Towards Semantics-directed System Design and Synthesis

William L. Harrison[1], **Benjamin Schulz**[1], **Adam Procter**[1], **Andrew Lukefahr**[2] **and Gerard Allwein**[3]
[1]Department of Computer Science, University of Missouri, Columbia, MO, USA.
[2]Department of Electrical Engineering and Computer Science, University of Michigan-Ann Arbor, MI, USA.
[3]US Naval Research Laboratory, Code 5543, Washington, DC, USA.

Abstract— *High assurance systems have been defined as systems "you would bet your life on." This article discusses the application of a form of functional programming—what we call "monadic programming"—to the generation of high assurance and secure systems. Monadic programming languages leverage algebraic structures from denotational semantics and functional programming—monads—as a flexible, modular organizing principle for secure system design and implementation. Monadic programming languages are domain-specific functional languages that are both sufficiently expressive to express essential system behaviors and semantically straightforward to support formal verification.*

Keywords: Formal Methods, Computer Security, Programming Languages

1. Introduction

System software is notoriously difficult to reason about either formally or informally and this, in turn, greatly complicates the construction of high-assurance, secure systems. In our view, the difficulty stems from the conceptual distance between abstract models of secure systems and their concrete implementations. System models are formulated in terms of high-level abstractions while system implementations reflect the low-level and concrete details of hardware, machine languages and C. Typically, there is no apparent relationship between the system model and implementation to exploit in verifying critical system properties, and this disconnect impedes the construction of computer systems with verified security policies.

This paper describes ongoing research that pursues a novel approach towards bridging this conceptual gap: synthesizing implementations of systems directly from formal models of security in a manner verified to preserve the system security property. The particular systems we focus on are Rushby's classic security kernel design: *separation kernels* [1]. Separation kernels partition processes by security level into distinct "domains" (where all interdomain communication is mediated by the kernel) and enforce a non-interference-style security discipline called *domain separation* (see Figure 1). In a separation kernel, processes on different domains behave as if they are on distinct nodes in a networked distributed system while they, in fact, execute on shared hardware.

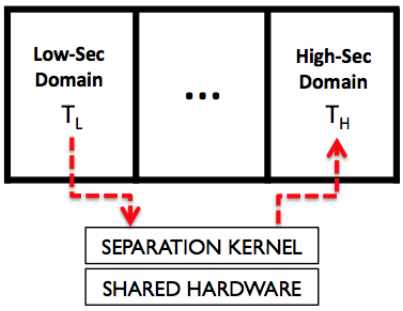


Fig. 1: A separation kernel mediates all inter-domain communication, thereby enforcing its security policy. The dotted arrow designates permitted information flows.

The methodology being explored is a calculational approach seeking the development, implementation and verification of compilation strategies that preserve domain separation. The main vehicle for our methodology is a domain-specific language, called HASK (for High-Assurance Security Kernel), and the aim of this approach is not to develop a single, specific security kernel, but rather a class of such kernels embodied as HASK programs.

This paper presents an overview of the design, implementation and verification of HASK programs. It does not report new technical innovations or scientific discoveries, but rather it presents an overview of the research, as well as the philosophy underlying the research, that is currently being pursued by the authors. Furthermore, an overview of both the philosophy that motivates our approach to high assurance and secure systems. The remainder of this section discusses and motivates the design of the HASK language. Section 2 places this research in the context of efforts to discover a science of cybersecurity. Section 3 describes two formalisms that provide a foundation for our work, monadic semantics [2] and channel theory [3].

1.1 Motivating the HASK Design

The "bread and butter" activities for any kernel include concurrency (e.g., task scheduling, synchronization, etc.), managing IO, and handling asynchronous exceptions and system calls. Implementing a kernel means supporting these functionalities among others.

If high assurance is a goal for the kernel, then the choice of implementation language is critical. Verifying formally that

the kernel possesses certain properties (e.g., that it obeys an information security discipline) means proving mathematically that the kernel code itself possesses the desired properties. This, in turn, requires that the implementation language must possess a rigorous semantics. For higher assurance, the kernel language compiler and possibly other links in the toolchain must also be verified to preserve the desired properties.

Formal verification of code? This is frequently the point at which functional languages are mentioned, because several such languages (e.g., Haskell and ML) have rigorous semantic specifications [4], [5], [6], [7], [8] for large parts of the language.

The choice of a functional language for implementing the kernel is complicated by several unpleasant facts: "bread and butter" kernel behaviors (e.g., file handling, concurrency, exceptions, destructive update, etc.) are called the "Awkward Squad" in the functional programming community [9] precisely because they are difficult to handle in functional languages.

For the sake of discussion, let us consider the Haskell functional language [10] as an implementation language. Systems-level programming in Haskell inevitably involves the "IO monad". And what is the IO monad? In the memorably colorful words of Simon Peyton Jones (the principal architect of the Glasgow Haskell Compiler and a leading light in the Haskell community), the IO monad is a giant "sin-bin" [9] into which all real world impurities are swept. These impurities (e.g., file handling, concurrency, exceptions, destructive update, etc.) are called the "Awkward Squad" in the functional programming community precisely because they are difficult to handle in functional languages. Yet, the Awkward Squad is essential to kernel programming—after all, if a kernel isn't about handling concurrency, exceptions, etc., what is it about?

1.2 Making the Awkward Squad Less Awkward

HASK handles the Awkward Squad in two ways. HASK is a standalone domain-specific language (DSL) with its own compiler. HASK looks like Haskell [10], and, in fact, HASK programs can be executed by Haskell implementations with only some trivial syntactic changes. But, HASK's expressiveness is intentionally reduced to simplify its semantics, implementation and verification.

The other way that the Awkward Squad is made less awkward is with the "monadic programming" model of HASK. This distinguishes this research from that of others applying functional languages to system software [11], [12]. Historically, monads are used in two contexts: denotational semantics and functional programming. In denotational semantics, monads are algebraic structures representing computational effects, while within functional programming, monads are a programming abstraction for modularity akin to object orientation. Programming in HASK differs from ordinary functional programming practice in at least one key respect: monads are first-class language constructs.

Now is as good a time as any to explain a confusing misnomer in the Haskell world: the aforementioned IO monad in the Haskell language is a monad in name only. A monad [2], [13] is an algebraic structure is the same sense that a group, a ring or a vector space is an algebra; that is, it has operations defined on it of a certain type that obey certain defining equations. The Haskell IO monad has operations of the right type defined on it, but they are not guaranteed to obey the defining equations of a monad. Further muddying this already muddy water, Haskell also has a built type class called *Monad* and members of that type class, like the IO monad, are not necessarily monads in the algebraic sense. A Haskell programmer, having defined a *Monad*, would have to prove that his *Monad* is really a monad. In this paper, when we write "monad", we really mean monad in the rigorous, mathematical sense of the word. Monads are explained further in Section 3.

At this point, one might reasonably ask whether a whole new language is even necessary? Can't an existing language suffice—why a standalone language and not just a new library? High assurance is one of our main desiderata, and, if we are to have a chance of verifying our designs and tools, then we must program them in a language with a rigorous semantics. There are simply no general purpose languages with such a semantics. Such languages have so much "stuff" in them that establishing a humanly tractable language semantics would seem to be difficult, if not impossible.

HASK is a standalone language and requires its own compiler. While implementing HASK via existing Haskell compilers is possible, this style of implementation leaves much to be desired, both from the points of view of efficiency and high assurance. From an efficiency perspective, the kernel implementation produced by the GHC compiler is large and retains artifacts from the Haskell run-time system ill-suited to an operating system kernel (e.g., garbage-collection and closures). Neither GHC nor its runtime system were designed for formal verification.

2. What is a Science of Security?

There has been a lot of interest recently among various funding agencies in the US federal government in research seeking to uncover the "science of security." This interest stems, no doubt, from our increasing reliance on computational systems for everything from national defense to commerce to medicine. Computing is becoming ubiquitous and, in order to have any confidence that all this computational infrastructure is really worthy of trust, we really need to understand security from first principles. If we do not possess a science of security, then how do we distinguish trustworthy systems from untrustworthy ones?

The remainder of this section considers issues presented by a science of security and, in particular, what the term "science" might entail when combined with "security." The presentation is admittedly high level and conversational in tone. It is the authors' intention to ask questions rather than provide answers. In fact, the authors hasten to add that they have no pretensions to possessing the answers to these questions. This section contains several digressions into the history of mature scientific and engineering disciplines—chemistry and civil engineering—with the purpose of indicating what "maturity" entails for security. The authors are neither chemists, civil engineers, nor historians of science. Our purpose is to present an overview of the philosophy that motivates our research in high assurance computing.

2.1 Is a "Science" of Security really necessary?

Think about it this way. If you worked for a company that manufactured dynamite and none of the company's chemists had ever heard of Mendeleev's periodic table of elements, would you continue to work for that company? If the chemists merely worked from a recipe for dynamite, but did not understand the underlying principles of chemistry, they would not be able to distinguish safe synthesis of dynamite from unsafe synthesis. The probable results would be quite dramatic and unpleasant for all concerned.

Security as a discipline is now a collection of ideas and techniques. A popular textbook on security by Matt Bishop [14] contains thirty-five chapters on subjects ranging from access control, cryptography, security models, etc. What unites these seemingly disparate subjects? One frequently has the feeling when reading the literature of computer security that these areas are somehow connected to one another on some deep level, but the precise nature of those connections is not clear now, nor will they become so, until security is understood from first principles.

Security has been studied since the early 1970's [15], making it a decade or two younger than computing science as a whole (the umbrella term "computing science" includes all computational disciplines; e.g., computer science, computer engineering, etc.). That it has not yet evolved into a mature discipline should not be surprising when one bears in mind that older, more mature scientific and engineering disciplines (e.g., chemistry and civil engineering) became so over the course of centuries.

A historical digression provides a useful perspective and sense of proportion on the prospects for security science. Consider civil engineering, which can be said, with some justice, to be the most mature engineering discipline, having roots at least as far back as ancient Egypt. How quickly did scientific theory have an impact on civil engineering practice?

Consider the (admittedly anecdotal) example of the application of the differential and integral calculus to civil engineering. Both Newton and Leibniz published their respective versions of the calculus by 1675. Charles Augustin de Coulomb's paper *Essai sur une application des maximis règles et de minimis à quelques problèmes de statique relatifs à l'architecture*, first published in 1776, was the first publication to apply differential and integral calculus to civil engineering problems and, even then, calculus did not penetrate the civil engineering practice until the early to mid-19^{th} century [16]. Thomas Tredgold's classic treatise [17], published in 1820, formulates construction techniques in terms of trigonometry alone and does not mention erstwhile advanced ideas from physics and mathematics (e.g., statics and calculus).

Coulomb and Tredgold illustrate a parallel between the civil engineering of the early 19^{th} century and computing science as we know it today. Coulomb was a theorist who focused mainly on what we might now call mathematical physics. His approach was formal and grounded in mathematics. Tredgold was a self-taught engineer and his aforementioned treatise provides a collection of techniques for building structures (e.g., bridges, buildings, etc.) along with practical advice on the kinds of materials to be used. Tredgold wanted to know how to construct a thing and Coulomb wanted to understand why that thing's construction worked.

2.2 Foundations for the Science of Security

Tredgold and Coulomb are the perfect "poster children" for two respective and (alas) currently distinct camps within computing science: practical computer engineers and formal methods scientists. Formal methods is the application of mathematics to the design and construction of hardware and software systems. By combining design and development with rigorous mathematical analysis, formal methods seeks to construct systems that provably possess particular properties (e.g., are secure, fault-tolerant, safe, correct, etc.).

According to Parnas [18], formal methods was founded in 1967 by Robert Floyd with his paper *Assigning Meanings to Programs* [19]. Why assign meaning to a program? Because if you want to prove that a program has a particular property, you must first possess a rigorous—i.e., mathematical—definition of that program: no mathematical meaning, no mathematical proof. Similarly, if you want to know why a system S is secure, you must first possess a mathematical definition, in some form, of S. Here, "system" is interpreted very broadly; S may be a program, operating system, circuit, or, perhaps, some combination of heterogenous systems each possessing an individual mathematical definition. Having a controlled vocabulary for computational systems is a principal motivation for our use of monadic semantics [2].

Any science of security worth its salt must provide the rigorous definition of "secure" for the context of S, irrespective of the form S may take. To be worthy of the title "science", notions of security must be independent of particular, concrete instances of secure systems. This is not

an exotic requirement in any sense. Arithmetic, for example, does not provide individual notions of natural numbers for each kind of object that one would count—there is one notion of natural number rather than "natural numbers for oranges" and "natural numbers for apples," etc. Having a logical framework for specifying information flow between heterogenous subsystems motivates our interest in channel theory[3].

2.3 Challenges to a Science of Security

Formal methods has acquired a reputation in some quarters as a purely academic endeavor, where "purely academic" is generally taken to mean "unpractical." This is not entirely fair as formal methods have made inroads into industrial practise [20], [21], [22], [23], [24], [25], but it must be recognized that formal methods are the exception and not the rule in industry [18].

David Parnas, one of the leaders in formal methods research since its inception, recently published a thought-provoking, critical assessment of the current state of the art in formal methods research [18]. While Parnas' essay was concerned with formal methods for software, his observations are relevant to formal methods generally. Parnas identifies three negative developments in software engineering, two of which are especially relevant to the development of a science of security: the gaps between research and practice and between computing science and classical mathematics. Each gap presents challenges to be overcome in the pursuit of a science of security and below we discuss these challenges, as well as several others, in the context of a science of security.

2.3.1 Gap between Computing Science & Mathematics

Parnas observes that mathematics and theoretical computing science are separate to a surprising degree. Mathematics possesses a vast, deep body of concepts, structures and techniques that it has developed over the course of millenia. Theoretical computing science has only partaken of a tiny sampling of what mathematics has to offer. Formal methods is now at an early stage in its development where a variety of formalisms from mathematics are being investigated for their relevance to a science of computation. Formal methods may appear sometimes to produce "toys." But the experimentation underlying formal methods research is the early part of an important intellectual exploration that must be made if computing science is to really become a science. Mathematics does not apply itself. It is hard work. Understanding Newton's presentation of differential and integral calculus in terms of "fluxions" does not immediately suggest Coulomb's application of calculus to calculating the strength of retaining walls. The mathematical raw material may be present in current mathematics, but fashioning that raw material into constructive engineering can take significant effort.

2.3.2 Gap between Computing Science Research & Practice

Parnas observes another gap between the Coulombs and Tredgolds in current computing science research. Formal methods research produces results that are not connected to engineering reality. Practical engineers sometimes rely on toolchains that are known to be deeply flawed, but continue to rely on them for lack of any better alternative.

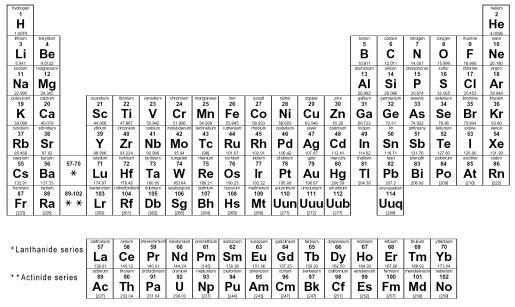
This is the problem of "legacy software" writ large. An investment in some form of intellectual capital is unlikely to be abandoned by the investor. A formal methods researcher may have invested significant time in the development of an idea or technique and will not readily abandon these cherished results even if they have not yet born fruit. A practical engineer wants to produce artifacts in the here and now and will continue to apply flawed technology to "get the job done."

Neither the Coulombs nor the Tredgolds are being unreasonable. The formal methods researcher knows that, historically, every powerful and revolutionary idea started weak and had to be developed before its power could be recognized. Demands that the acorn become a might oak overnight are themselves unreasonable. Practical engineers cannot be expected to abandon proven, yet flawed, technology to start from scratch with each project. Nothing would ever get done.

Yet both the Coulombs and the Tredgolds must recognize that their ideas, tools and technologies may have a shorter lifespan than they imagine. And that, in the history of computing science, the short lifespan of particular ideas may ultimately be judged to be a good thing.

2.3.3 The Tower of Babel

One of the things that makes chemistry a mature science is that it has a controlled vocabulary. Chemists do not argue over how to express the molecule for water; it is simply H_2O. Neither do chemists argue over the meaning of "double bond," "alkane" or "aromatic compound." Chemistry's maturity is made manifest in Mendeleev's periodic table of elements:



The periodic table obviously does not capture all of chemical knowledge, but it does provide a foundation with which chemists can develop and communicate ideas to other chemists. Computing science in general and computer secu-

rity in particular have no such organizing principles. There is a "Tower of Babel" problem in today's computing disciplines consisting a variety of ideas and techniques that, although they may seem somehow interdependent, the precise nature of their relationships is not clear. Indeed, exploring their connections does not appear to excite much interest.

The authors interest in channel theory (described below in Section 2.5) is motivated by its capability of expressing the relationships between subsystems characterized within different formalisms.

2.4 Security Theory in the Present

This section presents an example of a theory of security, Goguen and Meseguer's classic noninterference model [26], to illustrate the state of the art in security models. Noninterference may seem to be an odd choice to illustrate the state of the art since Goguen and Meseguer first published the model in 1982, but noninterference was and is enormously influential (e.g., as of this writing, it has 1145 distinct citations according to Google Scholar) and continues to inspire refinements and extensions. A manifestly incomplete list of noninterference descendants includes: Rushby [27], McCullough [28], McLean [29], Zakathinos & Lee [30]).

Noninterference is a formal model of security based on abstract state machines. It requires that, roughly speaking, low-security outputs are unaffected by high-security inputs to an abstract state representing the system. To understand noninterference, consider a system S with exactly two threads, A and B, and a trace t of a particular system execution. Trace t is a sequence of A and B operations reflecting the scheduling of the threads by S. There are three operations on S. For state σ, *run*(t, σ) executes trace t and *output*(σ, B) is the system output according to B. The trace *purge*(t, A) is the subtrace of t with all of A's operations removed. For a given execution sequence and input state, t and σ, thread A does not interfere with B if, and only if,

$$\textit{output}(\textit{run}(t, \sigma), B) = \textit{output}(\textit{run}(\textit{purge}(t, A), \sigma), B)$$

Most information flow security models are based either on the *noninterference* model of Goguen and Meseguer or on variants of it [31]. Such security models specify end-to-end system security policies in terms of the "views" of the system as a whole by groups of users/processes. Views are typically characterized by partitioning global system inputs and outputs and associating groups of users/processes with these partitions. These input and output partitions determine the view of its associated group. Noninterference-based security policies will require that, for example, changes in a high level security input partition will result in no change to a low level output partition.

2.5 Channel Theory and Information Security

It has long been held that information flow security models should be organized with respect to a theory of information,

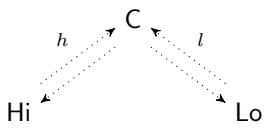


Fig. 2: Channel Diagram relates component-level views (Lo and Hi) to a global view (C).

but typically they are not. What, for example, is the unit of information transferred in the noninterference model described above? The appeal of a information-theoretic foundation for information flow security seems natural, compelling and, indeed, almost tautological. Channel theory—a theory of information based in logic—can provide a foundation for security models based in noninterference. The first and last authors' recent publication [32] demonstrates how a McLean's hierarchy of information security policies [29] can be neatly captured in channel theory. This result suggests that channel theory is a useful organizing principle for information flow security.

The partitioning associated with noninterference security specifications implicitly divides a system into subsystems or components and defines permissible information flow in terms of the inputs and outputs resulting from component interaction. The main constructions of channel theory—classifications, infomorphisms, and channels—can be used to formulate an information-theoretic characterization of information flow security that is, in contrast to the traditional approach, explicitly component-based.

Figure 2 presents a channel diagram in which information flow between local components, Lo and Hi, is mediated by a global component, C. Individual components are called *classifications*. Classifications are local logical characterizations of components and their views and each view corresponds to an individual classification in channel-theoretic information security. In a channel diagram, information flow between component classifications occurs along *infomorphisms* (h and l) and is controlled by the channel *core* (C).

Mathematical logic, as most of us learn it [33], is not modular: there is no structural notion of logic "components" and "composition" akin to modules or classes in programming languages. The standard approach presents a logic as a monolith containing a single logical language, a single set of inference rules and a single class of structures or interpretations for the language. Channel theory [3] is a framework for formulating logical specifications in a modular (i.e., "distributed") manner.

The authors view channel theory as playing a similar role in mathematical logic as computational monads play in the semantics of programming languages with effects. Both are frameworks that support structuring specifications in a modular manner and neither is, strictly speaking,

essential to their respective disciplines. That is, one can formulate language semantics without monads, just as one can formulate logical specifications without channel theory. The level of abstraction supported by monads have made monadic structuring one of the most important developments in language semantics since the late 1980's [2].

3. Monadic Programming for Secure Systems

This overview of monads and their security properties is, of necessity, extremely brief and high-level. It is presented with as few technical details as is possible so that the reader can quickly understand can quickly comprehend the gist of the approach. The risk of such a high-level presentation is that it may seem like we are describing nothing at all. The interested reader can find a thorough technical presentation in the references (especially, Harrison and Hook [34]).

3.1 Take Separation.

Goguen-Meseguer noninterference defines an abstract security model that does not include or imply any implementation strategy. We use an alternative to Goguen-Meseguer noninterference called *take separation* [34] that includes sufficient operational content to drive an implementation. Take separation is an algebraic security specification, formulated in terms of interactions of system operations or commands, and is provably equivalent to Goguen-Meseguer noninterference. Monadic information security possesses an advantage over traditional Goguen-Meseguer state machine security disciplines in that the approach relates to programs through language semantics, thus removing the need to rely upon external formalisms for program verification. One can always construct executable models obeying a take separation policy using monadic semantics [34].

Take separation is defined as follows for two processes, each of which are sequences of operations: $A = (a_0; a_1; \cdots)$ and $B = (b_0; b_1; \cdots)$. A system execution, ω, consists of any order-respecting interleaving of A and B's commands; a round-robin execution, for example, would be $\omega = (a_0; b_0; a_1; b_1; \cdots)$. To define $B\,|\,A$, we assume the existence of function that purges B operations from a system execution, called $takeA$, and an B operation that masks out B's outputs, called $maskB$. The function $takeA$ purges B operations, leaving only A operations, and the operation $maskB$ scrubs all of B's outputs. Equations defining $takeA$ and $maskB$ are: $takeA\, n\, \omega = (a_1\, ;\, \cdots\, ;\, a_n)$, for all $n \geq 0$, and $(b\, ;\, maskB) = maskB$ for any operation b on B. Process B is *take separate* from A in ω if, and only if, $(a_1\, ;\, b_1\, ;\, \cdots\, ;\, a_n\, ;\, b_n)\, ;\, maskB = (takeA\, n\, \omega)\, ;\, maskB$ for all $n \in \{0, 1, \cdots\}$. This statement requires that A's outputs are invariant with respect to the actions of B in ω. That B's outputs are masked on both sides of the equation is important because B's outputs must be the same on both sides of the equation.

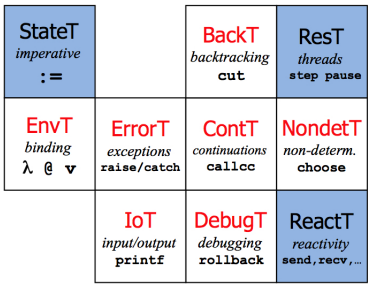


Fig. 3: The periodic table of programming languages. Each element in the table encapsulates a language constructor and molecules (i.e., individual languages) are combinations of elements. In the denotational semantics literature, "molecules" are also known as monads and elements are also known as "monad transformers" or "monad constructors."

3.2 Monadic Separation Kernels from 10,000 Feet

The best way to think of a monad is as a DSL. That is, it's just a small programming language. Monad DSLs are also modular. Bigger DSLs can be made by combining individual language building blocks together. Following MonadLab [35], an additive notation is used for composing building blocks; e.g., for blocks, b_1 and b_2, their composition is $b_1 \oplus b_2$. In the literature, DSL building blocks are usually called either monad transformers [36] or monad constructors [13]. Here's an important fact: any command from any programming language is expressible as a DSL constructor [13]. Just as Mendeleev's periodic table is a controlled vocabulary for chemistry, DSL constructors are a controlled vocabulary for defining programming languages (see Fig. 3).

The monad DSL for a Rushby-style separation kernel combines concurrency operations with one monad DSL for stateful effects per separated security domain. These "Rushby monad" DSLs come with properties by construction that are useful for verifying non-interference-based security properties. A monadic separation kernel with two domains, *Hi* and *Lo*, is defined in terms of five monad DSLs:

```
monad H  = ReactT Req Rsp ⊕ (StateT HA)
monad L  = ReactT Req Rsp ⊕ (StateT LA)
monad K  = (StateT HA) ⊕ (StateT LA)
monad R  = ResT ⊕ K
monad Re = ReactT Req Rsp ⊕ K
```

The building blocks, `(StateT HA)` and `(StateT LA)`, add commands for reading and writing from address spaces `HA` and `LA`, respectively. Note that this means that commands defined by these building blocks write to different address spaces by construction. The building block, `ReactT Req Rsp`, adds commands for interacting with the kernel. User processes running on either *Hi* and *Lo* can send a trap to the kernel of type `Req`. The kernel may then signal

a response of type `Rsp`. The building block `ResT` allows threads to be separated into time slices and executed by the kernel.

The monad DSLs `H` and `L` define the languages of user processes on *Hi* and *Lo*. This means that *Hi* (*Lo*) code can access the `HA` (`LA`) address space as well as signal the kernel. By construction, *Hi* and *Lo* processes can only interact, if permitted, via the kernel, because their commands cannot access each other's address spaces. The monad DSL for the separation kernel is defined by the three monads DSLs `K`, `Re` and `R`. The kernel has access to all of `HA` and `LA` via `K`. The kernel is a tail-recursive function that takes a user program (i.e., from either an `H` or `L`), services any request it may have signaled, and dispatches a time slice of the use r process (an `R` object). Each monad DSL has a sequencing operation, », that corresponds to ";" in languages like C or Java.

By construction, we know a number of properties about programs in the monad DSLs `H` and `L`. Each has a special operator, `zeroH` and `zeroL`, that initializes the address spaces `HA` and `LA`, respectively. For any commands `hi` and `lo`, from `(StateT HA)` and `(StateT LA)`, respectively, we know that a number of equations hold when executed by the kernel, including:

```
1. hi >> lo    = lo >> hi
2. lo >> zeroH = zeroH >> lo
3. hi >> zeroH = zeroH
```

The most important of these properties from the point of view of security is `1.` (called *atomic noninterference* [34]). Because any such `hi` and `lo` commute, they are independent of one another. Property `2.` says that `lo` and `zeroL` are also independent. But property `3.` shows that `zeroH` cancels `hi`. We also know that » is associative.

A trace of a system execution is expressed in the `R` DSL, meaning that the signals have already been serviced. The result of the kernel servicing a process request is reflected by an operation on that process's address space (e.g., copying the content from a return register into a local address). A prefix of a trace will contain interleaved operations from each domain; e.g., a prefix might have the form: $\texttt{lo}_0$ » $\texttt{hi}_0$ » $\texttt{lo}_1$. For such a trace prefix, cancel out all of the *Hi* operations with `zeroH`:

```
(lo₀ >> hi₀ >> lo₁) >> zeroH
    = lo₀ >> hi₀ >> lo₁ >> zeroH  { assoc. }
    = lo₀ >> hi₀ >> zeroH >> lo₁  { 2. }
    = lo₀ >> zeroH >> lo₁         { 3. }
    = lo₀ >> lo₁ >> zeroH         { 2. }
    = (lo₀ >> lo₁) >> zeroH       { assoc. }
```

One will observe that this is precisely what is required by take separation. The `zero` operations defined by the `StateT` building blocks play the same role as *purge* does for the Goguen-Meseguer model.

The proof above shows that, as long as the kernel does not allow an illegal information flow, then the whole system has take separation. This follows immediately by the construction of the monadic DSLs above. The kernel must still be verified. Because it can communicates with each domain, a kernel could break the security policy. The good news is that the kernel description is short; for an example, see Fig. 5, which is explained below in Section 4.1.

3.3 The State Monad

This section presents the definition of a monad in Haskell for the sake of completeness. This section is not required to understand the rest of the paper.

Monads are typically represented in functional programming languages like Haskell [10]. The representation of `S` in a Haskell-like notation consists of the function and type declarations in Figure 4. The monadic type `S a` is a function type taking an input state of type `Store` to a product of outputs of type `(a×Store)` representing the output value and state respectively. Lambda notation is used to express functions in functional languages; e.g., $\lambda \texttt{x}.\texttt{x} + 1$ is a function that takes an integer `x` as input and returns its increment as output. In `x»y`, the null bind operator (») threads an input state `s` through `x`, producing output state `s'`, that is then threaded through `y`.

The DSL for the state monad `S` is displayed in Figure 4 for some given some type `Store`. The command, `(return 99) : S Int`, is a trivial computation that returns the value `99`. The command, `get : S Store`, reads and returns the current `Store`. Neither `return` nor `get` change the current `Store` state. Given a function `f : Store→Store`, `(update f)` transforms the current state by applying `f` to it; it produces a nil value of type `()`. The bind commands, »= and », sequence `S`-computations together analogously to "`;`" in C or Java; we will only use the so-called null bind (») in this paper.

```
return : a → S a
(>>=)  : (S a) → (a → S b) → S b
(>>)   : S a → S b → S b
update : (Store→Store) → S ()
get    : S Store
```

```
S a      = Store→(a×Store)
x >> y   = λs.let (d,s') = x s in y s'
update f = λs.((),f s)
return v = λs.(v,s)
```

Fig. 4: The State Monad `S` (top) and its representation in a functional programming language (bottom).

```
-- DSL Construction:
type Hi  = Addr -> Int
type Lo  = Addr -> Int
data Req = Cont | Bcst Int | Rcv
data Rsp = Ack  | Rcvd Int
monad K  = (StateT Hi) ⊕ (StateT Lo)
monad R  = ResT ⊕ K
monad Re = (ReactT Req Rsp) ⊕ K

-- Kernel-level  Data
type Sys = (Q(Re()),Q Int,Q Int)
data HdlrOp a  = Halt | Serv_H a | Serv_L a
type Handshake = (Req,Rsp -> K (Re ()))

resched :: Sys -> Re () -> R ()
resched (ts,l,h) t  = kl (ts << t,l,h)
```

```
kl :: Sys -> R ()
kl (∅,_,_)     = return ()
kl (t◇ts,l,h) = disp (onsig t)
 where
   disp :: HdlrOp Handshake -> R ()
   disp Halt               = kl (ts,l,h)
   disp (Serv_H (Cont,k))   = step_H (k Ack) >>= resched (ts,l,h)
   disp (Serv_L (Cont,k))   = step_L (k Ack) >>= resched (ts,l,h)
   disp (Serv_H (Bcst m,k)) = kl (ts << rspd_H k Ack,l,h << m)
   disp (Serv_L (Bcst m,k)) = kl (ts << rspd_L k Ack,l<<m,h<<m)
   disp (Serv_H (Rcv,k))    = case h of
                    ∅      -> kl (ts << t,l,∅)
                    (m◇hs) -> kl (ts << rspd_H k (Rcvd m),l,hs)
   disp (Serv_L (Rcv,k))    = case l of
                    ∅      -> kl (ts << t,∅,h)
                    (m◇ls) -> kl (ts << rspd_L k (Rcvd m),ls,h)
```

Fig. 5: A separation kernel with two domains, `H` and `L`, written in HASK. Threads communicate via asynchronous broadcast & receive system calls. Note that messages broadcasted on `L` reach `H` but not vice versa. The unit and bind operations, `return` and »=, are overloaded for monads `K`, `R`, and `Re`. The thread operators (`onsig`, `step`, `rspd`) are discussed in Section 4.1.

4. HASK Language Design & Implementation

There were a number of issues we confronted and design decisions we made in the design of HASK. HASK is a pure functional language like the Haskell language, albeit with a number of strong constraints. The original experiments with monad-based security kernels were rendered in the Haskell functional programming language, but, for a number of reasons relating to its complicated semantics and implementation, Haskell was determined to be an unsuitable source language. HASK has been made essentially simpler than Haskell by introducing and enforcing a number of restrictions on its expressiveness.

HASK is a first-order language, meaning that functions are never proper values as they are in a higher-order language like Haskell. The rationale behind this somewhat unusual choice (unusual, but not unprecedented [37]) is that we did not want to include language constructs that would necessitate complex implementation strategies. As the resumption monadic paradigm developed in previous publications are all essentially first-order, HASK's economy of expressiveness is not a shortcoming and ultimately facilitates our implementation and verification efforts. Implementations of HASK programs are kernels and require the same basic runtime structures and characteristics; these are bounded usage of memory, loop-structured control flow and asynchronous interrupt handling. HASK disallows general recursion in favor of tail recursion.

Monads are first-class in HASK by which it is that there is a declaration form "$\mathbf{monad}\ M = L_1 \oplus \cdots \oplus L_n$" in HASK for defining the monad DSL, M. The monad layer (transformer) expressions, L_i, determine which commands are present in M. Monad declarations of this form are an integral part of HASK syntax in that they define the operations of the Rushby monad in which a kernel may the be specified. This part of the language design is discussed in detail in a recent publication [35].

4.1 HASK by Example

This section gives an introduction to the HASK language via an a small kernel (see Fig. 5). The monad DSL for this kernel is constructed along the same lines as the monad DSLs from Section 3 in the upper left column. The address spaces, `Hi` and `Lo`, are maps from addresses to `Int`. The request type, `Req`, has requests for a broadcast message service as well as a request to receive a message. System responses, `Rsp`, has responses of the form `(Rcvd i)` which send the data `i` back to a process. The `Cont` and `Ack` request and response are NOPs. The languages of user processes (not shown) are defined as they are in Section 3.

HASK has a built-in queues captured as the type constructor, `Q`. Type of queues holding values of type `a` are written, `Q a`. The queue `Q a` has constructors for the empty queue and item insertion; these are `∅ :: Q a` and `(«) :: Q a -> a -> Q a`, resp. The head and tail of a queue `q` are accessed via pattern-matching within a **`case`** expression:

```
case q of
     ∅       -> e1
     (h◇hs) -> e2
```

Expression `e1` is only evaluated when `q` is empty. The head (`h`) and tail (`hs`) of `q` may be accessed within `e2`.

Using the queue abstraction, we define the kernel-level data, `Sys` (line 6 of Fig. 5). A `Sys` value is a tuple, `(ts,l,h)`, consisting of a ready thread queue (`ts`), a `Lo`-domain message queue (`l`), and a `Hi`-domain message queue, (`h`). The handling of these message queues is critical to the kernel maintaining the domain separation policy.

The kernel itself is encapsulated by the mutually tail-recursive functions, `kl :: Sys -> R ()` and `disp :: HdlrOp HandShake -> R ()`. The `kl` picks a process `t` from the queue, if it exists, and runs it by passing it to `disp`. The dispatch function, `disp`, processes any request signal, executes the slice, and passes control back to `kl`. The security of the kernel is maintained by `disp`. If a high process broadcasts a message, it only affects the high security message queue. If a low security process requests a message, then it only receives those from the low security queue. The kernel is the only place where insecure flows could arise in this HASK kernel, so this kernel is secure. Verification of similar kernels is found in the references [34].

4.2 HASK Language Implementation

There are two implementation strategies for monadic models of take separation [38]. The first compiles models into three address code using a traditional compiler approach. The second automatically transforms models into abstract state machines via a program transformation known as defunctionalization [39].

The high-level architecture of a monad compiler is conventional, consisting of an intermediate code generation phase translating the source program (i.e., a program written in monadic style) to an appropriate intermediate representation (IR) and a phase translating the IR into an appropriate instruction set. Traditional language compilers also include considerable analysis and optimization in the compiler "back-end", which our monad compiler prototype currently does not perform.

Aside from the source language, the novelty in monadic compilers lies both in the process by which code generation is performed and in the form of the IR itself. This process of code generation involves the application of a classic semantics-preserving program transformation called *defunctionalization* [39] to ICG. This transformation provides a means by which to compile higher order functions to state machines which, in turn, are readily compiled to hardware. These abstract state machines may then be translated into VHDL using FSMLang [40].

4.3 Extended Typed Interrupt Calculus

The techniques described in the preceding sections achieve security by construction: the resumption-monad kernel model inherently excludes storage channels between processes, except when such channels are actively mediated by the kernel. Stock hardware typically does not provide such strong guarantees. While a typical CPU architecture provides *mechanisms* to enforce information flow policies (in the form of an MMU), these mechanisms do not provide inherent *guarantees* of security: the operating system still may grant low-security processes access to high-security data, in violation of the security policy. We address this discrepancy by introducing an intermediate language called

```
 1  var channel : int := 0
 2
 3  handler 1 {
 4    if pid==0
 5      then switch(1)
 6      else switch(0)
 7    fi
 8  }
 9
10  handler trap {
11    channel := R[5] ; switch(pid)
12  }
```

Fig. 6: A Simple ETIC Kernel

the *Enhanced Typed Interrupt Calculus* (ETIC), patterned after the Typed Interrupt Calculus developed by Palsberg and Ma [41], [42]. ETIC may be viewed as a domain-specific language for OS kernels. It is a simple imperative language, enriched with primitives for interfacing with interrupts and hardware-enforced memory protection. It has a precise semantics, suitable for reasoning about security properties, yet may also be compiled to kernel-level code running on real hardware in a straightforward way.

4.3.1 Overview of ETIC

Figure 6 is a simple example of an ETIC kernel. Consider a machine consisting of one processor with a single interrupt request line (numbered 1), connected to a periodic timer. A pre-emptive multitasking kernel for such a machine may be implemented in ETIC simply by declaring a single interrupt handler for the timer, which invokes a scheduler terminating in a *switch* statement, and a trap handler for system calls. The kernel in Figure 6 assumes there are only two processes, numbered 0 and 1, and a single system call that allows a process to write a value (stored in register 5) to a single shared storage location.

A partial grammar for core ETIC is given in Figure 7. An ETIC kernel consists of a set of *handler* declarations, one for each interrupt request line and one for software traps. The language of commands consists of conditional statements, loops, assignment, sequencing, and three special primitives for manipulating the page table and executing a context switch to a user process.

4.3.2 Semantics

The semantics of ETIC is defined in a standard operational style in Figure 8; i.e. as a set of state transition rules. Figure 8 gives the most important rules. We assume that a semantics for single processes, whose state consists of registers R and memory M, is defined by a transition relation $\xrightarrow{U}$. On top of this we define a "machine-level" semantics, representing a machine with multiple user process and a kernel. Machine state is represented by a tuple consisting of an execution mode m (which is either U for user mode or K for kernel

$$
\begin{array}{rcl}
\textit{Prog} & ::= & \textit{Handler}^* \\
\textit{Handler} & ::= & \text{handler } \textit{IRQ} \ \{ \ \textit{Cmd} \ \} \\
\textit{IRQ} & ::= & 1 \mid 2 \mid \cdots \mid n \mid \text{trap} \\
\textit{Cmd} & ::= & \text{if } \textit{Expr} \text{ then } \textit{Cmd} \text{ else } \textit{Cmd} \\
& \mid & \textit{Lhs} := \textit{Expr} \\
& \mid & \textit{Cmd} \ ; \ \textit{Cmd} \\
& \mid & \text{map}(\textit{Expr}, \textit{Expr}, \textit{Expr}) \\
& \mid & \text{unmap}(\textit{Expr}, \textit{Expr}) \\
& \mid & \text{switch}(\textit{Expr}) \\
\textit{Expr} & ::= & \textit{Literal} \mid \textit{Expr Binop Expr} \mid \textit{Variable} \mid R_i \\
\textit{Lhs} & ::= & \textit{Variable} \mid R_i
\end{array}
$$

Fig. 7: Grammar for Core ETIC

mode), the kernel command stream C, internal kernel state S, saved process registers R, user-process heap M, saved process identifier P, page table T, and finally the code for the interrupt handlers H (the interrupt handler for IRQ i is denoted by H_i).

Each rule in Figure 8 consists of a (possibly vacuous) assumption above the horizontal line, and a conclusion below it. We "lift" the single-process semantics into the machine semantics via rule LIFT: essentially, the single-process semantics determines what the system will do when it is in user mode. Assume that R is the user process registers, $\hat{M}$ is process P's view of the machine memory in page table T, and applying the single-process transition relation to $\langle R, \hat{M} \rangle$ results in registers R' and memory $\hat{M}'$. Then when process P is running the system may transition to a new state with registers R' and memory M', where M' results from "merging" the changes from $\hat{M}$ to $\hat{M}'$ into M according to process P's view of the memory. Rule INTR expresses the semantics for interrupts: whenever a user process is running, the system may transition to kernel mode and begin executing the code for an interrupt handler. Note that more than one transition rule may apply to any given machine state; in particular, both INTR and LIFT apply anytime a user process is running. This possibilistic nondeterminism reflects the assumption that interrupts may occur (or not occur!) at any time.

The transition rules also determine the behavior of the commands defined in Figure 7. The rule SWITCH says that executing the switch command overwrites the value in the system process ID register, and returns to user mode. The map command allows the kernel to update the page table by adding a mapping of PID-page pairs to physical memory addresses. Note that the rules for these commands only apply when the system is in kernel mode. The rules for the other kernel commands are similar and so are omitted here.

5. Related Work

A calculational approach to program construction [43] starts from a high-level specification and produces an implementation through a series of verifiable transformations. The motivation behind the calculational approach is that the formal connections between source specifications and their implementations support formal verification. The calculational approach to compiler construction [44], [45], [46], [47] follows this paradigm by transforming a high-level language semantics into compiler implementation.

Monadic semantics have played a role in other recent high assurance systems research as well. State-monadic semantics have been used recently for specifying ARM and x86 instruction sets [48], [49]. The design, construction and verification of a secure microkernel is described by Klein et al. [50], [51], [52]. This kernel—called seL4—is a version of the L4 microkernel [53] with security guarantees. Monads are used in the modeling phase and are implemented by hand translation into C. Monads have also play a role in recent information security models [54], [55]. Monads can be used as a scoping mechanism for side effects and this is central to the authors' approach to secure system construction. The security model advocated by the authors uses structural (i.e., "by construction") properties of monads as a central organizing principle of their approach, whereas the aforementioned models use monads more as a security level tag.

Monadic programs are typically implemented in higher order functional languages although, in certain cases, the notion of computation encapsulated by a monad could be more efficiently implemented directly. Monads of resumptions and state can be implemented efficiently via translation into simple imperative loop code [38]. The present approach relies on *monad compilation* to produce code rather than partial evaluation (as does pass separation). Monad compilation treats terms typed in a particular monad as a programming language unto itself. The syntactic monad-as-language treatment is not new: Hughes applied it in the development of a pretty-printing library [56] as did Hinze [57] in the derivation of backtracking monad transformers. Direct compilation of monads has also been explored by Danvy et al. [58] and Filinski [59].

Model-driven development (MDD) [60], [61] is a software engineering paradigm that has sparked considerable interest of late. Semantics-directed compilation is a form of MDD that has a long history within programming languages research. In the present work, the "models" are constructed with monad transformers. Such models have a dual nature: they are both denotational models (supporting formal reasoning) and operational models that may be executed.

Channel theory is closely related to Chu spaces, information systems and institutions. In the categorical view of channel theory, the objects are the same as those in Chu spaces [62], but the arrows are different. In Chu spaces, no

$$\frac{\langle R, \hat{M}\rangle \xrightarrow{U} \langle R', \hat{M}'\rangle \quad \hat{M} = \pi_{P,T}(M) \quad M' = \iota_{P,T}(M, \hat{M}, \hat{M}')}{\langle U, C, S, R, M, P, T, H\rangle \to \langle U, C, S, R', M', P, T, H\rangle} \text{ (LIFT)}$$

$$\frac{}{\langle U, C, S, R, M, P, T, H\rangle \to \langle K, H_i, S, R, M, P, T, H\rangle} \text{ (INTR)}$$

$$\frac{}{\langle K, \text{switch}(e) :: C, S, R, M, P, T, H\rangle \to \langle U, C, S, R, M, [\![e]\!]_S, T, H\rangle} \text{ (SWITCH)}$$

$$\frac{}{\langle K, \text{map}(e_1, e_2, e_3) :: C, S, R, M, P, T, H\rangle \to \langle K, C, S, R, M, P, T[\langle [\![e_1]\!]_S, [\![e_2]\!]_S\rangle \mapsto [\![e_3]\!]_S], H\rangle} \text{ (MAP)}$$

Fig. 8: Core ETIC Semantics

mention is made of the theory of a classification and most of the research appears to be directed at their categorical structure. By comparison, the categorical structure in channel theory, while useful, is not of primary importance. Scott's information systems [63] use similar structures specialized to computational domains. There is an extensive literature on institutions [64]. The aforementioned reference combines institutions with the work of Barwise and Seligman. An institution is a functor from an abstract category to the category of classifications.

6. Conclusion

To construct an embedded kernel using current technologies, how would you proceed? You would choose a target architecture for which one would presumably also have a C compiler and an assembler. Assume that the functionality that the kernel is to support is also known and has been provided to you. Let us say further that the kernel is to enforce some security policy like domain separation. With enough effort, you produce a kernel implementation in C with some additional assembly language routines. Now, the kernel is stress tested enough that you have become fairly confident that it works as intended. Now, you present this kernel to your boss, who demands that you justify your confidence that the kernel is indeed secure.

Your boss asks, "how do you know that high security information doesn't leak?" You answer, "Here are the lines of C in the kernel that enforce security." Puzzled and mildly irritated to have to look at code, your boss continues, "OK, all that gobble-de-gook code seems reasonable, but how do you know that the C compiler doesn't mess it all up somehow?" He never brought this up before, you think to yourself. The C compiler you used has over 5 million lines of code, so trying to validate that it doesn't somehow undermine your security mechanisms is not tractable. "Also," your boss continues, "there are all of these insidious code injection attacks with the architecture you chose, how does your kernel cope with them?" Come to think of it, precisely what did you need from the underlying hardware to build a secure system?

Like it or not, your boss's questions raise important issues. What sort of support do you need from hardware to build a secure system? Even if a software system is deemed secure, what are the underlying assumptions about the hardware that support this claim?

Security is an end-to-end concern. Even if we verify that a program obeys a security policy, there are still parts of the underlying system—e.g., the compiler, operating system and hardware—that could undermine our program's security when executed and this underlying system is almost certainly not built with your security policy in mind. Security must be taken into account through all phases of the system design and implementation—especially when the system security is to be formally verified. But, current toolchains are not designed with this end-to-end flow in mind, and no amount of pounding will drive the security square peg into the round hole of current methodologies. A true science of security will require radical intellectual retooling of how we design, construct and verify systems.

Functional languages are often heralded as a good foundation for high assurance computation, and we agree with this—as far as it goes. Languages like ML and Haskell do contain core languages that have been given rigorous semantics and this is critical to high assurance and formal verification. For certain applications and certain required levels of assurance, these languages may indeed provide sufficient frameworks for designing, verifying and construction of high assurance systems. However, there are applications—security microkernels in particular—where off-the-shelf general purpose functional languages do not suffice. For one thing, their semantically specified core languages do not tackle the Awkward Squad and this, in itself, makes high assurance development of system-level code problematic at best. For another, their compilers and run-time systems were not designed to be verified and, as a result, are every bit as difficult to verify as any other large compiler. Finally, their implementations have behaviors and functionalities that can also be problematic for system programming; for example, their executables may be too large for embedded systems and garbage collection can create space leaks and unpredictable timing behavior.

HASK retains only the parts of Haskell that we need while jettisoning the rest. Strong typing, pattern-matching, and functions are all beneficial. The Haskell type system

can be extended or refined with security types [65] and this approach can serve to simplify the verification process. We have specified much of the HASK language and its infrastructure in the Coq theorem proving system [66] and are currently investigating machine-checked verification of HASK. Following the domain-specific language design approach with HASK is key to making HASK verification tractable.

What most distinguishes this research from that of others applying functional languages [11], [12] is our programming model. While the HASK language is, in fact, functional, the design of the language follows from a consideration of the features strictly necessary to express essential computational effects that are otherwise typically considered the domain of low-level imperative languages such as C. Because monads provide a precise semantic basis for the "Awkward Squad", we choose them as the basis of our approach.

Acknowledgment

This research was supported by NSF CAREER Award 00017806, US Naval Research Laboratory Contract 1302-08-015S, and the Gilliom Cyber Security Gift Fund.

References

[1] J. Rushby, "Design and verification of secure systems," in *Proceedings of the ACM Symposium on Operating System Principles*, vol. 15, 1981, pp. 12–21.

[2] E. Moggi, "Notions of computation and monads," *Inf. Comput.*, vol. 93, no. 1, pp. 55–92, 1991.

[3] J. Barwise and J. Seligman, *Information Flow: The Logic of Distributed Systems*. Cambridge University Press, 1997, cambridge Tracts in Theoretical Computer Science 44.

[4] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (revised)*. Cambridge, MA: MIT Press, 1997.

[5] J. C. Mitchell and R. Harper, "The essence of ML," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '88, 1988, pp. 28–46.

[6] W. L. Harrison and R. B. Kieburtz, "The logic of demand in haskell," *Journal of Functional Programming*, vol. 15, no. 5, pp. 837–891, 2005.

[7] W. Harrison, "A simple semantics for polymorphic recursion," in *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS05)*, 2005, pp. 37–51.

[8] W. Harrison, T. Sheard, and J. Hook, "Fine control of demand in Haskell," in *6th International Conference on the Mathematics of Program Construction, Dagstuhl, Germany*, ser. Lecture Notes in Computer Science, vol. 2386. Springer-Verlag, 2002, pp. 68–93.

[9] S. Peyton Jones, "Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell," in *Engineering Theories of Software Construction*, ser. NATO Science Series. IOS Press, 2000, vol. III 180, pp. 47–96.

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

[11] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach, "A principled approach to operating system construction in Haskell," in *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP05)*. New York, NY, USA: ACM Press, 2005, pp. 116–128.

[12] P. Li and S. Zdancewic, "Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2007, pp. 189–199.

[13] E. Moggi, "An Abstract View of Programming Languages," Department of Computer Science, Edinburgh University, Tech. Rep. ECS-LFCS-90-113, 1990.

[14] M. Bishop, *Computer Security: Art and Science*. Addison-Wesley Professional, 2002.

[15] ——. (2011, Apr.) Computer security archives project. [Online]. Available: http://seclab.cs.ucdavis.edu/projects/history/

[16] Karl-Eugen Kurrer, *The History of the Theory of Structures: From Arch Analysis to Computational Mechanics*. John Wiley & Sons, 2008.

[17] T. Tredgold, *Elementary Principles of Carpentry; A Treatise on the Pressure and Equilibrium of Timber Framing; The Resistance of Timber; and the Construction of Floors, Roofs, Centres, Bridges, etc.*, 1820.

[18] D. L. Parnas, "Really rethinking 'formal methods'," *IEEE Computer*, vol. 43, pp. 28–34, 2010.

[19] R. W. Floyd, "Assigning Meanings to Programs," in *Proceedings of a Symposium on Applied Mathematics*, ser. Mathematical Aspects of Computer Science, J. T. Schwartz, Ed., vol. 19. American Mathematical Society, 1967, pp. 19–31.

[20] *Software Assurance*. IEEE Computer, September 2010.

[21] A. Hall, "Seven myths of formal methods," *IEEE Software*, vol. 7, pp. 11–19, September 1990.

[22] W. Gibbs, "Software's chronic crisis," *Scientific American*, pp. 86–95, Sep. 1994.

[23] J. P. Bowen and M. G. Hinchey, "Seven more myths of formal methods," *IEEE Software*, vol. 12, pp. 34–41, July 1995.

[24] G. J. Holzmann, "Proving the value of formal methods," in *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, 1995, pp. 385–396.

[25] M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T. Margaria, "Software engineering and formal methods," *Communications of the ACM*, vol. 51, pp. 54–59, September 2008.

[26] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of the 1982 IEEE Symposium on Security and Privacy*. IEEE Press, 1982, pp. 11–20.

[27] J. Rushby, "Noninterference, transitivity, and channel-control security policies," SRI International, Tech. Rep. CSL-92-02, December 1992.

[28] D. McCullough, "A hookup theorem for multilevel security," *IEEE Trans. Softw. Eng.*, vol. 16, no. 6, pp. 563–568, 1990.

[29] J. McLean, "A general theory of composition for a class of "possibilistic" properties," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 53–67, 1996.

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

[31] S. Zdancewic, "Challenges for information-flow security," in *Proceedings of the First International Workshop on Programming Language Interference and Dependence (PLID'04)*, 2004.

[32] G. Allwein and W. L. Harrison, "Partially-ordered modalities," in *Advances in Modal Logic*, vol. 8, 2010, pp. 1–21.

[33] E. Mendelson, *Introduction to Mathematical Logic*. Van Nostrand Reinhold Company, 1997.

[34] W. L. Harrison and J. Hook, "Achieving information flow security through monadic control of effects," *J. Comput. Secur.*, vol. 17, pp. 599–653, October 2009.

[35] P. Kariotis, A. Procter, and W. Harrison, "Making monads first-class with Template Haskell," in *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell (Haskell08)*, 2008, pp. 99–110.

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

[37] J. Goguen, "Higher-order functions considered unnecessary for higher-order programming," pp. 309–351, 1990.

[38] W. Harrison, A. Procter, J. Agron, G. Kimmel, and G. Allwein, "Model-driven engineering from modular monadic semantics: Implementation techniques targeting hardware and software," in *Proceedings of the IFIP Working Conference on Domain Specific Languages (DSL09)*. Berlin, Heidelberg: Springer-Verlag, July 2009, pp. 20–44.

[39] M. S. Ager, O. Danvy, and J. Midtgaard, "A functional correspondence between monadic evaluators and abstract machines for languages with computational effects." *Theor. Comput. Sci.*, vol. 342, no. 1, pp. 149–172, 2005.

[40] J. Agron, "Domain-specific language for hw/sw co-design for FPGAs," in *Proceedings of the IFIP Working Conference on Domain Specific Languages (DSL09)*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 262–284.

[41] J. Palsberg and D. Ma, "A typed interrupt calculus," in *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. London, UK: Springer-Verlag, 2002, pp. 291–310.

[42] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. A. Henzinger, and J. Palsberg, "Stack size analysis for interrupt-driven programs," *Inf. Comput.*, vol. 194, no. 2, pp. 144–174, 2004.

[43] R. Backhouse, *Program Construction: Calculating Implementations from Specifications*. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[44] A. Igarashi and M. Iwaki, "Deriving compilers and virtual machines for a multi-level language," in *Proc. of the 5th Asian conference on Programming languages and Systems*, 2007, pp. 206–221.

[45] E. Meijer, "Calculating compilers," Ph.D. dissertation, University of Nijmegen, 1992.

[46] G. Hutton and J. Wright, "Calculating an exceptional machine," in *Trends in Functional Programming*, H.-W. Loidl, Ed., Feb. 2006, vol. 5.

[47] O. Danvy, "A journey from interpreters to compilers and virtual machines," in *Proc. of the 2nd Intl. Conf. on Gener. Prog. and Comp. Eng.*, 2003, pp. 117–117.

[48] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, "The semantics of x86-cc multiprocessor machine code," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '09. New York, NY, USA: ACM, 2009, pp. 379–391.

[49] A. C. J. Fox and M. O. Myreen, "A trustworthy monadic formalization of the armv7 instruction set architecture," in *Interactive Theorem Proving (ITP)*, 2010, pp. 243–258.

[50] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an operating-system kernel," *Commun. ACM*, vol. 53, pp. 107–115, June 2010.

[51] 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 (TPHOLs'08)*, ser. Lecture Notes in Computer Science, O. A. Mohamed, C. M. noz, and S. Tahar, Eds., vol. 5170. Springer-Verlag, 2008, pp. 167–182.

[52] K. Elphinstone, G. Klein, and R. Kolanski, "Formalising a high-performance microkernel," in *Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 06), Microsoft Research Technical Report MSR-TR2006-117*, 2006, pp. 1–7.

[53] J. Liedtke, "On micro-kernel construction," in *Symposium on Operating System Principles*. ACM, 1995.

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

[55] K. Crary, A. Kliger, and F. Pfenning, "A monadic analysis of information flow security with mutable state," *Journal of Functional Programming*, vol. 15, no. 2, Mar. 2005.

[56] J. Hughes, "The Design of a Pretty-printing Library," in *Advanced Functional Programming*, ser. LNCS, vol. 925, 1995, pp. 53–96.

[57] R. Hinze, "Deriving backtracking monad transformers," in *International Conference on Functional Programming*, 2000, pp. 186–197. [Online]. Available: citeseer.nj.nec.com/hinze00deriving.html

[58] O. Danvy, J. Koslowski, and K. Malmkjaer, "Compiling monads," Kansas State University, Manhattan, Kansas, Tech. Rep. CIS-92-3, Dec. 91. [Online]. Available: ftp://ftp.diku.dk/pub/diku/semantics/papers/D-154.dvi.Z

[59] A. Filinski, "Representing layered monads," in *Proceedings of the 26st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM Press, 1999, pp. 175–188.

[60] D. Batory, "Program refactoring, program synthesis, and model-driven development." in *ETAPS Conference on Compiler Construction*, ser. LNCS, vol. 4420, April 2007, pp. 156–171.

[61] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, p. 25, 2006.

[62] M. Barr, "$*$-autonomous categories and linear logic," *Mathematical Structures Computer Science*, vol. 1, pp. 159–178, 1991.

[63] D. S. Scott, "Domains for denotational semantics," in *An extended version of the paper prepared for ICALP '82*. Springer–Verlag, 1982.

[64] J. Goguen, "Information integration in institutions," in *Thinking Logically: a Memorial Volume for Jon Barwise*, L. Moss, Ed. Indiana University Press, 200x.

[65] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, Jan. 2003.

[66] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. Springer Verlag, 2004.