

# All watched over by machines of loving grace

Dominic P. Mulligan ✉🏠

Automated Reasoning Group, Amazon Web Services, Cambridge, United Kingdom<sup>1</sup>

## Abstract

Modern operating systems are typically built around a trusted system component called the *kernel* which amongst other things is charged with enforcing system-wide security policies. Crucially, this component must be kept isolated from untrusted software at all times, which is facilitated by exploiting machine-oriented notions of separation: private memories, privilege levels, and similar.

Modern proof-assistants are typically built around a trusted system component called the *kernel* which is charged with enforcing system-wide soundness. Crucially, this component must be kept isolated from untrusted automation at all times, which is facilitated by exploiting programming-language notions of separation: module-private data structures, type-abstraction, and similar.

Whilst markedly different in purpose, in some essential ways operating system and proof-assistant kernels are tasked with the same job, namely enforcing system-wide invariants in the face of unbridled interaction with untrusted code. Yet the mechanisms through which the two types of kernel protect themselves are significantly different. In this paper, we introduce *Supervisory*, the kernel of a prototype programmable proof-checking system for Gordon’s HOL that is organised in a manner more reminiscent of operating systems than typical LCF-style proof-checkers. *Supervisory* implements a kernel that executes at a relative level of privilege compared to untrusted automation, with trusted and untrusted system components communicating across a limited system call boundary. Kernel objects, managed on behalf of user-space by the *Supervisory* kernel, are referenced by handles which are passed back-and-forth by *Supervisory*’s system calls.

Unusually, *Supervisory* has no “metalanguage” in the LCF sense, as the language used to implement the kernel, and the language used to implement automation, need not be the same. *Any* programming language can be used to implement automation for *Supervisory*, providing the resulting binary respects the *Supervisory* kernel calling convention and binary interface, with no risk to system soundness. Further, we observe that *Supervisory* allows arbitrary programming languages to be endowed with facilities for proof-checking. Indeed, the handles that *Supervisory* uses to reference kernel objects under its management may be thought of as a form of *capability*, in the computer security sense. Moreover, these capabilities are extremely expressive, essentially capturing the full expressive power of HOL, and can potentially be used to enforce fine-grained correctness and security properties of programs at runtime.

**2012 ACM Subject Classification** Theory of computation → Higher order logic; Theory of computation → Automated reasoning; Theory of computation → Logic and verification; Software and its engineering → Operating systems

**Keywords and phrases** Proof assistant design, operating systems, HOL, LCF, *Supervisory*, system description, capabilities

**Digital Object Identifier** [10.4230/LIPIcs...](https://doi.org/10.4230/LIPIcs...)

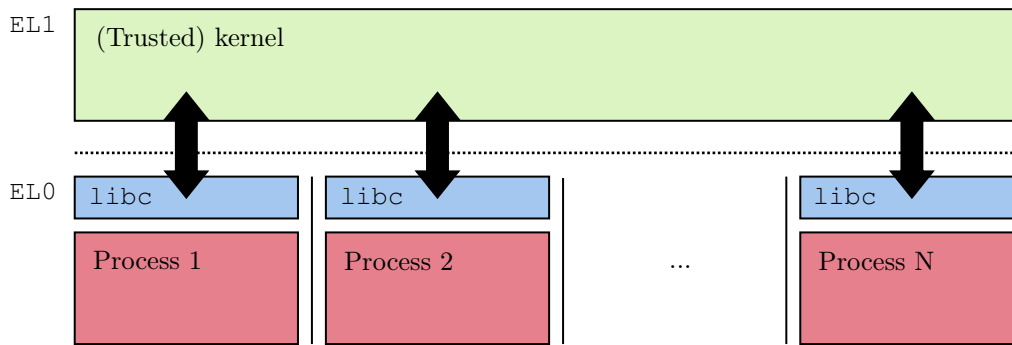
**Acknowledgements** We would like to thank Nick Spitale for many helpful conversations regarding *Supervisory*, and Nathan Chong for comments on an early version of this paper.

## 1 Introduction

This paper studies the intersection of operating system design and implementations of the foundations of mathematics. Research into the confluence of these two topics is, admittedly, a rather moribund affair at the moment. Nevertheless, with this paper we hope to convince

<sup>1</sup> All work done whilst employed within the Systems Research Group, Arm Research, Cambridge

## XX:2 Supervisory system description



■ **Figure 1** A schematic of the typical system organization of a commodity operating system and its associated user-space. The kernel (in green) executes at a relative level of privilege, enforced by hardware, compared to processes executing in user-space (red)—we follow the Arm convention and show the kernel executing at EL1 and user-space at EL0. The two communicate across a system call boundary (dashed line) using system calls (black arrows). User-space programs are typically written making use of an abstraction library, such as `libc` (blue), to abstract over this kernel interface.

45 the reader that probing the intersection of these two areas is potentially very interesting  
46 by introducing *Supervisory*, a programmable proof-checking system for Gordon’s HOL.  
47 This system has a novel system design, with some interesting properties, and moreover some  
48 interesting consequences. First, however, we begin with a scene-setting overview of common  
49 principles in operating system design and implementation.

### 50 1.1 On operating systems

51 Most commodity operating systems—that is, Microsoft Windows and Unix-derivatives<sup>2</sup>—fit  
52 a common pattern and are architected around a relatively self-contained, trusted component  
53 typically called the system *kernel*.

54 The kernel is the sole component that can interface unfettered with all system resources,  
55 including devices and other system hardware. Untrusted user-space applications make use of  
56 kernel interfaces in order to make use of a device or any other system resource managed by  
57 the kernel. As a result, the kernel is essentially a “pinch point” for gating access to system  
58 resources. The kernel also introduces a process abstraction in user-space and is responsible  
59 for ensuring the confidentiality and integrity of processes from other, concurrently executing  
60 processes. The kernel is therefore *the* key component responsible for enforcing system-wide  
61 security policies, and essentially forms the “root of all trust” within a computing system. It  
62 is therefore imperative that the kernel is itself isolated sufficiently from user-space software  
63 at all times, lest this role be undermined by a malefactor.

64 The kernel self-isolates by entering into a grand conspiracy with its host hardware. In  
65 support of this conspiracy, mainstream microprocessors have, over the years, accreted a  
66 variety of now-familiar security features that an operating system kernel can use to defend  
67 itself from prying or interference. These include *exception levels* or *privilege rings*—as they  
68 are variously called, depending on the instruction set architecture—which introduce a notion  
69 of *privilege* into the system. Here, software executing at higher-privilege—in our case, an

---

<sup>2</sup> *Commodity* here is used to guard against pedantic quibbling over research operating system designs—like exokernels [5] and other oddities—which arguably do not fit this pattern.

operating system kernel<sup>3</sup>—gains permission to program sensitive system registers, controlling how the system operates. Moreover, software executing at a higher-level of privilege can “peer in” and potentially modify the runtime state of software executing at a relatively lower-level of privilege, reading data from, or writing data to, a buffer within the memory space of an untrusted user-space process, for example. In this sense, a kernel can “supervise” or “watch over” untrusted user-space—essentially the inspiration for the, possibly now archaic, alternative phrasing for an operating system kernel: *supervisor*.

Modern microprocessors also provide a form of memory management built around page tables (see e.g. [2]). These data structures have a dual role: primarily, they are used for the virtualisation of system memory via address translation, granting user-space software the illusion that it owns the entire physical address space of the machine, presenting a virtual address space to user-space programs. This process induces a notion of *ownership* of pages of physical memory within the system, with a page of physical memory “owned” by a principal (either the operating system, or a user-space process) if it is “mapped in” to that principal’s address space. Moreover, page tables are also used for storing the attributes of pages of memory, including read-write-execute permissions. By correctly initialising and managing these tables the kernel is able to keep its own code and data structures isolated—in a kernel-private memory area—that only it can access, safe from prying or interference by untrusted user-space. As a result, for systems software on modern machines, isolation is enforced by a mix of low-level machine mechanisms: separate address spaces, private memory regions, and machine-enforced privilege checks on executing software.

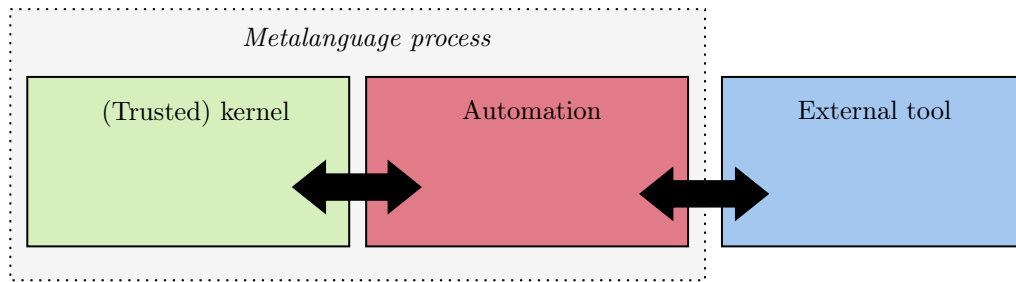
To make itself useful, the kernel exposes a limited interface, used by user-space to request intercession by the kernel on its behalf—for example by granting user-space access to some device, the filesystem, a socket, or some other system resource under kernel management. Dealing in generalities, to do this, the kernel exposes a suite of *system calls* which can be invoked by user-space programs with dedicated machine instructions provided by the microprocessor—see Figure 1 for a diagrammatic schematic, for example. On Arm platforms, with which the author is most familiar, these instructions induce a processor exception, forcing a *context switch* which flips the flow of control into the kernel’s system call handler, before eventually returning the flow of control back to the calling user-space program. From user-space’s point-of-view, system calls therefore have the appearance and effect of very CISC-like machine instructions, with the operating system kernel essentially presenting itself to user-space as “silicon by other means”, extending the user fragment of the instruction set architecture of the microprocessor with new instructions.

Note that for this two-way dance to work, user-space and the kernel must work together by adopting a series of joint conventions. These include a *calling convention* describing how arguments and results are passed back-and-forth across the system call interface, and a *binary interface* detailing how system calls are identified, how errors are reported back to user-space, and other miscellanea.<sup>4</sup> To help programmers adhere to these conventions, the operating system typically provides an abstraction layer to user-space, which on Unix variants typically takes the form of the system’s C library, `libc`. Note that this is generally just a convenience, and user-space software can always invoke system calls directly if wanted by invoking the

---

<sup>3</sup> Note that “Cloud hosting” as a viable business proposition essentially rests on this trick being repeated again, with a hypervisor sat in a position of privilege compared to an operating system kernel—executing out of an even higher exception level—and enforcing separation betwixt operating system instances.

<sup>4</sup> For more detail on the role of the ABI, its other aspects, and its very real effects on the semantics of programs, see this [10] outrageously under-cited overview.



■ **Figure 2** A schematic of the system organisation of a typical LCF-style proof assistant. The trusted kernel (green) is linked against untrusted automation (red) existing within the same metalanguage process (dotted line) and communicate with each other using the kernel’s API (leftmost black arrow). External tools existing as separate processes (blue), must communicate with a shim layer written in the proof assistant’s metalanguage to access the kernel (rightmost black arrow).

112 correct machine instruction and adhering to the appropriate calling convention.<sup>5</sup>

113 However, crucially, it is *generally* not the case that the operating system kernel and  
 114 untrusted user-space applications must be written in the same programming language for  
 115 this all to work. In particular, whilst most operating system kernels are written in C, or a  
 116 C-language derivative, user-space programs can be written in a variety of languages, and are  
 117 also commonly composed of multiple libraries, written in different programming languages,  
 118 linked together. Despite this, all are able to make use of system resources exposed by the  
 119 kernel’s system call interface by ensuring that they adhere to the calling convention and  
 120 binary interface expected by the kernel. In this respect, for commodity operating systems,  
 121 the C-language may have prominence as a favoured language of system implementation, but  
 122 by-and-large it is not *special* or given an unduly prominent status by the kernel itself.

## 123 1.2 On programmable proof-checkers

124 Most modern proof-assistants—for example, systems in the wider HOL family [14, 8, 15],  
 125 Coq [9], Matita [3], NuPRL [1], and similar—fit a common pattern and are architected  
 126 around a relatively self-contained, trusted component typically called the system *kernel*.

127 The system kernel is the sole component that can authenticate claims as legitimate  
 128 theorems of the implemented logic. Untrusted automation, residing outside of the kernel,  
 129 must “drive” the kernel to derive a theorem on its behalf. The kernel is therefore *the*  
 130 component responsible for ensuring system-wide soundness, and represents the “root of all  
 131 trust” within the system. It is therefore imperative that the kernel is able to isolate itself  
 132 sufficiently from untrusted automation at all times. This kernel-centric method of system  
 133 organisation is known as *the LCF approach* after Milner’s eponymous system [6] which first  
 134 introduced it, and is now the most common way of organising proof-checking systems today.  
 135 See Figure 2 for a diagrammatic representation.

136 Most modern proof-assistants tend to be written in a “metalanguage” which serves as the  
 137 implementation language for both the kernel and the majority of the untrusted automation  
 138 that modern proof-assistants provide to users. This metalanguage is typically a strongly-

<sup>5</sup> This is the case on Linux, though does not hold universally on all Unix derivatives. For example Apple’s MacOS and some BSD Unix variants generally consider the programming interface of the system C library as the interface of the kernel, proper, in some cases preventing any user-space code other than the system’s `libc` library from invoking system calls directly, as a security mechanism.

typed functional programming language, for example an ML derivative such as OCaml or SML [13], which offer strong modularity and abstraction features. The kernel exploits these programming language features to hide its own data structures from untrusted automation and moreover exposes a carefully limited API for proof-construction and manipulation. Notably, in an LCF-style system, the *only* mechanism automation has for constructing an authenticated theorem is by using this API, with the inference rules of the logic exposed as a suite of “smart constructors” manipulating an abstract type of theorems. As a result, the kernel is therefore a “pinch point” for any proof-construction activity within the system.

Untrusted automation and the system kernel are linked together, and reside side-by-side in the same process when the proof-assistant is executed. As a result, system soundness ultimately rests on the soundness of the implementation metalanguage’s type-system—specifically its ability to correctly isolate module-private data structures—that is, its ability to correctly enforce type abstraction. Moreover, the system metalanguage is, in a sense, unique amongst all programming languages, in that it is the *only* language capable of interfacing directly with the kernel, which is, after all, “just” a module written in that language, like any other. Whilst an external tool, or automation written in another programming language, *can* interface with the kernel, it must do so indirectly, making use of a shim layer written in the system metalanguage.

### 1.3 Introducing the Supervisory system

In many respects, as the text above intimates, the role of the kernel in both an operating system and in a proof-assistant is, at least in an abstract sense, the same: both components must enforce system-wide invariants in the face of unbridled interaction with untrusted code; both components also act as the “root of all trust” for their respective systems; both components act as “pinch points” that untrusted code cannot help interact with, if it wishes to engage in some kernel-gated activity. Consequently, both type of kernel need to correctly isolate their data structures and runtime state from interference by untrusted code. However, the two mechanisms through which this self-isolation are enforced are different: for operating system kernels<sup>6</sup> self-isolation is enforced using machine-oriented mechanisms; for LCF-style proof-assistants, self-isolation is enforced using programming language-oriented mechanisms.

In this paper we introduce *Supervisory*, the kernel of a novel programmable proof-assistant for Gordon’s HOL.<sup>7</sup> Whilst detailed further in Section 3, we note here that Supervisory’s system design has more in common with the typical system organisation of an operating system than comparable implementations of HOL. Specifically, the Supervisory kernel executes at a relative level of privilege compared to untrusted automation, which can be thought of as executing as a process in something akin to Supervisory’s version of “user space”. The trusted kernel, and untrusted user space, communicate across a system call boundary which is carefully designed in order to maintain system soundness.

One consequence of this design is that the Supervisory kernel immediately takes on a different character to an LCF kernel. All of the paraphernalia of a typical HOL implementation—type-formers, types, constants, terms, and theorems—are managed as

<sup>6</sup> Barring unikernels, or library operating systems, like Mirage [11, 12]. If we are really pushing this analogy note that unikernels are in some respects quite similar to LCF-style proof-assistants in this regard, having their kernel linked with untrusted “user-space” and separated using programming language features like modules, rather than privilege and memory isolation.

<sup>7</sup> Many of the ideas presented henceforth are logic-independent. Though we have chosen to use HOL in our prototype, the ideas presented herein can be applied to a wide variety of other logics and type theories with relatively straightforward changes.

“kernel objects” kept safely under the management of the kernel itself, in kernel-private memory areas. These kernel objects are never exposed *directly* to user-space, rather, they are manipulated by the Supervisory kernel on user-space’s behalf. Handles—which can be thought of as pointers, pointing into Supervisory’s private memories—are used by a user-space process to identify kernel objects that the kernel should manipulate or query.

Notably, Supervisory is also not implemented in a typed functional programming language, as is typical of most programmable proof-assistants, but is rather implemented in the decidedly *unsafe* systems programming language, Rust. Note that this decision introduces no risk to system soundness, as Supervisory’s soundness ultimately rests on the continued separation of kernel-private data from Supervisory’s analogue of user-space—using privilege and private memories—and not on the type system of the implementation programming language. Moreover, as user-space and kernel communicate across a defined system call interface, untrusted user-space may also be written in *any* programming language capable of producing code that is binary-compatible with the Supervisory kernel. Supervisory therefore has no “metalanguage” in the LCF sense, but rather an implementation language, with automation potentially written in multiple languages—maybe even a mix.

For ease of implementation—and use!—we implement Supervisory as a WebAssembly [7] (or Wasm, henceforth) host. We extend a Wasm virtual machine with new system calls that perform a context switch into Supervisory—the *host*—which has its own memory isolated from the memory of the executing user-space Wasm process running under its supervision, and inaccessible to it. Note that this separation is only one way: the Supervisory kernel can “peer in” to the runtime state of a running Wasm process executing under its supervision and read from, or write to, its private memories. This decision means we may experiment with the fundamental ideas behind Supervisory—namely isolating the kernel using private memory areas, the split between kernel- and “user-space”, a kernel system call interface—without becoming bogged down in extraneous detail associated with the booting ceremony of a real machine. Moreover, we harness work on porting compiler and linker toolchains, allowing our “user space” to be written in any programming language with a toolchain capable of targeting Wasm.

Lastly, and more speculatively, Supervisory’s handles can be passed around a program, between different programs executing concurrently or sequentially under Supervisory’s management, or between the user-space program and the kernel. Whilst this property is not unique to Supervisory—values of the abstract type of theorems may also be passed around within any LCF-style system, for example—the objects which these handles denote need not be necessary truths of pure mathematics, but can be contingent truths, themselves *functions* of the runtime state of the program itself, or of the Supervisory kernel. This property *is* unique to Supervisory, as it rests on Supervisory’s dual status as a proof-assistant kernel, capable of generating theorems, and an extension of a general purpose virtual machine, capable of executing arbitrary programs. Here, Supervisory once again represents a “pinch point” that user-space cannot help pass through in order to have any sort of computational effect—other than heating the CPU—on the machine.

Whilst exploring this idea in full is future work, some ideas of how this idea could develop are discussed in Section 4. But quickly, here, consider one example where this may be useful: Supervisory could present an extended system call interface to user-space, including system calls for filesystem manipulation, socketed network communication, and all other services typical of an operating system. Prior to opening or modifying a file or other system resource on a program’s behalf, Supervisory could “challenge” the program to first produce a handle denoting a theorem whose statement expresses some instantiation of a prevailing security

$$\begin{array}{c}
\frac{r : \tau}{\Gamma \vdash r = r} \quad \frac{\Gamma \vdash r = s}{\Gamma \vdash s = r} \quad \frac{\Gamma \vdash r = s \quad \Gamma' \vdash s = t}{\Gamma \cup \Gamma' \vdash r = t} \quad \frac{\phi \in \Gamma}{\Gamma \vdash \phi} \quad \frac{\Gamma \vdash \perp \quad \phi : \text{bool}}{\Gamma \vdash \phi} \\
\\
\frac{\Gamma \vdash r = s \quad \Gamma' \vdash t = u}{\Gamma \cup \Gamma' \vdash r \ t = s \ u} \quad \frac{\Gamma \vdash r = s \quad x_\tau \notin fv(\Gamma)}{\Gamma \vdash \lambda x_\tau. r = \lambda x_\tau. s} \quad \frac{}{\Gamma \vdash \top} \\
\\
\frac{\Gamma \vdash \phi \quad \Gamma' \vdash \psi}{\Gamma \cup \Gamma' \vdash \phi \wedge \psi} \quad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \quad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi} \quad \frac{\Gamma \cup \{\phi\} \vdash \psi \quad \phi : \text{bool}}{\Gamma \vdash \phi \longrightarrow \psi} \\
\\
\frac{\Gamma \vdash \phi \longrightarrow \psi \quad \Gamma' \vdash \phi}{\Gamma \cup \Gamma' \vdash \psi} \quad \frac{\Gamma \vdash \phi \quad \psi : \text{bool}}{\Gamma \vdash \phi \vee \psi} \quad \frac{\Gamma \vdash \psi \quad \phi : \text{bool}}{\Gamma \vdash \phi \vee \psi} \quad \frac{\Gamma \vdash \phi = \psi \quad \Gamma' \vdash \phi}{\Gamma \cup \Gamma' \vdash \psi} \\
\\
\frac{\Gamma \vdash \phi \vee \psi \quad \Gamma' \cup \{\phi\} \vdash \xi \quad \Gamma'' \cup \{\psi\} \vdash \xi}{\Gamma \cup \Gamma' \cup \Gamma'' \vdash \xi} \quad \frac{\Gamma \vdash \phi \longrightarrow \psi \quad \Gamma' \vdash \psi \longrightarrow \phi}{\Gamma \cup \Gamma' \vdash \phi = \psi} \\
\\
\frac{\Gamma \vdash \exists x_\tau. \phi \quad \Gamma \cup \{\phi[x_\tau := y_\tau]\} \vdash \psi \quad y_\tau \notin fv(\psi) \cup fv(\Gamma) \cup \{x_\tau\}}{\Gamma \vdash \psi} \quad \frac{\Gamma \vdash \phi = \psi \quad \Gamma' \vdash \psi}{\Gamma \cup \Gamma' \vdash \phi} \\
\\
\frac{\Gamma \cup \{\phi\} \vdash \perp \quad \phi : \text{bool}}{\Gamma \vdash \neg \phi} \quad \frac{\Gamma \vdash \neg \phi \quad \Gamma' \vdash \phi}{\Gamma \cup \Gamma' \vdash \perp} \quad \frac{\Gamma \vdash \forall x_\tau. \phi \quad r : \tau}{\Gamma \vdash \phi[x_\tau := r]} \\
\\
\frac{\Gamma \vdash \phi[x_\tau := r]}{\Gamma \vdash \exists x_\tau. \phi} \quad \frac{\Gamma \vdash \phi \quad x_\tau \notin fv(\Gamma)}{\Gamma \vdash \forall x_\tau. \phi} \quad \frac{s : \tau' \quad r : \tau}{\Gamma \vdash (\lambda x_\tau. s) r = s[x_\tau := r]} \quad \frac{\Gamma \vdash \exists x_\tau. \phi}{\Gamma \vdash \phi(\epsilon x_\tau. \phi)} \\
\\
\frac{f : \tau \Rightarrow \tau' \quad x_\tau \notin fv(f)}{\Gamma \vdash \lambda x_\tau. (f \ x) = f} \quad \frac{\Gamma \vdash \phi \quad r : \tau}{\Gamma[x_\tau := r] \vdash \phi[x_\tau := r]} \quad \frac{\Gamma \vdash \phi}{\Gamma[\alpha := \tau] \vdash \phi[\alpha := \tau]}
\end{array}$$

Figure 3 The Natural Deduction relation for Gordon's HOL.

or correctness policy. In particular, the statement of these challenges can be parametric in the runtime state of the executing program, of the kernel, or of the arguments passed to the system call—a function of the filename of the file to be opened or modified, for example, or the content to be written to the file. Essentially now, Supervisory's handles are transformed from simple pointers denoting kernel objects into *capabilities*, in the information security sense of the word, and which can be used by the system to dynamically enforce arbitrary security or correctness properties, written in HOL, of running programs. With this, the distinction between static verification and runtime monitoring of programs is blurred.

## 2 Implemented logic

Supervisory implements a variant of Gordon's HOL, a classical higher-order logic which can be intuitively understood as Church's Simple Theory of Types [4] extended with ML-style top-level polymorphism. We introduce the basics of this logic here, introducing just enough material so that the unfamiliar reader can follow the rest of the paper.

We fix a denumerable set of *type variables* and use  $\alpha, \beta, \gamma$ , and so on, to range arbitrarily over them. We work with *simple types* generated by the following recursive grammar:

$$\tau, \tau', \tau'' ::= \alpha \mid f(\tau, \dots, \tau')$$



Here  $f$  is a *type-former* which has an associated *arity*—a natural number indicating the number of type arguments that it expects. If all type-formers within a type are applied to a number of types matching their arity we call the type *well-formed*—that is, arities introduce a trivial or degenerate form of *kinding* for types. We will only ever work with well-formed types in Supervisory. We write  $tv(\tau)$  for the *set of type-variables* appearing within a type, and write  $\tau[\alpha := \tau']$  for the *type substitution* replacing all occurrences of  $\alpha$  with  $\tau'$  in the type  $\tau$ . From the outset we assume two primitive type-formers built-in to the logic itself and necessary to bootstrap the rest of the logic: **bool**, the type-former of the Boolean type—and also the type of propositions—with arity 0, and  $- \Rightarrow -$ , the type-former of the HOL function space, with arity 2. Note we will abuse syntax and also write **bool** for the *type* of Booleans and propositions, and also write  $\tau \Rightarrow \tau'$  for the function space type.

For each well-formed type  $\tau$  we assume a countably infinite set of *variables* and *constant symbols*. We use  $x_\tau, y_\tau, z_\tau$ , and so on, to range over the variables associated with type  $\tau$ , and use  $C_\tau, D_\tau, E_\tau$ , and so on, to also range over the constants associated with type  $\tau$ . With these, we recursively define *terms* of the explicitly-typed  $\lambda$ -calculus, as follows:

$$r, s, t ::= x_\tau \mid C_\tau \mid rs \mid \lambda x:\tau. r$$

Note that there is an “obvious” simple-typing relation on terms, which we omit here, though we write  $r : \tau$  to assert that term  $r$  has type  $\tau$  according to this typing relation. We call any term with a type *well-typed*, and we will only ever work with well-typed terms in Supervisory. Further, we call a term with type **bool** a *formula* and use  $\phi, \psi, \xi$ , and so on, to suggestively range over terms that should be understood as being formulae in the rest of the paper. We work with terms identified up-to  $\alpha$ -equivalence, write  $fv(r)$  for the set of *free variables* of the term  $r$ , write  $r[x_\tau := t]$  for the usual *capture-avoiding substitution* on terms, and write  $r[\alpha := \tau]$  for the recursive extension of the type substitution action to terms.

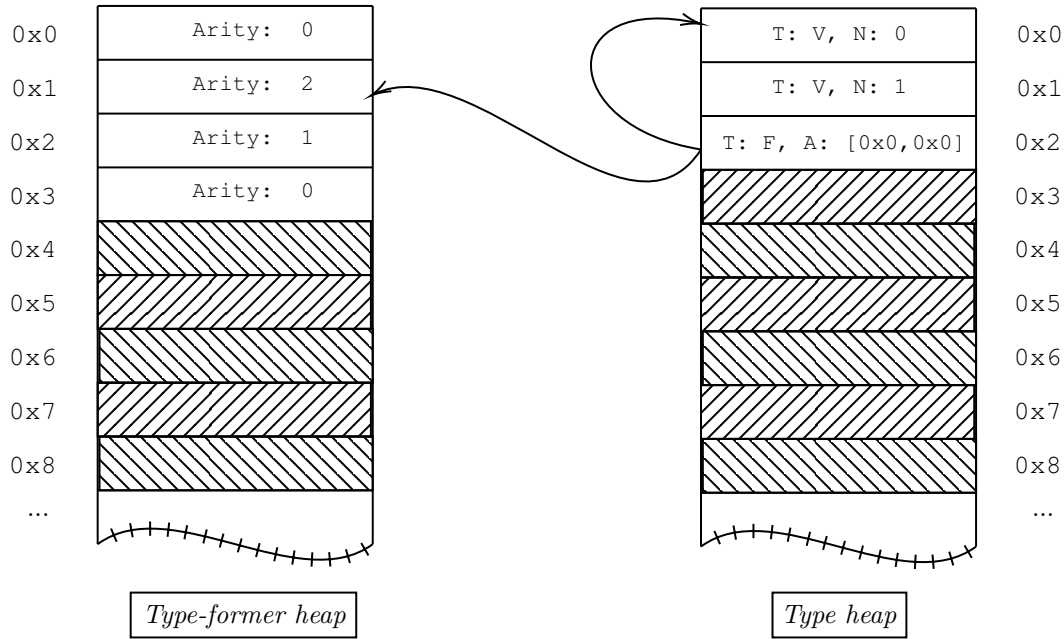
Like with type-formers, from the offset we assume a collection of typed constants needed to bootstrap the rest of the logic, summarised in the table below:

	$=$	$\alpha \Rightarrow \alpha \Rightarrow \mathbf{bool}$
	$\top, \perp$	<b>bool</b>
	$\neg$	<b>bool</b> $\Rightarrow$ <b>bool</b>
270	$\wedge, \vee, \longrightarrow$	<i>with type</i> <b>bool</b> $\Rightarrow$ <b>bool</b> $\Rightarrow$ <b>bool</b>
	$\forall, \exists$	$(\alpha \Rightarrow \mathbf{bool}) \Rightarrow \mathbf{bool}$
	$\epsilon$	$(\alpha \Rightarrow \mathbf{bool}) \Rightarrow \alpha$

Most of the constant above are the familiar logical constants and connectives of first-order logic, lifted into our higher-order setting, and are introduced without further explanation. Only the  $\epsilon$  constant—Hilbert’s *description operator*, a form of choice—may be unfamiliar. In HOL, this can be used to “select”, or “choose” an element of a type according to some predicate, and is otherwise undefined if no such element exists. Note therefore that all HOL types are inhabited by at least one element, with the term  $\epsilon x_\tau. \perp$  inhabiting every type. We adopt conventional associativity, fixity, and precedence levels when rendering terms using these constants, writing  $\phi \longrightarrow \psi$  instead of  $(\longrightarrow \phi)\psi$ , for example, and also suppress explicit type substitutions required to make terms involving polymorphic types well-typed, for example writing  $\forall x_\tau. \phi$  instead of  $\forall[\alpha := \tau](\lambda x_\tau. \phi)$ .

We call a finite set of formulae a *context*, ranged arbitrarily over by  $\Gamma, \Gamma', \Gamma''$ , and so on. We write  $\Gamma[x_\tau := r]$  and  $\Gamma[\alpha := \tau]$  for the pointwise-lifting of the capture-avoiding substitution and type substitution on terms to contexts, and write  $fv(\Gamma)$  for the set  $\bigcup\{fv(r) \mid r \in \Gamma\}$ . We introduce a two-place *Natural Deduction relation* between contexts and formulae using





**Figure 4** Entries within the Supervisory kernel’s type heap referencing entries within the type-former heap. Cross-hatched heap cells are as-yet unallocated by the kernel. The cell allocated at address 0x2 in the type heap is tagged with the F tag, indicating it is a type-former applied to a list of argument types, and points-to the cell at address 0x1 in the type-former heap, with arity 2. Two copies of the type stored in the cell with address 0x0, containing a type-variable with name 0, are used as the argument of the type-former to produce a complete, well-formed type. Adopting the convention that type-variable  $\alpha$  is at 0x0 in the type heap, and the function-space type-former  $\Rightarrow$  is at 0x1 in the type-former heap, then this represents an encoding of the type  $\alpha \Rightarrow \alpha$ .

the rules in Figure 3, and write  $\Gamma \vdash \phi$  to assert that a derivation tree rooted at  $\Gamma \vdash \phi$  and constructed according to the rules presented in this figure exists.

Note that our Natural Deduction relation can be simplified following the equational treatment of the quantifiers and connectives discovered by Quine and Henkin, and implemented in the HOL Light proof assistant. We prefer a more explicit treatment closer to a textbook presentation of Natural Deduction.

### 3 The Supervisory kernel state

Supervisory’s kernel manages a series of *heaps*, or private memories, in addition to other bits of book-keeping data. These heaps contain *kernel objects*, of various kinds: type-formers, types, constants, terms, and theorems. These follow the progression of the different kinds of HOL object, discussed in Section 2, and their interdependencies, and are discussed in turn.

#### 3.1 The type-former heap

The most foundational of all of the heaps is the heap of type-formers, which is manipulated and queried using a series of dedicated system calls. Each cell within the heap is either *unallocated* or *allocated* and, in the latter case, contains a natural number *arity* for a type-former, encoded as an unsigned 64-bit machine word. New type-formers are registered within the heap by invoking a dedicated system call from user-space—`TypeFormer.Register`—which takes as

## XX:10 Supervisory system description

input the arity of the type-former and in response allocates a fresh cell, returning the address of the cell back to user-space as the output of the system call. This address is the handle to the new type-former kernel object, now under management by the Supervisory kernel, and must be used by user-space to refer to this object henceforth. For example, a handle can be passed to the system call `TypeFormer.IsRegistered` system call to test whether a handle denotes a registered type-former. Alternatively, the `TypeFormer.Resolve` system call can be used to *dereference* a handle, in order to obtain an arity, providing that it does indeed denote a registered type-former, otherwise returning a defined error code.

Note that type-formers are essentially “named” by their handle: there may be many type-formers with the same arity registered with the kernel, and the particular meaning of any type-former is largely a convention of user-space, outside of the purview of the Supervisory kernel. However, there are exceptions to this rule. Two primitive type-formers are pre-registered within the type-former heap on system boot and hold special significance for the kernel. These are the `bool` type-former, registered at address `0x0` with arity 0, and the function-space type-former  $\Rightarrow$ , registered at address `0x1`, with arity 2. The existence of these pre-registered type-formers must be understood by user-space, and essentially forms part of the Supervisory system interface. Note that this is similar to how the distinguished file handles `stdout` and `stdin` are part of the POSIX system interface, and their hard-coded values must be understood by user-space to write or read from standard output and input.

### 3.2 The type heap

Building atop the heap of type-formers is the heap of types. This heap is queried and manipulated using another series of system calls. Note that the interface for working with types is much more complex than that for type-formers, so is only summarised here.

Recalling the grammar of HOL types, introduced in Section 2, we note that types are either a type-variable or a *combination* of a type-former applied to a list of types, and all entries within the type heap are therefore tagged indicating whether they are a type-variable or a combination. Type-variable entries only contain one datum: the *name* of the type-variable, which we take to be an unsigned 64-bit machine word. On the other hand, cells tagged with the combination tag also contain a pointer into the type-former heap, indicating which type-former is being applied, and contain a list of pointers back into the type heap itself, identifying the type arguments of the combination. Figure 4 shows a schematic diagram of dependencies between cells within the two heaps, wherein we use *V* to tag type-variables and *F* to tag combinations.

Like the type-former heap, Supervisory also boots with some entries in the type heap pre-registered, corresponding to common or useful types used to bootstrap the rest of the logic. These include the Boolean type, `bool`, common type variables— $\alpha$  and  $\beta$ , for example—as well as larger, more complex types such as the type of the polymorphic equality,  $\alpha \Rightarrow \alpha \Rightarrow \text{bool}$ . Again, the handles for all of these pre-registered entities must be understood by user-space.

Further derived types, built from primitive objects or otherwise, may be built using `Type.Register.Variable` and `Type.Register.Combination` system calls for constructing basic types. The first takes as input only a 64-bit machine word—the name of the variable—and immediately registers a new type in the type heap, returning the newly-allocated handle. On the other hand, `Type.Register.Combination` takes as input a handle pointing to a registered type-former in the type-former heap and a list of handles pointing back into the type heap. The system call fails if any of these handles dangle, or denote an object of the wrong kind, or if a list of type handles is presented with a length differing from the registered arity of the type-former. Lists of handles are passed to system calls by passing a base pointer,

denoting the beginning of the list (or rather, array) with an explicit length. Substitutions, for the `Type.Substitution` system call, which performs a type-substitution, are passed as two lists: one for the domain of the substitution, another for the range.

It is sometimes convenient to test the structure of a type pointed-to by a handle. This can be done using system calls like `Type.Test.Combination` which takes a handle and returns a Boolean value indicating whether the corresponding type is a combination. A family of “splitting” system calls—`Type.Split.Variable`, for example—can also be used to deconstruct a type. This takes a handle and returns the name of the variable pointed-to by the handle, if it is indeed a type-variable. Similar functions also exist for type combinations, and allow user-space to “pattern match” on types.

A system call, `Type.Variables`, also exists for computing the type-variables appearing within a term. Implementing this as a system call is a challenge, as the number of variables to be returned—and hence the size of buffer that user-space needs to set aside to hold them, and which Supervisory will write into—is unpredictable. To resolve this, the kernel exposes another system call, `Type.Size`, which computes the *size* of a type which bounds the number of variables appearing within a type. By querying this, user-space can first allocate sufficient memory within its own address space to hold the set of type-variables before calling `Type.Variables` with a pointer to the base of the allocated buffer.

Obviously, the Supervisory kernel must be careful in its management of its heaps, and this topic becomes pressing now we have introduced two heaps with dependencies between them. In particular, Supervisory maintains a series of *kernel invariants* which hold immediately out of boot and must be preserved by all system calls. One key invariant is the idea that heaps only ever *grow* monotonically, and allocated entries are immutable. Once an object is allocated into the heap it cannot be removed or modified in any way, lest we introduce an unsoundness for example, by modifying the `bool` type, or the truth constant,  $\top$ , or something similarly catastrophic. Moreover, heaps should always remain “inductive”, in the sense that their cells do not contain any dangling pointers that do not point-to allocated cells in the same or other heaps. Essentially, this latter property forces the various objects under Supervisory’s management to correctly follow the grammar of types and terms introduced in Section 2, with larger objects being gradually “built up” out of smaller ones.

### 3.3 The constant and term heap

Building on the heap of types is the heap of constants, keeping track of registered term constants. Again, this is pre-provisioned with a series of primitive constants, corresponding to the logical constants and connectives, at boot-time. The system call interface for constants is similar to that for type-formers, exposing just three system calls for registering new constants, dereferencing handles, and testing whether a handle denotes a registered constant.

Another, further heap—the heap of terms—is also used to construct and manipulate terms, with heap cells tagged with whether they represent a variable, constant, application, or lambda-abstraction, in a similar style to the tagging used for cells in the type heap. System calls for constructing, testing, and pattern matching on terms are provided, similar to those previously discussed within the context of other heaps. Further, new special-purposes system calls, for example `Term.Type.Infer` allow user-space to infer the type of a registered term, if any, whilst `Term.Substitute` performs a capture-avoiding substitution on a term. Note that handles for terms actually denote  $\alpha$ -equivalence classes of terms—at present, we use a name-carrying syntax, but could implement this using De Bruijn indices or levels, leading to

394 a more efficient implementation.<sup>8</sup>

### 395 3.4 The theorem heap

396 The final, and most important heap maintained by the Supervisory kernel is the heap of  
 397 theorems. Every other Supervisory heap exists to support this heap, and Supervisory  
 398 considers a theorem proved only if it appears in this heap. Cells within the theorem heap  
 399 contain a *sequent*, consisting of an (ordered) set of handles of formulae, representing the  
 400 assumptions of the theorem, and a single handle for the conclusion.

401 Each inference rule of the HOL logic—as presented in Section 2—is implemented by a  
 402 system call, accepting a handle denoting a theorem for each premiss of the inference rule.

403 **TODO: finish**

### 404 3.5 Specifying kernel functions

405 Implementing and using the Supervisory kernel is an extended exercise in pointer manipu-  
 406 lation. To specify the behaviour of some of our kernel system calls, we therefore reach for an  
 407 existing tool used to specify pointer-manipulating programs: Separation Logic.

408 Working abstractly, we represent kernel object handles as elements of the set  $\mathbb{N}$  of natural  
 409 numbers. We use  $h, h', h''$ , and so on, to range over handles. For a fixed set  $A$ , we say that  
 410 a partial-function  $f : \mathbb{N} \rightarrow A$  is *finitely-supported* when the set  $\text{dom}(f) = \{x \mid f \text{ } x \text{ defined}\}$  is  
 411 finite. We call such a finitely-supported partial map an *A-heap*. We write 0 for an empty  
 412 *A-heap*, and for two *A-heaps*  $f$  and  $g$  we write  $f \# g$  to assert that their domains are disjoint,  
 413 so  $\text{dom}(f) \cap \text{dom}(g) = \{\}$ . This relation is symmetric and  $0 \# g$  always, for any  $g$ . Moreover,  
 414 for two *A-heaps*  $f$  and  $g$  we write  $f \oplus g$  for the function defined piecewise as:

$$\begin{aligned} 415 \quad (f \oplus g) \ x &= f \ x \text{ if } x \in \text{dom}(f) \\ 416 \quad (f \oplus g) \ x &= g \ x \text{ if } x \in \text{dom}(g) \\ 417 \quad (f \oplus g) \ x &\text{ is undefined otherwise} \end{aligned}$$

419 Note that  $f \oplus g$  is well-defined whenever  $f \# g$ . Finally, for  $a \in A$ , we write  $h \mapsto a$  for the  
 420 *singleton A-heap* mapping  $h$  to  $a$  and undefined at all other points.

421 Fix a set of kernel states  $K$ . We use  $k, k', k''$ , and so on, to range over kernel states,  
 422 each of which is a 5-tuple  $\langle F, Ty, C, Tm, Th \rangle$  consisting of a type-former heap, a type heap,  
 423 a constant heap, a term heap, and a theorem heap respectively. We write  $F \ k$  to project the  
 424 type-former heap out of a kernel state  $k$ , and analogously for other projection functions  $Ty$ ,  
 425  $C$ ,  $Tm$ , and  $Th$ . We extend our notion of disjointness to kernel states, and write  $k \# k'$  to  
 426 assert that all of the respective heaps in kernel states  $k$  and  $k'$  are disjoint. We further abuse  
 427 notation and write 0 for the *empty kernel state* consisting of five empty heaps, and  $k \oplus k'$  for  
 428 the “gluing” of two kernel states together, wherein each of the respective heaps in  $k$  and  $k'$   
 429 are joined using  $\oplus$ . Note that, again,  $k \oplus k'$  is well-defined whenever  $k \# k'$ .

430 We define *assertions* as sets of kernel states, use  $A, B, C$ , and so on to range over them  
 431 and write  $k \models A$  to assert that  $k \in A$ . We pay especial attention to some particular assertions:

$$\begin{aligned} 432 \quad \bullet &\equiv \{k \mid k = \langle 0, 0, 0, 0, 0 \rangle\} \quad A \star B \equiv \{k'' \mid \exists k. \exists k'. k'' = k \oplus k', k \# k', k \models A, k' \models B\} \\ 433 \quad h \mapsto_{\text{Arity}} a &\equiv \{ \langle h \mapsto a, 0, 0, 0, 0 \rangle \} \end{aligned}$$

<sup>8</sup> Naturally, this would come at the continued mental cost of actually having to use, or even think about, De Bruijn indices.

System calls  $e, f, g$ , and so on, are modelled as potentially-failing functions from kernel states to kernel states which also produce a value as a side-effect, that is  $e : K \multimap K$ . Note that though a kernel system call may fail—for example, if its inputs are in an unexpected form, or similar—it should never *crash*, but rather return a specific error code back to the user-space program and maintain the state of the kernel as it was before the system call was invoked. Crashes—or *kernel panics*—are reserved for unrecoverable errors, for example the failure of an internal invariant, or similar.

With this in mind, for a function  $e$  from we define a Separation Logic triple  $A \vdash e \dashv B$

•  $\vdash \text{TypeFormer.Register}(a) \dashv \lambda h. h \mapsto_{\text{Arity}} a$

$h \mapsto_{\text{Term}} t \vdash \text{Theorem.Register.Refl}(h) \dashv$

$\lambda r. \exists h'. h \mapsto_{\text{Term}} t \star h' \mapsto_{\text{Term}} h = h' \star r \mapsto_{\text{Theorem}} \{\} \vdash h'$

TODO: finish

### 3.6 Programming the kernel

The system call interface above presents a very low-level, austere interface to user-space code. To make programming above Supervisory less tedious a utility library, similar in function to `libc`, needs to be provided to user-space in order to raise the level of abstraction above the raw system call interface. In Supervisory, this is provided as `libsupervisory`, currently implemented only for the Rust programming language, but could in theory be ported to the C-language, or any other language that can be compiled to Wasm.

Note that further layers, built on top of `libsupervisory`, can provide pretty-printing and parsing routines for types and terms, automation, proof-state management, and other functions typical of a proof-assistant.

## 4 Capabilities on steroids

We now take a more speculative turn, discussing future work. The ideas presented in this section are perhaps the most interesting consequence of Supervisory’s design, and we therefore dedicate a section solely to them.

As described, Supervisory is a proof-checking system implemented in an unusual way, but also a virtual machine, capable of executing arbitrarily complex programs compiled to the Wasm instruction set, from a variety of source programming languages.

However, at present, these Wasm programs are limited in the *effects* that they can make on the system—specifically, the only effect that they can actually make, other than heating the CPU, is to construct types, terms, and theorems, in Supervisory’s various heaps, using the series of system calls progressively introduced in Section 3. Programs executing under Supervisory are so-far incapable of opening files on the user’s machine, communicating over sockets, or querying the system time, because Supervisory does not provide any system calls to allow a program to perform any of those activities.

However, it could. For example, we could implement a system interface that provided all of the system calls needed by “real” programs wishing to make some effect on a user’s machine. By doing this, Supervisory is transformed into a general-purpose virtual machine, akin to the Java Virtual Machine, capable of executing arbitrary programs—calculators,

## XX:14 Supervisory system description

477 simulations, file search utilities, and so on—albeit with a bizarre set of extra system calls  
478 dedicated to theorem proving. In short, by extending Supervisory with system calls for  
479 querying and manipulating the system state, Supervisory is *both* a proof-assistant and a  
480 general-purpose virtual machine—though these two facets of the system are kept separate,  
481 as two different families of system calls. But: what happens if we merge the two?

482 In particular, prior to allowing a user-space program to open or read a file, Supervisory  
483 could first demand that a (handle to a) theorem is supplied to it as an extra argument  
484 to the file-open system call, `fopen`, for example. Interestingly, because Supervisory  
485 executes at a relative level of privilege, and can “peer in” to the runtime state of a user-space  
486 program, the statement of this desired theorem can be a *function* of the runtime state  
487 of the user-space program itself, of the runtime state of the Supervisory kernel, and  
488 also of the various arguments and other details of the system call being invoked. This  
489 statement—which we will call the *challenge*—can be any arbitrary formula written in HOL,  
490 and can be generated dynamically by the kernel, perhaps in accordance with a global *policy*  
491 enforced by Supervisory. A failure to produce a handle to address a particular challenge  
492 causes the system call to fail, with a runtime failure.

493 For concreteness, suppose we fix HOL types `wstate`, `kstate`, and `cstate`, which you may  
494 imagine as being record types capturing details of the runtime state of the executing Wasm  
495 process, the runtime state of the Supervisory kernel, and the details of the system  
496 call being invoked. Supervisory can dynamically *reflect* the actual runtime states of  
497 the user-space program and kernel, and the invoked system call, into inhabitants of these  
498 HOL types. Then, supposing our prevailing security policy,  $p$ , is a HOL function of type  
499  $wstate \Rightarrow kstate \Rightarrow cstate \Rightarrow bool$ , a challenge is obtained by dynamically applying  $p$  to the  
500 reflected records, described above. Note that two special security policies exist:

$$501 \quad \lambda w_{wstate}. \lambda k_{kstate}. \lambda c_{cstate}. \perp \quad \text{and} \quad \lambda w_{wstate}. \lambda k_{kstate}. \lambda c_{cstate}. \top$$

502 When applied to a reflected runtime state, these two policies generate the challenges  $\perp$  and  
503  $\top$ , respectively. The first policy is therefore the *deny all* policy, which essentially prevents a  
504 user-space program from invoking *any* system call, and making any effect on the system state,  
505 whilst the second is the *allow all* policy which can always be trivially satisfied by passing  
506 the handle to HOL’s truth introduction theorem.<sup>9</sup> However, between these two extremal  
507 points are a variety of other interesting policies. For example, if we assume that our `cstate`  
508 record contains a field `cname` of type  $cstate \Rightarrow string$  capturing the name of the system call  
509 being invoked, then we may selectively prevent particular system calls from being executed  
510 by a program. For example, the following policy prevents any invocation of the `fopen` and  
511 `fclose` system calls from succeeding:

$$512 \quad \lambda w_{wstate}. \lambda k_{kstate}. \lambda c_{cstate}. cname\ c \notin \{fopen, fclose\}$$

513 Note that this type of policy is expressible using existing security mechanisms on main-  
514 stream operating systems: modern Linux distributions use small eBPF programs to block  
515 programs from invoking particular system calls at runtime, according to a security policy,  
516 for example. However, the mechanism sketched above goes far beyond the expressivity  
517 of these existing systems. For example, *correctness* properties can also be captured by  
518 a policy. Assuming, for example that the `cstate` record also exposes a field `cargs` of type

---

<sup>9</sup> Note that, if we allow arbitrary axioms to be introduced into the Supervisory global theory, as many proof-assistants allow, then we need some form of *taint tracking* to ensure that challenges may only be answered by theorems deduced without axioms.



519 `cstate`  $\Rightarrow$  `nat`  $\Rightarrow$  `option (list (word 8)), which returns the byte-representation of the  $n^{\text{th}}$   
 520 argument passed to the invoked system call. With this, and assuming HOL functions strbytes  
 521 and intbytes for converting string and machine word datatypes into byte lists, respectively,  
 522 we can then capture a policy:`

```
523    $\lambda w_{\text{wstate}}. \lambda k_{\text{kstate}}. \lambda c_{\text{cstate}}. \text{cname } c = \text{fwrite} \longrightarrow$ 
524    $\text{cargs } c \ 0 = \text{Some } (\text{strbytes } \text{"foo.txt"}) \longrightarrow$ 
525    $\text{cargs } c \ 1 = \text{Some } (\text{intbytes } (\epsilon i_{\text{word } 64}. 3i^2 - 2i - 1 = 0))$ 
526
```

527 preventing any write to the file `foo.txt` from being written unless the 64-bit machine word  
 528 being written is *some* zero of a particular polynomial. In particular, the policy above  
 529 demonstrates an important point: Supervisory's policies can use any aspect of HOL,  
 530 quantifiers, choice, and all.

531 Until now, all examples have focussed on the `cstate` record which captures information  
 532 about the invoked system call. Other interesting policies can also be written in terms of  
 533 the runtime state of the Supervisory kernel itself—especially the case if we extend the  
 534 kernel with new structures. For example, by extending Supervisory to keep a log of all  
 535 system calls invoked thus far by a user-space program—for example, exposing this log as  
 536 a field `wlog` in the `wstate` record with type `wstate`  $\Rightarrow$  `nat`  $\Rightarrow$  `option event` describing a log of  
 537 system events encountered by the kernel thus far—we can then capture *trace properties* of  
 538 the program being executed. These could include policies expressing the fact that particular  
 539 system calls must be balanced in some way—for example, exactly one file may be opened  
 540 with `fopen` at a time, and opening a second file first requires the program close the other  
 541 with `fclose`—to deeper properties, including adherence to some protocol.

542 Returning back to the subject of quantification, one common security pattern deployed  
 543 by software is gradual *jailing*, or shedding of capabilities—for example, OpenBSD's `pledge`  
 544 system call allows a program to dynamically promise that it will no longer invoke particular  
 545 classes of system call, gradually dropping capabilities during a self-jailing phase, with the  
 546 operating system aborting execution of the program if this promise is ever broken. Within the  
 547 context of the discussion above, to offer a similar facility for Supervisory, we need to allow  
 548 a program to dynamically *strengthen* the prevailing policy being enforced by Supervisory.  
 549 Given the prevailing policy  $p$  we can allow the user-space program to self-jail by switching  
 550 to a new policy  $q$  if the program can *prove* to Supervisory that the new policy is more  
 551 restrictive than the previous one, in the sense that:

```
552    $\forall w_{\text{wstate}}. \forall k_{\text{kstate}}. \forall c_{\text{cstate}}. q \ w \ k \ c \longrightarrow p \ w \ k \ c$ 
```

553 That is, if we take the view that Supervisory's policies identify sets of possible behaviours,  
 554 then the user-space program must prove to Supervisory's satisfaction that the set of  
 555 permissible behaviours that may occur from now on are a subset of the behaviours that  
 556 Supervisory was previously happy to accept.

557 The material in this section has some similarity with an existing idea: *proof-carrying*  
 558 *code*. In at least one model of proof-carrying code the operating system or virtual machine  
 559 loader is modified to check proof certificates bundled with binaries for adherence to some  
 560 security or correctness property, for example memory safety, before the binary is executed.  
 561 Note, however, that these certificates are constructed *up front*, in a separate step, and merely  
 562 checked by the operating system loader. In contrast, the ideas presented above are more akin  
 563 to *proof-generating code*, wherein the user-space program and Supervisory work together  
 564 to dynamically come to an understanding that the runtime behaviour of the program adheres



to a prevailing policy. In effect, HOL is used as a *lingua franca* used to communicate demands by, and intent of, the Supervisory kernel and user-space program, respectively.

Note that the ideas above blur the lines between static and dynamic, or runtime, verification. Supervisory can be used like any other proof assistant, to statically establish properties of models of software or hardware systems, or reason about necessary truths within the rarefied domain of pure mathematics. However, it may also be used to dynamically check the runtime behaviour of programs executing under its supervision, interestingly also using theorem proving. Moreover, Supervisory used in the mode described above allows *any* program written in any programming language to be endowed with support for theorem-proving, and reasoning about its own behaviour. Indeed, a program executing on the Supervisory virtual machine *must* be prepared to explain its adherence to the system security or correctness policy in order to have any hope of performing a side-effect. With Supervisory, proof is no longer the domain of special purpose programming languages like Agda or Idris, but can be extended to any language merely by porting `libsupervisory`.

## 5 Conclusions

### 5.1 Related work

The closest related work to Supervisory is *VeriML*, an ML-like higher-order programming language with native support for theorem proving in Gordon’s HOL. Essentially, VeriML “internalises” a typical HOL kernel implementation within a higher-order programming language, promoting the abstract type of theorems—typically *defined* within the system metalanguage—into a native type of the language that can be queried and modified with new, dedicated, domain-specific expressions for theorem construction and manipulation.

From the point-of-view of a typical HOL kernel implementation, VeriML essentially “pushes the kernel down one layer” in the hierarchy of abstractions, moving the kernel from a library within the language to a first-class programming language feature. However, Supervisory “pushes” the kernel even further, moving support for theorem proving out of the programming language and into the underlying operating system—or, in our case, virtual machine. (As we will discuss below, in Subsection 5.2, this “pushing” of the kernel down through the different layers of abstractions can be taken to its logical conclusion, by moving the kernel into hardware.) Note, however, that despite the general idea behind the two projects being essentially the same, the two differ markedly in where the kernel is “pushed to”, and a myriad of design details which have some important consequences: for example, automation in Supervisory is inherently programming-language agnostic, whereas VeriML is inherently tied to one particular language—VeriML itself.

Interestingly, Supervisory can also be “pushed up one layer” in the hierarchy of abstractions, and in this light the Separation Logic specifications presented in Subsection 3.5 can be re-interpreted as *local axioms* describing the behaviour of expressions in a programming language for manipulating and querying heaps of type-formers, types, and other kernel objects. Note here, however, that the natural programming language that one obtains from extending these local axioms in the obvious way is imperative, in contrast to VeriML’s use of a mostly-functional programming language as base.

In Section 4 we observed that Supervisory’s handles can be reinterpreted as *capabilities*, in the information security sense of that word. Note that capability machines are, at the present time, having a minor resurgence, driven by the success of the CHERI capability extensions for MIPS, Arm AArch64, and RISC-V. Capabilities in hardware have a long and storied history—dating back at least to the Cambridge CAP machine developed in the

1960s—but capability-based security has also previously been applied to software, including systems software like operating systems. Whilst contemporary operating systems like seL4 and Google’s Fuchsia also have a security model built around capabilities, perhaps the best well-known historical example of a capability-based operating system was KeyKOS and its many derivatives, including EROS, the Extremely Reliable Operating System.

However, despite this long history, the Supervisory conception of capabilities differs markedly from other implementations. In particular, hardware-based capability systems like CHERI, are relatively inexpressive, extending traditional pointer types with information on valid memory regions within which they may point, and memory access permissions. This is because existing hardware-based capability systems are optimised to prevent spatial and temporal memory safety issues, inherent with widespread use of unsafe systems programming languages like the C-language, and derivatives, and must also provide an easy “on ramp” allowing existing software to adopt them. Supervisory’s conception of capabilities differs, here, in being markedly more expressive, allowing complex security and correctness properties to be expressed. However, Supervisory’s capabilities are also much more intrusive, and much harder to make use of: software must be aware of the prevailing security or correctness policy in force at the time, when trying to open a file for example, in order to be able to correctly answer the “challenge”. Moreover, using a Supervisory capability to open a file, for example, may require unbounded amounts of reasoning first, in order to address the “challenge” posed by Supervisory. This is not the case with other forms of capability, which typically act as generally-passive tokens of authority.

Lastly, Supervisory, as an implementation of HOL, is closely related to several extant systems in the wider HOL family: Isabelle/HOL, HOL4, HOL Light, Candle, and so on. The kernels of all of these systems implement very similar logics, albeit with minor modification. However, unlike the aforementioned systems, Supervisory does not follow the typical LCF-style of system organisation, nor is it written in an ML-derivative.

## 5.2 Even more future work

In addition to the ideas discussed in Section 4, below, we detail two further novel areas of future research enabled by Supervisory that we feel are worth drawing attention to:

### Hardware-accelerated proof-checking

As noted earlier, from the perspective of user-space software a system call presents as a suite of particularly CISC-like machine instructions with a rather unorthodox method of invocation. Indeed, the combination of the Supervisory system calls and the host Wasm instruction set can be, itself, thought of as a new, derived instruction set extending Wasm, with strange new domain-specific instructions for proof construction and management. Moreover, it should be quite clear that there is nothing Wasm-specific about Supervisory, and indeed Wasm was chosen merely as a relatively pain-free way of experimenting with the core ideas behind Supervisory. Indeed, Supervisory could have been implemented as real, privileged systems software for an existing instruction set in a relatively straightforward manner.

As a result, the Supervisory system call interface is already quite well-suited to an implementation in hardware, perhaps as an extension of an existing instruction set architecture like Arm AArch64 or RISC-V. The mechanism through which the Supervisory kernel isolates itself, via private memories, is rather “hardware like”, and maps nicely onto existing hardware features, and whilst the present Supervisory system call interface makes extensive use of “pointer-like” handles to refer to kernel objects, on a real hardware

656 implementation these handles could *literally* be pointers into private memories, or similar.  
657 Moreover, the system call interface itself is also further carefully designed to avoid arbitrarily  
658 large recursive structures, difficult for an instruction set architecture to handle, from being  
659 passed across the kernel system call boundary.

660     We could therefore “push Supervisory down one layer” again, into the underlying  
661 instruction set implemented by hardware. With this, the ideas presented in Section 4  
662 potentially take on a new light, as the underlying system hardware is now capable of  
663 expressing, and enforcing, arbitrarily complex security and correctness properties. We leave  
664 exploring this to future work.

## 665    **Transferring theorems between systems**

666 Assuming that an existing HOL implementation—HOL Light, for example—can be compiled  
667 into Wasm, we note that it should be possible to modify this HOL implementation by  
668 “ripping out” its kernel and replacing it with a shim layer exposing the same interface but  
669 calling Supervisory’s system calls to implement kernel functionality. From this, one can  
670 immediately “import” the entire HOL Light library directly into Supervisory, merely by  
671 having the system bootstrap itself, progressively registering new theorems in the kernel’s  
672 heaps as it executes via the Supervisory virtual machine and proves results.

673     However, intriguingly, this approach could also be used to transfer results from one HOL  
674 implementation to another by essentially performing the same shim trick with a second  
675 system. Then, this second system can execute on an instance of the Supervisory virtual  
676 machine *after* its heaps have already been populated by the first system. With this, the  
677 second HOL implementation can make reference to results populated by the first system, or  
678 build on top of them, if desired.

679     This method could not only provide a quick way of bootstrapping a library of formalized  
680 mathematics for Supervisory, by essentially “borrowing” the library of another imple-  
681 mentation, but also provides an alternative to OpenTheory for transferring results between  
682 systems. We leave investigating this in more detail for future work.

## 683    **5.3    Closing remarks**

684 We have presented Supervisory, a prototype kernel for an implementation of Gordon’s  
685 HOL. In contrast to most implementations of HOL, Supervisory is not based on the LCF  
686 architectural pattern, but is instead implemented in a style more reminiscent of a typical  
687 operating system. Supervisory is interesting in its own right: it completely dispenses with  
688 the typical metalanguage associated with an LCF-style proof-assistant, allowing automation  
689 to be written in any programming language capable of respecting the Supervisory kernel  
690 binary interface and calling conventions.

691     However, perhaps the most interesting aspect of Supervisory are some of the con-  
692 sequences of its design, and the possibilities for future work, which are yet to be fully explored.  
693 These include adopting the Supervisory kernel interface as the foundation of a hardware  
694 implementation of HOL—wherein HOL’s inference rules are implemented as machine instruc-  
695 tions that modify private memories—and the use of Supervisory as a general-purpose  
696 virtual machine that uses its proof-checking abilities to “challenge” user-space programs to  
697 explain their adherence to some system-wide security or correctness policy.

## References

- 1 Stuart F. Allen, Robert L. Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The Nuprl open logical environment. In David A. McAllester, editor, *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, volume 1831 of *Lecture Notes in Computer Science*, pages 170–176. Springer, 2000. doi:10.1007/10721959\12.
- 2 Arm. AArch64 virtual memory system architecture. <https://developer.arm.com/documentation/ddi0406/b/System-Level-Architecture/Virtual-Memory-System-Architecture--VMSA->, 2022. Arm virtual memory system architecture.
- 3 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In Nikolaj S. Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69. Springer, 2011. doi:10.1007/978-3-642-22438-6\7.
- 4 Alonzo Church. A formulation of the Simple Theory of Types. *J. Symb. Log.*, 5(2):56–68, 1940. doi:10.2307/2266170.
- 5 D. R. Engler, M. F. Kaashoek, and J. O’Toole. Exokernel: an operating system architecture for application-level resource management. In *SOSP ’95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM. URL: <http://portal.acm.org/citation.cfm?id=224076>, doi:<http://doi.acm.org/10.1145/224056.224076>.
- 6 Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. doi:10.1007/3-540-09724-4.
- 7 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with webassembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200. ACM, 2017. doi:10.1145/3062341.3062363.
- 8 John Harrison. HOL light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009. doi:10.1007/978-3-642-03359-9\4.
- 9 Gérard P. Huet and Hugo Herbelin. 30 years of research and development around Coq. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20-21, 2014*, pages 249–250. ACM, 2014. doi:10.1145/2535838.2537848.
- 10 Stephen Kell, Dominic P. Mulligan, and Peter Sewell. The missing link: explaining ELF static linking, semantically. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 607–623. ACM, 2016. doi:10.1145/2983990.2983996.
- 11 Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *SIGARCH Comput. Archit. News*, 41(1):461–472, mar 2013. doi:10.1145/2490301.2451167.
- 12 Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference*

## XX:20 Supervisory system description

- 750        *on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13,  
751        page 461–472, New York, NY, USA, 2013. Association for Computing Machinery. doi:  
752        [10.1145/2451116.2451167](https://doi.org/10.1145/2451116.2451167).
- 753    13    Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, 1990.
- 754    14    Lawrence C. Paulson, Tobias Nipkow, and Makarius Wenzel. From lcf to isabelle/hol. *Form.*  
755        *Asp. Comput.*, 31(6):675–698, dec 2019. doi:[10.1007/s00165-019-00492-1](https://doi.org/10.1007/s00165-019-00492-1).
- 756    15    Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed,  
757        César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st*  
758        *International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*,  
759        volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi:  
760        [10.1007/978-3-540-71067-7\\_6](https://doi.org/10.1007/978-3-540-71067-7_6).