get} \mid \texttt{upd}\ e \\
& & \mid \texttt{fail} \mid \texttt{mask} \mid \texttt{out} \mid \texttt{step} \mid \texttt{unfold} \mid \texttt{zero} \mid \texttt{succ} \mid \texttt{natRec} \\
A, B \in \mathit{Type} & ::= & A \to B \mid \texttt{K}^{\sigma}\ A \mid \texttt{R}^{\sigma}\ A \mid \texttt{Re}^{\sigma}\ A \mid \texttt{Sto} \mid \texttt{Nat} \mid () \mid A + B \mid A \times B
\end{array}
$$

**Fig. 4:** *Abstract Syntax.* Assume $D = \{d_1, \ldots, d_n\}$ and $\sigma \in \mathcal{P}(D)$.

we will use teletype font when expressing a type in CC (e.g. `K Nat`), and an italic font when referring to semantic domains (e.g. $K\ Nat$).

**Abstract Syntax.** Figure 4 presents the abstract syntax for the CC. The finite set of domains, $D$ contains labels for all thread domains in the system ($D$ replaces *Region* from Wadler's original presentation of MONAD [29]). We also diverge slightly from Wadler's language in that we do not track the "sort" of effects that a computation may cause: reading, writing, and failure are all treated the same.

The monadic expression language *Exp* has familiar computational $\lambda$-calculus constructs as well as imperative operations (`get, upd`), an imperative operation (`mask`) used in specifying the isolation of imperative effects [14], and others for resumption monadic computations (`out`, `step`, and `unfold`). Intuitively, the `mask` operation has the effect of resetting or "zeroing out" a particular domain. Expressions `unfold`, `step` and `out` are resumption-monadic operations. The `unfold` operator encapsulates corecursion, and its semantics are structured to allow only guarded recursion. More will be said about `fail`, `step` and `out` later in this section. Finally, the expressions `zero`, `succ`, and `natRec` allow construction of, and primitive recursion over, natural numbers. Note that the *only* forms of recursion permitted by CC are primitive recursion over naturals (via `natRec`), and guarded corecursion over resumptions (via `unfold`).

$$
\frac{\Gamma \rhd e : A}{\Gamma \rhd \texttt{return}\ e : \texttt{K}^{\emptyset} A}
\qquad
\frac{\Gamma \rhd e : \texttt{K}^{\sigma} A \quad \Gamma, x : A \rhd e' : \texttt{K}^{\sigma'} B}{\Gamma \rhd e\ \texttt{>>=}\ \lambda x.e' : \texttt{K}^{\sigma \cup \sigma'} B}
\qquad
\frac{\Gamma \vdash e : \texttt{K}^{\sigma} A \quad \sigma \subseteq \sigma'}{\Gamma \vdash e : \texttt{K}^{\sigma'} A}
$$

$$
\frac{}{\Gamma \vdash \texttt{get} : \texttt{K}^{\{d\}} \texttt{Sto}}
\qquad
\frac{\Gamma \vdash f : \texttt{Sto} \to \texttt{Sto}}{\Gamma \vdash \texttt{upd}\ f : \texttt{K}^{\{d\}} ()}
\qquad
\frac{}{\Gamma \rhd \texttt{mask} : \texttt{K}^{\{d\}} ()}
$$

**Fig. 5:** Type System for Imperative Effects

The type syntax in Figure 4 contains three monads. The monads `K`, `R` and `Re` encapsulate layered state, concurrency and system executions, and concurrent threads, respectively. The effect system can express fine-grained distinctions about computations and, in particular, allows the domain of a thread to be expressed in its type.

**Types and Effects for the Confinement Calculus..** The type and effect system for the CC is presented in Figures 5-7 and that figure is divided into three sections. Figure 5 contains the system for the imperative core of CC—i.e., the computations in the monad `K`. Figure 6 contains the type system for concurrency. Figure 7 contains the type system for threads—i.e., computations

$$\frac{\Gamma \rhd e : A}{\Gamma \rhd \texttt{return}\ e : \texttt{R}^{\emptyset} A} \qquad \frac{\Gamma \rhd e : \texttt{R}^{\sigma} A \quad \Gamma, x : A \rhd e' : \texttt{R}^{\sigma'} B}{\Gamma \rhd e\ \texttt{>>=}\ \lambda x.e' : \texttt{R}^{\sigma \cup \sigma'} B} \qquad \frac{\Gamma \rhd e : \texttt{K}^{\sigma} A}{\Gamma \rhd \texttt{step}\ e : \texttt{R}^{\sigma} A}$$

$$\frac{\Gamma \rhd p : \texttt{R}^{\sigma} A}{\Gamma \rhd \texttt{out}\ p : \texttt{K}^{\sigma}(\texttt{R}^{\sigma} A)} \qquad \frac{\Gamma \rhd p : A \quad \Gamma, x{:}A \rhd q : \texttt{K}^{\sigma}(A + B)}{\Gamma \rhd \texttt{unfold}\ p\ (\lambda x.q) : \texttt{R}^{\sigma} B} \qquad \frac{}{\Gamma \rhd \texttt{natRec} : A{\to}(A{\to}A){\to}Nat{\to}A}$$

**Fig. 6:** Type System for Concurrency. The rule for `natRec` is also included.

in the `Re` monad. The `Re` monad expresses the same notion of computation as the `R` monad, except that it also contains a signal `fail`. A thread may use `fail` to generate a fault, and it is up to the kernel component to limit the extent of the fault to the thread's domain.

$$\frac{\Gamma \rhd e : A}{\Gamma \rhd \texttt{return}\ e : \texttt{Re}^{\emptyset} A} \qquad \frac{\Gamma \rhd e : \texttt{Re}^{\sigma} A \quad \Gamma, x : A \rhd e' : \texttt{Re}^{\sigma'} B}{\Gamma \rhd e\ \texttt{>>=}\ \lambda x.e' : \texttt{Re}^{\sigma \cup \sigma'} B} \qquad \frac{\Gamma \rhd e : \texttt{K}^{\sigma} A}{\Gamma \rhd \texttt{step}\ e : \texttt{Re}^{\sigma} A}$$

$$\frac{\Gamma \rhd p : \texttt{Re}^{\sigma} A}{\Gamma \rhd \texttt{out}\ p : \texttt{K}^{\sigma}(\texttt{Re}^{\sigma} A)} \qquad \frac{\Gamma \rhd p : A \quad \Gamma, x{:}A \rhd q : \texttt{K}^{\sigma}(A + B + ())}{\Gamma \rhd \texttt{unfold}\ p\ (\lambda x.q) : \texttt{Re}^{\sigma} B} \qquad \frac{}{\Gamma \rhd \texttt{fail} : \texttt{Re}^{\{d\}} A}$$

$$\frac{A <: B \qquad \sigma_0 \subseteq \sigma_1}{\texttt{K}^{\sigma_0} A <: \texttt{K}^{\sigma_1} B} \qquad \frac{A <: B \qquad \sigma_0 \subseteq \sigma_1}{\texttt{R}^{\sigma_0} A <: \texttt{R}^{\sigma_1} B} \qquad \frac{A <: B \qquad \sigma_0 \subseteq \sigma_1}{\texttt{Re}^{\sigma_0} A <: \texttt{Re}^{\sigma_1} B}$$

**Fig. 7:** Type System for Reactive Concurrency; Subtyping Relation

Figure 7 gives the rules for subtyping monadic computations (standard rules for reflexivity, transitivity, and for arrow, product, and sum types are omitted). The intuition is that one monadic computation $\varphi$ may stand in for another computation $\gamma$ without breaking type safety, if and only if the result type of $\varphi$ is a subtype of that of $\gamma$, *and* $\varphi$'s affected domains are a subset of $\gamma$'s. An effect-free computation of type $\texttt{K}^{\emptyset} A$ also has type $\texttt{K}^{\{d\}} A$ (but not vice versa).

**Denotational Semantics of the Confinement Calculus.** The dynamic semantics of CC is a typed denotational semantics, meaning that the denotation of terms depends in part on their typing derivations. This allows us to overload the monad operations. The denotation of types does not depend on effect annotations and is completely standard. The `out` operation accesses the first step of a concurrent (`R`-typed) computation, producing a `K`-typed computation. Omitted from Figure 8 is the denotation of `natRec`, which is defined by structural induction on naturals. The CC term `run` and its denotation are defined as:

$$\texttt{run n}\ \varphi_0\ \texttt{= natRec (return}_K\ \varphi_0\texttt{) (}\lambda\varphi\texttt{. (}\varphi\ \texttt{>>= out}_R\texttt{))\ n}$$

$$\begin{aligned} &\mathit{run} : \mathit{Nat} \to \mathit{RA} \to \mathit{K}(\mathit{RA}) \\ &\mathit{run}\ 0\ \varphi \quad\ = \mathit{return}\ \varphi \\ &\mathit{run}\ (n+1)\ \varphi = \mathit{out}\ \varphi\ \gg\!= \mathit{run}\ n \end{aligned}$$

$$
\begin{array}{ll}
[\![\Gamma \triangleright x : A]\!]\rho & = \rho\ x \\
[\![\Gamma \triangleright \lambda x.e : A \to B]\!]\rho & = \lambda v.[\![\Gamma, x : A \triangleright e : B]\!](\rho[x \mapsto v]) \\
[\![\Gamma \triangleright e\ e' : A]\!]\rho & = ([\![\Gamma \triangleright e : B \to A]\!]\rho)\ ([\![\Gamma \triangleright e' : B]\!]\rho) \\
[\![\Gamma \triangleright \texttt{return}\ e : M^{\emptyset} A]\!]\rho & = return_M\ ([\![\Gamma \triangleright e : A]\!]\rho) \\
[\![\Gamma \triangleright \texttt{get} : \mathtt{K}^{d_i}\ Sto]\!]\rho & = lift_i\ g \\
[\![\Gamma \triangleright \texttt{upd}\ \delta : \mathtt{K}^{d_i}\ ()]\!]\rho & = lift_i\ (u\ ([\![\Gamma \triangleright \delta : Sto \to Sto]\!]\rho)) \\
[\![\Gamma \triangleright \texttt{mask} : \mathtt{K}^{d_i}\ ()]\!]\rho & = lift_i\ (u\ (\lambda x.s_0)) \\
[\![\Gamma \triangleright e\ \texttt{>>=}\ \lambda x.e' : M^{\sigma \cup \sigma'} B]\!]\rho & \\
\quad = [\![\Gamma \triangleright e : M^{\sigma} A]\!]\rho \texttt{>>=}_M \lambda v.[\![\Gamma, x{:}A \triangleright e' : M^{\sigma'} B]\!](\rho[x \mapsto v]) & \\
[\![\Gamma \triangleright \texttt{unfold}\ e\ (\lambda x.e') : \mathtt{R}^{\sigma} B]\!]\rho & \\
\quad = unfold_{\mathtt{R}}\ ([\![\Gamma \triangleright e : A]\!]\rho)\ (\lambda v.[\![\Gamma, x : A \triangleright e' : \mathtt{K}^{\sigma}(A + B)]\!]\ (\rho[x \mapsto v])) & \\
[\![\Gamma \triangleright \texttt{out}(p) : \mathtt{K}^{\sigma}(\mathtt{R}^{\sigma} A)]\!]\rho & = \begin{cases} return\,(Done\ v) & \text{if } [\![p]\!]\rho = Done\ v \\ \varphi & \text{if } [\![p]\!]\rho = Pause\ \varphi \end{cases} \\
[\![\Gamma \triangleright \texttt{fail} : \mathtt{Re}^{d_i}\ A]\!]\rho & = Fail \\
[\![\Gamma \triangleright \texttt{unfold}\ e\ (\lambda x.e') : \mathtt{Re}^{\sigma} B]\!]\rho & \\
\quad = unfold_{Re}\ ([\![\Gamma \triangleright e : A]\!]\rho)(\lambda v.[\![\Gamma, x : A \triangleright e' : \mathtt{K}^{\sigma}(A + B + ())]\!]\ (\rho[x \mapsto v])) & \\
[\![\Gamma \triangleright \texttt{out}(p) : \mathtt{K}^{\sigma}(\mathtt{Re}^{\sigma} A)]\!]\rho = & \begin{cases} return\,(Just(Dn\ v)) & \text{if } [\![p]\!]\rho = Dn\ v \\ \varphi \gg\!= (return \circ Just) & \text{if } [\![p]\!]\rho = Ps\ \varphi \\ return\,Nothing & \text{if } [\![p]\!]\rho = Fail \end{cases}
\end{array}
$$

**Fig. 8:** Denotational Semantics. $M$ stands for the $K$, $R$, or $Re$ monads. $K$, $R$, and $Re$ are defined in Section 2.

**Equational Logic.** The rules of the equational logic encode known facts about the denotational semantics proven in an earlier publication [14]. For instance, the *run* operator "unrolls" $R$ computations:

$$run\ (n+1)\ (step\ \varphi \gg\!=_R f) = \varphi \gg\!=_K run\ n \circ f \tag{1}$$

$$run\ (n+1)\ (return_R\ x) = return_K\ (return_R\ x) \tag{2}$$

Properties (1) and (2) justify our introduction of the following rules:

$$\frac{\Gamma \triangleright n : Nat \quad \Gamma \triangleright \varphi : \mathtt{K}^{\sigma} A \quad \Gamma, x : A \triangleright e : \mathtt{R}^{\sigma} B}{\Sigma \vdash \Gamma \triangleright\ \texttt{run}\ (n{+}1)\ (\texttt{step}\,\varphi\ \texttt{>>=}\ \lambda x.e) = \varphi\ \texttt{>>=}\ \lambda x.\,\texttt{run}\ n\ e\ :\ \mathtt{K}^{\sigma}(\mathtt{R}^{\sigma} B)} \quad \text{(run-step)}$$

$$\frac{\Gamma \triangleright n : Nat \quad \Gamma \triangleright e : A}{\Sigma \vdash \Gamma \triangleright\ \texttt{run}\ (n{+}1)\ (\texttt{return}_R\, e) = \texttt{return}_K\,(\texttt{return}_R\, e)\ :\ \mathtt{K}^{\sigma}(\mathtt{R}^{\sigma}\ A)} \quad \text{(run-return)}$$

A straightforward induction on the structure of type derivations justifies the soundness of the following rules. This induction makes use of previous work (specifically Theorems 1-3 on page 17 [14]) and the "lifting law" of Liang [18]: $lift(x \gg\!= f) = liftx \gg\!= lift \circ f$.

$$\frac{\Gamma \triangleright \varphi : \mathtt{K}^{\sigma_0} A \qquad \Gamma \triangleright \texttt{mask} : \mathtt{K}^{\sigma_1}() \qquad \sigma_0 \subseteq \sigma_1}{\Sigma \vdash \Gamma \triangleright\ \varphi\ \texttt{>>}\ \texttt{mask} = \texttt{mask}\ :\ \mathtt{K}^{\sigma_1}()} \quad \text{(clobber)}$$

$$\frac{\Gamma \triangleright \varphi : \mathtt{K}^{\sigma_0}() \qquad \Gamma \triangleright \gamma : \mathtt{K}^{\sigma_1}() \qquad \sigma_0 \cap \sigma_1 = \emptyset}{\Sigma \vdash \Gamma \triangleright\ \varphi\ \texttt{>>}\ \gamma = \gamma\ \texttt{>>}\ \varphi\ :\ \mathtt{K}^{\sigma_0 \cup \sigma_1}()} \quad \text{(atomic n.i.)}$$

## 4 Isolation Kernels in Confinement Calculus

In this section, we turn our attention the construction of *isolation kernels* within the confinement calculus. An isolation kernel is a function which interleaves the execution of two or more threads in different domains, without introducing any interactions across domains. Put another way, an isolation kernel must have the property that a computation in domain $d$, when interleaved with a computation in $d' \neq d$, behaves exactly the same as it would if the $d'$ computation had never happened.

The formal definition of isolation is made in terms of a notion called *domain similarity*. Two computations $\varphi$ and $\gamma$ are domain similar in a domain $d$ if and only if for every finite prefix of $\varphi$, there exists a finite prefix of $\gamma$ whose effects in $d$ are the same.

**Definition 1 (Domain Similarity Relation).** *Consider two computations* $\varphi, \gamma : \mathtt{R}^{d_1 \cup \ldots \cup d_n}\, A$. *We say* $\varphi$ *and* $\gamma$ *are* similar with respect to domain $d_i$ (*written* $\varphi \sim_{d_i} \gamma$ ) *if and only if the following holds.*

$$\forall\, n{\in}N.\, \exists\, m{\in}N.\, \mathtt{run}\, n\, \varphi\, \mathtt{>>}_K\, \mathtt{mask} \;=\; \mathtt{run}\, m\, \gamma\, \mathtt{>>}_K\, \mathtt{mask}$$

*where* $\mathtt{mask} = \mathtt{mask}_{d_1}\mathtt{>>} \ldots \mathtt{>>mask}_{d_{i-1}}\mathtt{>>mask}_{d_{i+1}}\mathtt{>>} \ldots \mathtt{>>mask}_{d_n}$.

**Kernels.** Assume in Definitions 2-4 that $D = \{d_1, \ldots, d_n\}$ is a fixed set of domains. We first define a notion of *state*: namely a tuple containing one element representing the kernel's internal state, and an additional element for the state of each confinement domain. The domain states are wrapped in a *Maybe* constructor to represent the possibility of failure within a domain.

**Definition 2 (Domain and Kernel State).** *The state of domain* $i$, $Dom_i$ *is defined as:*

$$Dom_i = \; \mathtt{Re}^{\{d_i\}}()$$

*The type of kernel states for a type* $t$ *is:*

$$S = t{\times}(\mathtt{Maybe}\ Dom_1){\times} \ldots {\times}(\mathtt{Maybe}\ Dom_n)$$

A kernel is parameterized by *handlers* for each domain. A handler is a state transition function on domain states, which may also have effects on the global state (hence the presence of $K$ in the type). The only restriction on a handler for domain $d_i$ is that its effects must be restricted to $d_i$.

**Definition 3 (Domain Handler Function).** *The type of handler functions for domain* $d_i$ *is:*

$$\Delta_{d_i} = Dom_i \to \mathtt{K}^{\{d_i\}}(\mathtt{Maybe}\ Dom_i)$$

*The handler vector type for* $D$ *is:*

$$\Delta_D = \Delta_1{\times} \ldots {\times}\Delta_n$$

Putting the pieces together brings us to the definition of a kernel. Note that a kernel in our sense is parameterized over handler vectors.

**Definition 4 (Kernel).** *Given a handler vector $\Delta_D$, kernel state type $S$, and an answer type $Ans$, the type of kernels is defined by the following:*

$$\Delta_D \to S \to \mathtt{K}^D(S + Ans)$$

**Defining and Proving Isolation.** To simplify the presentation, we will restrict our attention for the remainder of this section to the case of two domains, called $\mathsf{A}$ (for *Athens*) and $\mathsf{S}$ (for *Sparta*). All the results here generalize naturally to more than two domains.

We define isolation by an extensional property on kernels. A kernel is said to be isolating if the result of "eliminating" the computation in any one domain has no effect on the outcome of any other. This is a property akin to noninterference [11]. We express this formally by replacing the domain handler with `return` – the "do-nothing" computation – and replacing the domain state with `Nothing`.

**Definition 5 (Isolation in the Presence of Faults).** *A kernel $k$ is* isolating in the presence of faults *if and only if*

$$k\ (f_\mathsf{A}, f_\mathsf{S})(s, d_\mathsf{A}, d_\mathsf{S}) \sim_\mathsf{A} k\ (f_\mathsf{A}, \mathtt{return})(s, d_\mathsf{A}, \mathtt{Nothing})$$
$$k\ (f_\mathsf{A}, f_\mathsf{S})(s, d_\mathsf{A}, d_\mathsf{S}) \sim_\mathsf{S} k\ (\mathtt{return}, f_\mathsf{S})(s, \mathtt{Nothing}, d_\mathsf{S})$$

The following theorem shows that kernel `k` of Figure 1 satisfies this definition.

**Theorem 1 (`k` is isolating).** *The kernel `k` of Figure 1 is isolating in the presence of faults.*

*Proof.* We will show the $\mathsf{S}$-similarity side of the proof, since the $\mathsf{A}$-similarity proof is analogous. N.b., Definition 1 allows the number of steps on each side of the equation ($n$ and $m$) to be different. Here it suffices to fix $m = n$:

$$\mathtt{run}\ n\ (\mathtt{k}\ (f_\mathsf{A}, f_\mathsf{S})\ (s, d_\mathsf{A}, d_\mathsf{S}))\ \mathtt{>>}\ \mathtt{mask}_\mathsf{A}$$
$$= \mathtt{run}\ n\ \ (\mathtt{k}\ \ (\mathtt{return}, f_\mathsf{S})\ \ (s, \mathtt{Nothing}, d_\mathsf{S}))\ \mathtt{>>}\ \mathtt{mask}_\mathsf{A}$$

The proof is by induction on $n$. The base case is trivial. We proceed by cases on $s$, $d_\mathsf{A}$, and $d_\mathsf{S}$. The most interesting case is when $s = \mathsf{A}$ and $d_\mathsf{A} = \mathtt{Just}\ x$; all others involve no more than straightforward evaluation and an application of the induction hypothesis. Let $s = \mathsf{A}$ and $d_\mathsf{A} = \mathtt{Just}\ x$. Then:

$$\begin{aligned}
&\mathtt{run}(n+1)(\mathtt{k}(f_\mathsf{A}, f_\mathsf{S})(\mathsf{A}, \mathtt{Just}\ x, d_\mathsf{S}))\mathtt{>>}\mathtt{mask}_\mathsf{A} \\
&= \mathtt{run}\,1\ (\mathtt{k}\ (f_\mathsf{A}, f_\mathsf{S})\ (\mathsf{A}, \mathtt{Just}\ \ x, d_\mathsf{S}))\ \mathtt{>>=}\ \mathtt{run}\,n\ \mathtt{>>}\ \mathtt{mask}_\mathsf{A} && \{\text{prop run}\} \\
&= f_\mathsf{A}\ x\ \mathtt{>>=}\ \lambda d'_\mathsf{A}.\ \mathtt{run}\,n\ (\mathtt{k}\ (f_\mathsf{A}, f_\mathsf{S})\ (\mathsf{S}, \mathtt{Just}\ \ x, d_\mathsf{S}))\ \mathtt{>>}\ \mathtt{mask}_\mathsf{A} && \{\text{evaluation}\} \\
&= f_\mathsf{A}\ x\ \mathtt{>>=}\ \lambda d'_\mathsf{A}.\ \mathtt{run}\,n\ (\mathtt{k}\ (\mathtt{return}, f_\mathsf{S})\ (\mathsf{S}, \mathtt{Nothing}, d_\mathsf{S}))\ \mathtt{>>}\ \mathtt{mask}_\mathsf{A} && \{\text{i.h.}\} \\
&= f_\mathsf{A}\ x\ \mathtt{>>=}\ \lambda d'_\mathsf{A}.\ \mathtt{mask}_\mathsf{A}\ \mathtt{>>}\ \mathtt{run}\,n\ (\mathtt{k}\ (\mathtt{return}, f_\mathsf{S})\ (\mathsf{S}, \mathtt{Nothing}, d_\mathsf{S})) && \{\text{atomic n.i.}\} \\
&= f_\mathsf{A}\ x\ \mathtt{>>}\ \mathtt{mask}_\mathsf{A}\ \mathtt{>>}\ \mathtt{run}\,n\ (\mathtt{k}\ (\mathtt{return}, f_\mathsf{S})\ (\mathsf{S}, \mathtt{Nothing}, d_\mathsf{S})) && \{d'_\mathsf{A} \text{ does not occur}\} \\
&= \mathtt{mask}_\mathsf{A}\ \mathtt{>>}\ \mathtt{run}\,n\ (\mathtt{k}\ (\mathtt{return}, f_\mathsf{S})\ (\mathsf{S}, \mathtt{Nothing}, d_\mathsf{S})) && \{\text{clobber}\}
\end{aligned}$$

$$
\begin{array}{ll}
= \texttt{run}\, n\, (\texttt{k}\, (\texttt{return}, f_{\mathsf{S}})\, (\mathsf{S}, \texttt{Nothing}, d_{\mathsf{S}}))\, \texttt{>>}\, \texttt{mask}_{\mathsf{A}} & \{\text{atomic n.i.}\} \\
= \texttt{return}\, ()\, \texttt{>>}\, \texttt{run}\, n\, (\texttt{k}\, (\texttt{return}, f_{\mathsf{S}})\, (\mathsf{S}, \texttt{Nothing}, d_{\mathsf{S}}))\, \texttt{>>}\, \texttt{mask}_{\mathsf{A}} & \{\text{left unit}\} \\
= \texttt{run}\, 1\, (\texttt{k}\, (\texttt{return}, f_{\mathsf{S}})\, (\mathsf{A}, \texttt{Nothing}, d_{\mathsf{S}}))\, \texttt{>>=}\, \texttt{run}\, n\, \texttt{>>}\, \texttt{mask}_{\mathsf{A}} & \{\text{evaluation}\} \\
= \texttt{run}\, (n+1)\, (\texttt{k}\, (\texttt{return}, f_{\mathsf{S}})\, (\mathsf{A}, \texttt{Nothing}, d_{\mathsf{S}}))\, \texttt{>>}\, \texttt{mask}_{\mathsf{A}} & \{\text{prop run}\}
\end{array}
$$

## 5 Mechanizing the Logic in Coq

The syntax, denotational semantics, and equational logic of CC have all been mechanized in the Coq [8] theorem prover. In lieu of separate syntaxes for type judgments and terms as presented in the preceding sections, the Coq formulation uses a strongly-typed term formulation as suggested by Benton et al [7]. We combine this with a dependently typed denotational semantics along the lines of Chlipala [9] (see Chapter 9). The payoff of this approach is that we do not need to establish many of the usual properties such as progress and subject reduction that usually accompany an operational approach. Furthermore, strongly-typed terms are much more amenable to the use of Coq's built-in system of parametric relations and morphisms. Just a handful of relation and morphism declarations lets us reuse many of Coq's standard tactics (e.g., `replace` and `rewrite`) when reasoning in the CC logic.

Figure 9 presents an example equational judgment rule from the Coq development, representing the clobber rule. The only major difference between the Coq formalism and the corresponding rule in Section 3 has to do with the need for explicit subtyping: the term constructor `subsume` casts a term from a supertype to a subtype, and requires as an argument a proof term showing that the subtyping relationship holds (here called `S_just`). The full development in Coq is available by request.

## 6 Related Work

Klein et al. [15] describe their experience in designing, implementing and verifying the seL4 secure kernel. Monads are applied as an organizing principle at all levels of the seL4 design, implementation and verification. Their model of effects is different from the one described here. The seL4 monadic models encapsulate errors, state and non-determinism. The notions of computation applied here include concurrency and interactivity ($R$ and $Re$, respectively) as well as layered state ($K$). Also, the type system underlying the seL4 models does not include

```
J_clobber : ∀ (Γ:list ty) (d:domain) (t:ty)
              (te:term Γ (tyK (onedom d) t)),
   let S_just := S_tyK (onedom d) (union (onedom d) (onedom d))
                       (S_refl tynil) (clobber_obligation _)
   in  eq_judgment (subsume S_just (nullbindK te (mask Γ d))) (mask Γ d)
```

**Fig. 9:** Expressing the clobber rule in Coq

effect types. The present work models faults via simulation on distinct domains rather than as part of an integrated model of effects. Cock, Klein and Sewell [10] apply Hoare-style reasoning to prove that a design is fault free. Kernels in the CC are fault-free as a by-product of our type system and it would be interesting to investigate whether the application of CC in the seL4 construction and verification process would alleviate some verification effort.

Another similarly interesting question is to consider the impact of integrating layered state and resumption-based concurrency into their abstract, executable and machine models. One design choice, for example, concerns the placement of preemption points—i.e., places where interrupts may occur—within the seL4 kernel specifications. It seems plausible that interrupt handling in seL4 might be simplified by the presence of an explicit concurrency model—i.e., resumptions. It also seems plausible that aspects of the seL4 design and verification effort might be reduced or abstracted by the inclusion of effect-scoping mechanisms like layered state and effect types—e.g., issues arising from the separation of kernel space from user space.

Language-based security [27] seeks to apply concepts from programming languages research to the design, construction and verification of secure systems. While fault isolation is generally considered a safety property, it is also a security property as well in that an unconfined fault may be used for a denial of service attack and also as a covert channel. Monads were first applied within the context of language-based security by Abadi et al. [1], although the use of effect systems seems considerably less common in the security literature. Bartoletti et al. [4, 5] apply effect systems to history-based access control and to the secure orchestration of networked services. Bauer et al. [6] applied the combination of effect systems and monads to the design of secure program monitors; their work appears to be the first and only previously published research in security to do so. The current work differs from theirs mainly in our use of interaction properties of effects that follow by the construction of the monads themselves. These by-construction properties provide considerable leverage towards formal verification. Scheduler-independent security [26, 19]) considers the relationship between scheduling and security and investigates possibilistic models of security that do not depend on particular schedulers. A natural next step for the current research is to investigate scheduler freedom with respect to CC kernels (e.g., Definition 4).

## 7 Conclusions

The research described here seeks to apply tools and techniques from programming languages research—e.g., monads, type theory, language compilers—to the production of high assurance systems. We are interested, in particular, in reducing the cost of certification and re-certification of verified artifacts. The questions we confront are, in terms of semantic effects, what can untrusted code do and, given a semantic model of untrusted code, how can we specify a system for running it safely in isolation? Untrusted code can read and write store obviously.

It can fail—i.e., cause an exception. It can also signal the operating system via a trap. These effects are inherited from the machine language of the underlying hardware.

Previous work [14] explored the application of modular monadic semantics to the design and verification of separation kernels. Each domain is associated with an individual state monad transformer [18] and the "layered" state monad constructed with these transformers has "by-construction" properties that are helpful for verifying the information flow security. The present work builds upon this in the following ways. Our previous work did not consider fault effects, and the layering approach taken in that work does not generalize to handle faults. It is also interesting, albeit less significant, that the semantics of the CC effect system is organized by state monad transformers. Structuring the semantics of Wadler's MONAD language with monad transformers was first suggested by Wadler [29] and the present work, to the best of our knowledge, is the first to actually do so. Although the current work focuses on isolation, we believe that it can be readily adapted to MILS (multiple independent levels of security) systems by refining the effect types to distinguish, for example, reads and writes on domains (as Wadler's original MONAD language did). One could then express, for example, a high security handler that is allowed to read from, but not write to, domains lower in the security hierarchy.

Kobayashi [16] proposed a general framework for reasoning about monadic specifications based in modal logic in which monads are formalized as individual modalities. Nanevski elaborated on the monad-as-modality paradigm, introducing a modal logic for exception handling as an alternative to the exception monad [23]. Nanevski's logic is an S4 modal logic in which necessity encodes a computation which may cause an exception—i.e., $\Box_C A$ represents a computation of an $A$ value that may cause an exception named in set $C$. Modal logics have been adapted to security verification by partially ordering modalities to reflect a security lattice by Allwein and Harrison [2]. We are currently investigating the integration of the monad-as-modality paradigm with security-enabled partially-ordered modalities into a modal logic for verifying the security of monadic specifications in the confinement calculus and related systems.

## References


1. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *26th ACM Symp. on Principles of Programming Languages*, pages 147–160, 1999.
2. G. Allwein and W. L. Harrison. Partially-ordered modalities. In *Advances in Modal Logic*, pages 1–21, 2010.
3. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.
4. M. Bartoletti, P. Degano, and G. L. Ferrari. History-based access control with local policies. In *8th International Conference on the Foundations of Software Science and Computation Structures*, FOSSACS'05, pages 316–332, 2005.
5. M. Bartoletti, P. Degano, and G.L. Ferrari. Types and effects for secure service orchestration. In *19th IEEE Computer Security Foundations Workshop*, 2006.

6. L. Bauer, J. Ligatti, and D. Walker. Types and effects for non-interfering program monitors. In *Software Security—Theories and Systems.*, volume 2609 of *LNCS*, pages 154–171, 2003.
7. N. Benton, C.-K. Hur, A. Kennedy, and C. McBride. Strongly Typed Term Representations in Coq. *Journal of Automated Reasoning (to appear)*, pages 1–19.
8. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.
9. A. Chlipala. Certified programming with dependent types. Book draft of April 12, 2012, available online at http://adam.chlipala.net/cpdt/.
10. D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 167–182, 2008.
11. J. Goguen and J. Meseguer. Security policies and security models. In *Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.
12. S. Goncharov and L. Schröder. A coinductive calculus for asynchronous side-effecting processes. In *Proceedings of the 18th International Conference on Fundamentals of Computation Theory*, pages 276–287, 2011.
13. W. L. Harrison. The essence of multitasking. In *11th International Conference on Algebraic Methodology and Software Technology*, pages 158–172, July 2006.
14. W. L. Harrison and James Hook. Achieving information flow security through monadic control of effects. *Journal of Computer Security*, 17(5):599–653, 2009.
15. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009.
16. S. Kobayashi. Monad as modality. *Theor. Computer Science*, 175(1):29 – 74, 1997.
17. B. Lampson. A note on the confinement problem. *CACM*, 16(10):613–615, 1973.
18. S. Liang. *Modular Monadic Semantics and Comp.* PhD thesis, Yale Univ., 1998.
19. H. Mantel and H. Sudbrock. Flexible scheduler-independent security. In *Proc. of the 15th European Conf. on Research in Comp. Security*, pages 116–133, 2010.
20. W. Martin, P. White, F. S. Taylor, and A. Goldberg. Formal construction of the mathematically analyzed separation kernel. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 133–141, 2000.
21. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.
22. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.
23. A. Nanevski. A Modal Calculus for Exception Handling. In *Intuitionistic Modal Logics and Applications Workshop (IMLA '05)*, June 2005.
24. F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis.* 1999.
25. S. Peyton Jones, editor. *Haskell 98 Language and Libraries, the Revised Report.* Cambridge University Press, 2003.
26. A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. of the 19th IEEE Workshop on Comp. Sec. Found.*, pages 177–189, 2006.
27. A. Sabelfeld and A.C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
28. W. Swierstra and T. Altenkirch. Beauty in the beast. In *Proceedings of the ACM SIGPLAN Haskell Workshop (Haskell07)*, pages 25–36, 2007.
29. P. Wadler. The marriage of effects and monads. In *Proceedings of the* $3^{rd}$ *ACM SIGPLAN International Conference on Functional Programming*, pages 63–74, 1998.