All Watched Over by Machines of Loving Grace

Dominic P. Mulligan
Automated Reasoning Group, Amazon Web Services, Cambridge, UK[1]

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 *Supervisionary*, the kernel of a programmable proof-checking system for Gordon's HOL, organised in a manner more reminiscent of an operating system than a typical LCF-style proof-checker. Supervisionary's kernel 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 Supervisionary, are referenced by handles and are passed back-and-forth by system calls. Unusually, Supervisionary 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 Supervisionary, providing the resulting binary respects the kernel calling convention and binary interface, with no risk to system soundness. Lastly, Supervisionary allows arbitrary programming languages to be endowed with facilities for proof-checking. Indeed, the handles that Supervisionary uses to denote kernel objects may be thought of as an extremely expressive form of *capability* – in the computer security sense of that word – 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, Supervisionary, system description, capabilities

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.1

Supplementary Material *Software (Source Code)*: `https://github.com/DominicPM/supervisionary`
archived at `swh:1:dir:7478757cd08c06735cf3a1a056246d0200100c45`

Acknowledgements We would like to thank Nick Spinale for many insightful conversations regarding Supervisionary, and Nathan Chong and two anonymous referees for their helpful feedback on earlier drafts of this paper.

[1] All work done whilst employed within the Systems Research Group, Arm Research, Cambridge.

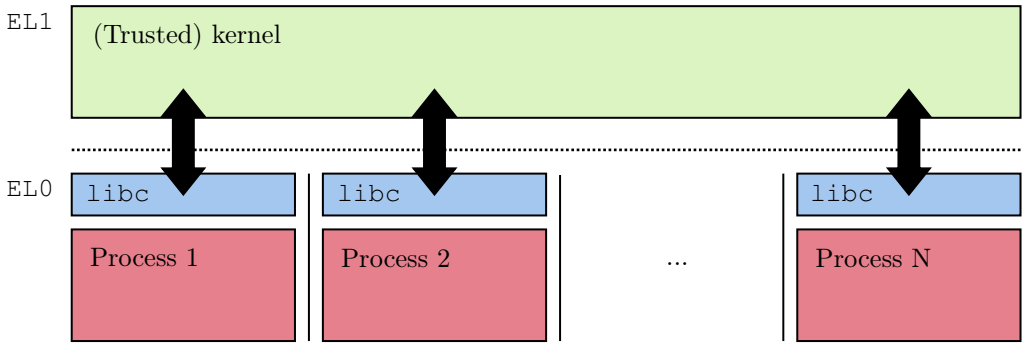

Figure 1 A schematic of the typical system organization of a commodity operating system and its associated user-space. The kernel (in green) executes at a relative level of privilege, enforced by hardware, compared to processes executing in user-space (red) – we follow the Arm convention and show the 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.

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 the reader that probing the intersection of these two areas is potentially very interesting by introducing *Supervisionary*, a programmable proof-checking system for Gordon's HOL. This system has a novel system design, with some interesting properties, and moreover some interesting consequences. We first, however, begin with a scene-setting overview of common principles in operating system design and implementation.

1.1 On operating systems

Most commodity operating systems – that is, Microsoft Windows and Unix-derivatives[2] – fit a common pattern and are architected around a relatively self-contained, trusted component typically called the system *kernel* [35].

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

The kernel self-isolates by co-operating with its host hardware. In support of this, mainstream microprocessors have, over the years, accreted a variety of now-familiar security features that an operating system kernel can use to defend itself from prying or interference. These include *exception levels* or *privilege rings*, as they are variously called, depending on the instruction set architecture, and which introduce a notion of *privilege* into the system.

[2] *Commodity* here is used to guard against pedantic quibbling over research operating system designs – like exokernels [9] and other oddities – which arguably do not fit this pattern.

Here, software executing at higher-privilege – in our case, an operating system kernel[3] – gains permission to program sensitive system registers, adjust hardware operating frequencies and voltages, and generally control 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, for example reading data from, or writing data to, a buffer within the memory space of an untrusted user-space process.

Modern microprocessors also provide a form of memory management built around page tables (see e.g. [3]). 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 translation 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, a user-space process, or both – if it is *mapped in* to that principal's address space. Moreover, page tables are also used for storing the metadata attributes of pages of memory, including read-write-execute permissions. By correctly initialising and managing these tables the kernel can 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 computers, 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 largely synchronous *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-space fragment of the instruction set architecture of the microprocessor with new macro 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.[4] 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`. Generally, this is just a convenience, and user-space software may invoke system calls directly if wanted by invoking the correct machine instruction and adhering to the appropriate calling convention.[5]

[3] 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.

[4] For more detail on the role of the system ABI, its other aspects, and its very real effects on the semantics of executing programs, see this [18] outrageously well-written yet criminally under-cited overview.

[5] 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.

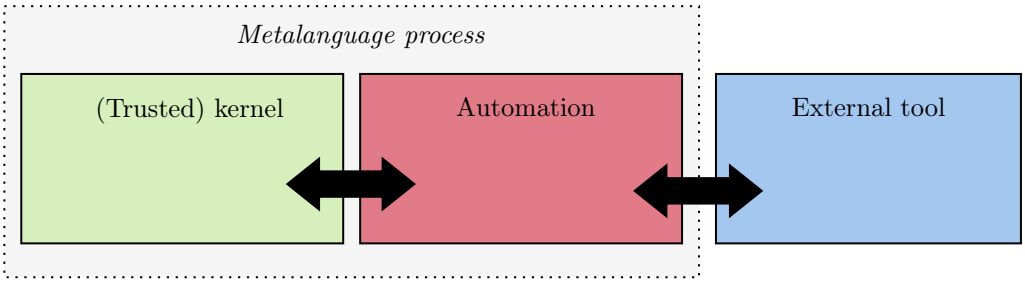

Figure 2 A schematic of the system organisation of a typical LCF-style proof assistant. The trusted kernel (green) is linked against untrusted automation (red) existing within the same metalanguage process (dotted line) and communicate with each other using the kernel's API (leftmost black arrow). External 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).

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

1.2 On programmable proof-checkers

Most modern proof-assistants – for example, systems in the wider HOL family [27, 14, 33], Coq [15], Matita [4], NuPRL [2], and similar – fit a common pattern and are architected around a relatively self-contained, trusted component typically called the system *kernel*.

The system kernel is the sole component that can authenticate claims as legitimate theorems of the implemented logic. Untrusted automation, residing outside of the kernel, must "drive" the kernel to derive a theorem on its behalf. The kernel is therefore *the* component responsible for ensuring system-wide soundness, and represents the "root of all trust" within the system. It is therefore imperative that the kernel is able to isolate itself sufficiently from untrusted automation at all times. This kernel-centric method of system organisation is known as *the LCF approach* after Milner's eponymous system [11]. See Figure 2 for a diagrammatic representation.

Most modern proof-assistants tend to be written in a *metalanguage* which serves as the implementation language for both the kernel and the majority of the untrusted automation that modern proof-assistants provide to users. This metalanguage is typically a strongly-typed functional programming language, for example an ML derivative such as OCaml or SML [22], and which offers strong modularity and abstraction features. The kernel exploits these programming language features to hide its own data structures from untrusted automation and expose 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. 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 and enforce type abstraction. Moreover, for systems that use ephemeral proof construction, and lack an explicit notion of serialised proof-representation such as a *proof-term* or similar, the system metalanguage is 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 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 Supervisionary system

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 types 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 is enforced are different: for operating system kernels[6] 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 *Supervisionary*, the kernel of a novel programmable proof-assistant for Gordon's HOL.[7] Supervisionary's design has more in common with a typical operating system than comparable implementations of HOL. Specifically, the Supervisionary 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 Supervisionary'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 Supervisionary 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 *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 Supervisionary kernel on user-space's behalf. Handles – which can be thought of as pointers, indexing Supervisionary's private memories – are used by a user-space process to identify kernel objects that the kernel should manipulate or query.

Notably, Supervisionary 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 [17]. Note that this decision introduces no risk to system soundness, as Supervisionary's soundness ultimately rests on the continued separation of kernel-private data from Supervisionary's analogue of user-space – using privilege

[6] Barring unikernels, or library operating systems, like Mirage [20, 21]. 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.

[7] Many of the ideas presented henceforth are logic-independent. Though we have chosen to use HOL the ideas presented herein can be applied to a wide variety of other logics and type theories with relatively straightforward changes.

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 Supervisionary kernel. Supervisionary 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 Supervisionary is implemented as a WebAssembly [12] (Wasm henceforth) host. We extend a Wasm virtual machine with new system calls that perform a context switch into Supervisionary, which has its own memory isolated from the memory of the executing user-space Wasm process running under its supervision, and inaccessible to it. This separation is only one way: the kernel can "peer in" to the runtime state of a running Wasm process and read from, or write to, its private memories. This decision means we may experiment with the fundamental ideas behind Supervisionary – 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, for example. 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. Supervisionary's design will be fully described in Section 3.

Lastly, and more speculatively, Supervisionary's handles can be passed around a program, between different programs executing concurrently or sequentially under Supervisionary's management, or between the user-space program and the kernel. Whilst this property is not unique to Supervisionary – 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 Supervisionary kernel. Handles to these theorems act as a form of *capability*, in the computer security sense of that word. This property is unique to Supervisionary, as it rests on Supervisionary's dual status as a proof-assistant kernel, capable of generating and checking theorems, and an extension of a general purpose virtual machine, capable of executing arbitrary programs. Here, Supervisionary exploits its status as a "pinch point" that user-space cannot help pass through in order to have any sort of computational effect, to force user-space to pass a handle denoting a theorem that *proves* that it is acting correctly, per some system-wide policy. Some ideas of how this idea could develop are discussed later, in Section 4.

2 Implemented logic

Supervisionary implements a variant of Gordon's HOL [10], a classical higher-order logic. This can be intuitively understood as Church's Simple Theory of Types [7] extended with ML-style top-level polymorphism. We introduce the basics of this logic here, introducing just enough material that the unfamiliar reader can follow the rest of the paper.

We fix a denumerable set of *type variables* and use α, β, γ, and so on, to range arbitrarily over them. We work with *simple types* generated by the following recursive grammar:

$$\tau, \tau', \tau'' ::= \alpha \mid \mathsf{f}(\tau, \ldots, \tau')$$

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

$$\frac{r : \tau}{\Gamma \vdash r = r} \qquad \frac{\Gamma \vdash r = s}{\Gamma \vdash s = r} \qquad \frac{\Gamma \vdash r = s \quad \Gamma' \vdash s = t}{\Gamma \cup \Gamma' \vdash r = t} \qquad \frac{\phi \in \Gamma}{\Gamma \vdash \phi} \qquad \frac{\Gamma \vdash \bot \quad \phi : \mathsf{bool}}{\Gamma \vdash \phi}$$

$$\frac{\Gamma \vdash r = s \quad \Gamma' \vdash t = u}{\Gamma \cup \Gamma' \vdash r\ t = s\ u} \qquad \frac{\Gamma \vdash r = s \quad x_\tau \notin fv(\Gamma)}{\Gamma \vdash \lambda x_\tau.r = \lambda x_\tau.s} \qquad \frac{}{\Gamma \vdash \top}$$

$$\frac{\Gamma \vdash \phi \quad \Gamma' \vdash \psi}{\Gamma \cup \Gamma' \vdash \phi \wedge \psi} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi} \qquad \frac{\Gamma \cup \{\phi\} \vdash \psi \quad \phi : \mathsf{bool}}{\Gamma \vdash \phi \longrightarrow \psi}$$

$$\frac{\Gamma \vdash \phi \longrightarrow \psi \quad \Gamma' \vdash \phi}{\Gamma \cup \Gamma' \vdash \psi} \qquad \frac{\Gamma \vdash \phi \quad \psi : \mathsf{bool}}{\Gamma \vdash \phi \vee \psi} \qquad \frac{\Gamma \vdash \psi \quad \phi : \mathsf{bool}}{\Gamma \vdash \phi \vee \psi} \qquad \frac{\Gamma \vdash \phi = \psi \quad \Gamma' \vdash \phi}{\Gamma \cup \Gamma' \vdash \psi}$$

$$\frac{\Gamma \vdash \phi \vee \psi \quad \Gamma' \cup \{\phi\} \vdash \xi \quad \Gamma'' \cup \{\psi\} \vdash \xi}{\Gamma \cup \Gamma' \cup \Gamma'' \vdash \xi} \qquad \frac{\Gamma \vdash \phi \longrightarrow \psi \quad \Gamma' \vdash \psi \longrightarrow \phi}{\Gamma \cup \Gamma' \vdash \phi = \psi}$$

$$\frac{\Gamma \vdash \exists x_\tau.\phi \quad \Gamma \cup \{\phi[x_\tau := y_\tau]\} \vdash \psi \quad y_\tau \notin fv(\psi) \cup fv(\Gamma) \cup \{x_\tau\}}{\Gamma \vdash \psi} \qquad \frac{\Gamma \vdash \phi = \psi \quad \Gamma' \vdash \psi}{\Gamma \cup \Gamma' \vdash \phi}$$

$$\frac{\Gamma \cup \{\phi\} \vdash \bot \quad \phi : \mathsf{bool}}{\Gamma \vdash \neg\phi} \qquad \frac{\Gamma \vdash \neg\phi \quad \Gamma' \vdash \phi}{\Gamma \cup \Gamma' \vdash \bot} \qquad \frac{\Gamma \vdash \forall x_\tau.\phi \quad r : \tau}{\Gamma \vdash \phi[x_\tau := r]}$$

$$\frac{\Gamma \vdash \phi[x_\tau := r]}{\Gamma \vdash \exists x_\tau.\phi} \qquad \frac{\Gamma \vdash \phi \quad x_\tau \notin fv(\Gamma)}{\Gamma \vdash \forall x_\tau.\phi} \qquad \frac{s : \tau' \quad r : \tau}{\Gamma \vdash (\lambda x_\tau.s)r = s[x_\tau := r]} \qquad \frac{\Gamma \vdash \exists x_\tau.\phi}{\Gamma \vdash \phi(\epsilon x_\tau.\phi)}$$

$$\frac{f : \tau \Rightarrow \tau' \quad x_\tau \notin fv(f)}{\Gamma \vdash \lambda x_\tau.(f\ x) = f} \qquad \frac{\Gamma \vdash \phi \quad r : \tau}{\Gamma[x_\tau := r] \vdash \phi[x_\tau := r]} \qquad \frac{\Gamma \vdash \phi}{\Gamma[\alpha := \tau] \vdash \phi[\alpha := \tau]}$$

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

types in Supervisionary. 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 α with τ' in the type τ. From the outset we assume two primitive type-formers built-in to the logic itself and necessary to bootstrap the rest of the material: 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 τ we assume a countably infinite set of *variables* and *constant symbols*. We use x_τ, y_τ, z_τ, and so on, to range over the variables associated with type τ, and similarly use C_τ, D_τ, E_τ, and so on, to also range over the constants associated with type τ. With these, we recursively define *terms* of the explicitly-typed λ-calculus, as follows:

$$r, s, t ::= x_\tau \mid \mathsf{C}_\tau \mid rs \mid \lambda x{:}\tau.r$$

Note that there is an "obvious" simple-typing relation on terms, which is presented in Figure 4. We write $r : \tau$ to assert that a derivation tree rooted at $r : \tau$ and constructed according to the rules in Figure 4 exists, or more intuitively, that r has type τ. We call any term with a type *well-typed*; we will only ever work with well-typed terms in Supervisionary. Also, we call a term with type bool a *formula* and use ϕ, ψ, ξ, and so on, to suggestively range over terms that should be understood as being formulae in the rest of the paper. We work with

$$\frac{}{x_\tau : \tau} \qquad \frac{}{\mathsf{C}_\tau : \tau} \qquad \frac{r : \tau \Rightarrow \tau' \quad s : \tau}{rs : \tau'} \qquad \frac{r : \tau'}{\lambda x_\tau . r : \tau \Rightarrow \tau'}$$

Figure 4 The typing relation on terms.

terms identified up-to α-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.

As with type-formers, from the outset we assume a collection of typed constants needed to boostrap the rest of the logic, summarised in the table below:

$=$		$\alpha \Rightarrow \alpha \Rightarrow \mathsf{bool}$
$\top$, $\bot$		bool
$\neg$		$\mathsf{bool} \Rightarrow \mathsf{bool}$
$\wedge$, $\vee$, $\longrightarrow$	*with type*	$\mathsf{bool} \Rightarrow \mathsf{bool} \Rightarrow \mathsf{bool}$
$\forall$, $\exists$		$(\alpha \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool}$
ϵ		$(\alpha \Rightarrow \mathsf{bool}) \Rightarrow \alpha$

Most of the constants 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 ϵ constant – Hilbert's *description operator* [23], 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 $\epsilon x_\tau . \bot$ 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 Γ, Γ', Γ'', 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 dyadic *Natural Deduction relation* betwixt contexts and formulae using 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 [14], a point we touch on later in Section 5. We prefer a more explicit treatment here, closer to a standard textbook presentation of Natural Deduction.

3 The Supervisionary kernel state

Supervisionary'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 objects and their interdependencies, as introduced in Section 2.

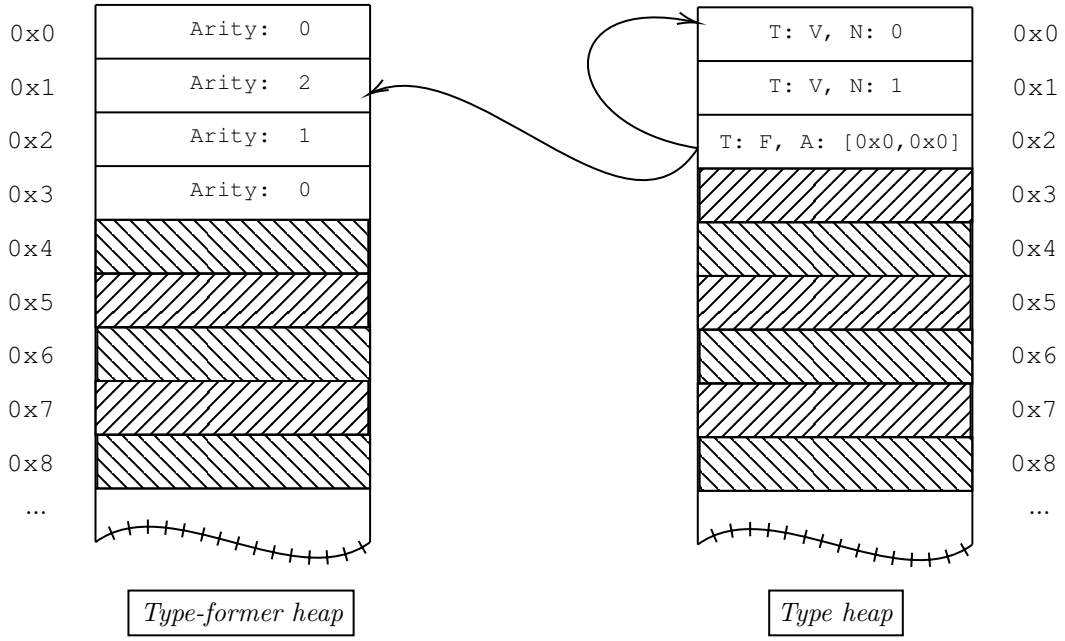

Figure 5 Entries within the Supervisionary kernel's type heap referencing entries within the type-former heap. Cross-hatched heap cells are as-yet unallocated by the kernel. The cell allocated at address `0x2` in the type heap is tagged with the `F` tag, indicating it is a type-former applied to a list 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 α 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 $\alpha \Rightarrow \alpha$.

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 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 Supervisionary 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 the purview of Supervisionary. Two primitive type-formers, pre-registered in the type-former heap on system boot, are however exceptions to this rule 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

⇒, registered at address `0x1`, with arity 2. The existence of these type-formers must be understood by user-space, as they form part of the Supervisionary system interface, similar to how the distinguished file handles `stdout` and `stdin` are part of the POSIX system interface and must be understood by user-space and kernel alike to support file I/O.

3.2 The type heap

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

Recalling Section 2, types are either a type-variable or a *combination* of a type-former applied to a list of types. All entries within the type heap are therefore tagged indicating whether they are a type-variable or a combination. Type-variable entries contain one datum: the *name* of the type-variable, an unsigned 64-bit machine word. Combination entries 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 5 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.

Supervisionary also boots with some entries in the type heap pre-registered, corresponding to common or useful types used to boostrap the rest of the logic. These include the Boolean type, `bool`, common type variables – α and β, for example – as well as larger, more complex types such as the type of the polymorphic equality, $\alpha \Rightarrow \alpha \Rightarrow$ `bool`. The handles for all of these pre-registered entities must likewise 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 Supervisionary 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 Supervisionary 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, Supervisionary 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 Supervisionary'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 α-equivalence classes of terms – at present, we use a name-carrying syntax, but could implement this using De Bruijn indices or levels [8], leading to a more efficient implementation.

3.4 The theorem heap

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

A theorem kernel object can be deconstructed using the `Theorem.Split.Assumptions` and `Theorem.Split.Conclusion` system calls, to obtain the list of assumption and conclusion of the theorem object, respectively. However, the only way that a new entry in the heap of theorems can be constructed is by using one of a series of system calls corresponding to an inference rule of the logic's Natural Deduction relation, presented in Section 2, or of the definitional principles of HOL. Taking the *negation introduction* system call, for example:

$$\frac{\Gamma \cup \{\phi\} \vdash \bot \quad \phi : \mathsf{bool}}{\Gamma \vdash \neg\phi}$$

We have a corresponding system call `Theorem.Register.Negation.Introduction` which takes a handle pointing-to a sequent, $\Gamma \cup \{\phi\} \vdash \bot$, in the kernel's theorem heap, and a handle pointing-to a term, ϕ, in the kernel's term heap, and returns a handle pointing-to a theorem, $\Gamma \vdash \neg\phi$, also residing in the kernel's theorem heap if all error checks pass for the inputs.

Like terms, theorem handles point-to α-equivalence classes of theorem objects, wherein two sequents are considered the same if their respective constituent handles point-to the same α-equivalence classes of terms. Moreover, the Supervisionary kernel also enforces *maximal sharing* in all of its kernel heaps, and an attempt to register an object that has already been registered, up-to α-equivalence, does not allocate a new slot in the respective kernel heap, but merely returns the existing handle to the object. These two decisions make some operations within the Supervisionary kernel easier to implement, at the expense of slowing down the registering of new objects. For example, we know that objects are α-equivalent when their handles are identical. Moreover, in `Theorem.Register.Negation.Introduction` above, we know that the formula ϕ is not in the context $\Gamma \cup \{\phi\}$ if the second input handle, mentioned above, does not appear in the list of handles representing the assumptions of the sequent. Note that this would not be the case if we did not enforce maximal sharing: *another* handle pointing-to the term ϕ may be present in the list of assumptions of the theorem, different from the handle passed in from user-space, and this would force Supervisionary to have to perform a "deep scan" of its heaps in trying to work out whether the two handles supposedly pointed-to the same HOL formula. As a result of this sharing, Supervisionary's heaps remain inductive in the sense previously discussed, but recursively-defined objects represented within them are not necessarily encoded as trees, but rather directed acyclic graphs.

Moreover, we previously mentioned that heaps must continue to grow monotonically at all times, lest we inadvertently introduce an unsoundness into the system by allowing the HOL `bool` type, or similar, to be redefined. However, note that this invariant *could* be weakened, somewhat, by "working backwards" from the kernel's theorem heap and removing objects in other kernel heaps that are not referenced via a transitive points-to relation. Essentially this would represent a form of *mark-and-sweep garbage collection* [31] wherein objects in the kernel's theorem heaps are root objects, with other objects deallocated if they are not reachable from these roots. Care must be taken to ensure that the primitive kernel objects, pre-provisioned into the heaps at system boot, can never be deallocated, even if currently unreachable. Whilst possible, this garbage collection process is not at present implemented in Supervisionary, as sharing compresses the heaps with no pressing need to remove objects from them. Moreover, within the context of garbage collection, user-space cannot be sure that a handle generated by the kernel, and previously denoting a registered kernel object, is stable and now does not dangle, complicating the Supervisionary programming model.

3.5 Specifying kernel functions

Implementing and using the Supervisionary kernel is an extended exercise in heap and pointer manipulation, and until now the kernel's system calls were explained in an intuitive, informal sense. To specify the behaviour of some of our kernel system calls, we therefore reach for an existing tool used to specify pointer-manipulating programs: Separation Logic [30, 16].

Working abstractly, we represent handles as elements of the set $\mathbb{N}$ of natural numbers, and use h, h', h'', and so on, to range over handles. For a fixed set A, we say that a partial-function $f : \mathbb{N} \rightharpoonup A$ is *finitely-supported* when the set $dom(f) = \{x \mid f\ x \text{ defined}\}$ is finite. We call such a finitely-supported partial map into a set A an *A-heap*. We write 0 for an empty A-heap, and for two A-heaps f and g we write $f \sharp g$ to assert that their domains are disjoint, so $dom(f) \cap dom(g) = \{\}$. This relation is symmetric and $0 \sharp g$ always, for any g. Moreover, for two A-heaps f and g we can "glue them together", using the function $f \oplus g$, to form a larger A-heap. This function is defined piecewise as:

$(f \oplus g)\ x = f\ x$ if $x \in dom(f)$
$(f \oplus g)\ x = g\ x$ if $x \in dom(g)$
$(f \oplus g)\ x$ is undefined otherwise

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

We define *types*, *constants*, *terms*, and *theorems* by the following non-recursive grammars, where m ranges over arbitrary natural numbers:

$$\begin{aligned} t, t', t'' &::= \texttt{TyVar}\ m \mid \texttt{TyFm}\ h\ (h_1, \ldots, h_n) \\ C, C', C'' &::= \texttt{TConst}\ h\ h' \\ r, r', r'' &::= \texttt{Var}\ m\ h \mid \texttt{Const}\ h\ h' \mid \texttt{App}\ h\ h' \mid \texttt{Lam}\ m\ h\ h' \\ s, s', s'' &::= \texttt{Seq}\ (h_1, \ldots, h_n)\ h \end{aligned}$$

We call heaps over types a *type-heap*; similarly for constants, terms, and theorems. We also call heaps over natural number arities a *type-former heap*.

Fix a set of kernel states K. We use k, k', k'', and so on, to range over kernel states, each of which is a 5-tuple $\langle F, Ty, C, Tm, Th \rangle$ consisting of a type-former heap, a type heap, a constant heap, a term heap, and a theorem heap respectively. We extend our notion of disjointness to kernel states, and write $k \sharp k'$ to assert that all of the respective heaps in kernel states k and k' are disjoint. We further abuse notation and write 0 for the *empty kernel state* consisting of five empty heaps, and $k \oplus k'$ for the "gluing" of two kernel states together, wherein each of the respective heaps in k and k' are joined pointwise using $\oplus$. Note that, again, $k \oplus k'$ is well-defined whenever $k \sharp k'$.

We define *assertions* as sets of kernel states, use A, B, C, and so on, to range over them, and write $k \vDash A$ to assert that $k \in A$. We pay especial attention to some particular assertions of note that will be useful in specifying some of our system calls:

$$\begin{aligned} \bullet &\equiv \{\langle 0, 0, 0, 0, 0 \rangle\} \\ A \star B &\equiv \{k'' \mid \exists k\ k'.k'' = k \oplus k' \text{ and } k \sharp k' \text{ and } k \vDash A \text{ and } k' \vDash B\} \\ h \mapsto_{\text{Aty}} a &\equiv \{\langle h \mapsto a, 0, 0, 0, 0 \rangle\} \\ h \mapsto_{\text{Typ}} t &\equiv \{\langle 0, h \mapsto t, 0, 0, 0 \rangle\} \end{aligned}$$

We further define the standard logical constants and connectives as abbreviations for setwise operations, writing $\bot$ for $\{\}$, $C \wedge D$ for $C \cap D$, and $\exists x.C\ x$ for $\bigcap_x . C\ x$, for example.

Fix a set of *values*, V, consisting *at least* of handles and numeric error codes. System calls e, f, g, and so on, are modelled as total functions from kernel states to kernel states which also produce a value as a side-effect, that is $e : K \to V \times K$. 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 – the Supervisionary equivalent of a "blue screen of death".

With this in mind, we define a Separation Logic triple as a three-place relation between an assertion, a system-call, and a function from values to assertions by:

$A \vdash e \dashv \lambda r.B$ iff for any C if $k \vDash A \star C$ and $e\ k = \langle v, k' \rangle$ then $k' \vDash (\lambda r.B)v \star C$

With this, we specify the behaviour of the `TypeFormer.Register` system call as follows:

- $\bullet \vdash \texttt{TypeFormer.Register}(a) \dashv \lambda h.h \mapsto_{Aty} a$

Note that this specification correctly captures the fact that the call can never fail: it will always return a handle pointing-to a new cell in the type-former heap, containing the required arity, with no other effects on the kernel heaps.

Specifying system calls which manipulate types, constants, terms, or theorems is more complex as we must assume that any handles contained within these structures point-to allocated cells in an appropriate kernel heap. To do this, we use of a family of *shape predicates* relating encodings of objects within the kernel's heaps to the recursively-defined structures of Section 2. Assuming a bijection V between natural numbers and type-variables, and a bijection F between handles and type-formers, we inductively define the relation $\texttt{TYPE}\ h\ \tau$:

$$\frac{h \mapsto_{Typ} \texttt{TVar}\ m \quad (V\ m\ \alpha)}{\texttt{TYPE}\ h\ \alpha}$$

$$\frac{h \mapsto_{Typ} \texttt{TyFm}\ h'\ (h_1, \ldots, h_n) \star h' \mapsto_{Aty} n \star \texttt{TYPE}\ h_i\ \tau_i \quad (1 \leq i \leq n, F\ h'\ \texttt{f})}{\texttt{TYPE}\ h\ \texttt{f}(\tau_1, \ldots, \tau_n)}$$

We omit comparable shape predicates for constants, terms, and theorems, as the pattern should be clear. Note that the basic allocation functions for types, upon success, generate kernel states wherein the `TYPE` relation holds. For example, assuming a correspondence, $V\ n\ \alpha$, between the natural number n and type-variable α:

- $\bullet \vdash \texttt{Type.Register.Variable}(n) \dashv \lambda h.\texttt{TYPE}\ h\ \alpha$

Similarly, we have:

$$\begin{array}{l} \quad h \mapsto_{Aty} n \star \texttt{TYPE}\ h_1\ \tau_1 \star \ldots \star \texttt{TYPE}\ h_n\ \tau_n \\ \vdash \texttt{Type.Register.TypeFormer}(h, h_1, \ldots, h_n) \dashv \\ \lambda r.h \mapsto_{Aty} n \star \texttt{TYPE}\ h_1\ \tau_1 \star \ldots \star \texttt{TYPE}\ h_n\ \tau_n \star \texttt{TYPE}\ r\ \texttt{f}(\tau_1, \ldots, \tau_n) \end{array}$$

Which also captures the fact that existing well-formed kernel heaps remain well-formed after invocation of a system call, with shape predicate invariants formally capturing the kernel invariants previously informally introduced.

3.6 Programming in user-space

The system call interface presents a very low-level, austere interface to user-space code. To make programming Supervisionary less tedious, a utility library, similar in function to `libc`, is provided to user-space in order to raise the level of abstraction above the raw system call interface. This is provided as `libsupervisionary`, 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 `libsupervisionary`, can provide pretty-printing and parsing routines for types and terms, automation, proof-state management, and other functions typical of a proof-assistant.

4 Future work

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

4.1 Capabilities on steroids

As described, Supervisionary 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 Supervisionary's various heaps, using the series of system calls progressively introduced in Section 3. Programs executing under Supervisionary are so-far incapable of opening files on the user's machine, communicating over sockets, or querying the system time, because Supervisionary does not provide any system calls to allow a program to perform any of those activities. However, it could.

Specifically, Supervisionary 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, Supervisionary is transformed into a general-purpose virtual machine, akin to the Java Virtual Machine, capable of executing arbitrary programs – calculators, simulations, file search utilities, and so on – albeit with a bizarre set of extra system calls dedicated to theorem proving. In short, by extending Supervisionary with system calls for querying and manipulating the system state, Supervisionary is *both* a proof-assistant and a general-purpose virtual machine – though these two facets of the system are kept separate.

These two families of system call need not be kept separated, however. Prior to allowing a user-space program to open or read a file, Supervisionary could first demand that a (handle to a) theorem is supplied to it as an extra argument to the file-open system call, `fopen`, for example. Interestingly, because Supervisionary executes at a relative level of privilege, and can "peer in" to the runtime state of a user-space program, the statement of this desired theorem can be a *function* of the runtime state of the user-space program itself, of the runtime state of the Supervisionary kernel, and also of the various arguments and other details of the system call being invoked. This statement – which we will call the *challenge* – can be any arbitrary formula written in HOL, and can be generated dynamically by the kernel, perhaps in accordance with a *global policy* enforced by Supervisionary. A failure to produce a handle to address a particular challenge causes the system call to fail, with a runtime failure.

For concreteness, suppose we fix HOL types wstate, kstate, and cstate, which you may imagine as being record types capturing details of the runtime state of the executing Wasm process, the runtime state of the Supervisionary kernel, and the details of the system call being invoked. Supervisionary can dynamically *reflect* the actual runtime states of the user-space program and kernel, and the invoked system call, into inhabitants of these HOL types. Then, supposing our prevailing security policy, p, is a HOL function of type $\mathsf{wstate} \Rightarrow \mathsf{kstate} \Rightarrow \mathsf{cstate} \Rightarrow \mathsf{bool}$, a challenge is obtained by dynamically applying p to the reflected records, described above. Two particularly special security policies exist:

$$\lambda w_{\mathsf{wstate}}.\lambda k_{\mathsf{kstate}}.\lambda c_{\mathsf{cstate}}.\bot \qquad \text{and} \qquad \lambda w_{\mathsf{wstate}}.\lambda k_{\mathsf{kstate}}.\lambda c_{\mathsf{cstate}}.\top$$

When applied to a reflected runtime state, these two policies generate the challenges $\bot$ and $\top$, respectively. The first policy is therefore the *deny all* policy, which essentially prevents a user-space program from invoking *any* system call, and making any effect on the system state, whilst the second is the *allow all* policy which can always be trivially satisfied by passing the handle to HOL's truth introduction theorem.[8] Between these two extremal points are

[8] Note that, if we allow arbitrary axioms to be introduced into the Supervisionary global theory, as many

a variety of other interesting policies, however. For example, if we assume that our cstate record contains a field cname of type cstate ⇒ string capturing the name of the system call being invoked, then we may selectively prevent particular system calls from being executed by a program. The following policy prevents any invocation of the `fopen` and `fclose` system calls from succeeding, for example:

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

This policy is expressible using existing security mechanisms on mainstream operating systems: modern Linux distributions use small eBPF programs to block programs from invoking particular system calls at runtime, according to a security policy, for example. However, the mechanism sketched above goes far beyond the expressivity of these existing systems as *correctness* properties can also be captured by a policy, for example. Assume, for example, that the cstate record also exposes a field cargs of type cstate ⇒ nat ⇒ option list word 8, which returns the byte-representation of the n^{th} argument passed to the invoked system call. With this, and assuming HOL functions strbytes and intbytes for converting string and machine word datatypes into byte lists, respectively, we can then express

$$\begin{array}{l}\lambda w_{\mathsf{wstate}}.\lambda k_{\mathsf{kstate}}.\lambda c_{\mathsf{cstate}}.\mathsf{cname}\ c = \texttt{fwrite} \longrightarrow \\ \quad \mathsf{cargs}\ c\ 0 = \mathsf{Some}\ (\mathsf{strbytes}\ \text{"foo.txt"}) \longrightarrow \\ \quad\quad \mathsf{cargs}\ c\ 1 = \mathsf{Some}\ (\mathsf{intbytes}\ (\epsilon i_{\mathsf{word}\ 64}.3i^2 - 2i - 1 = 0))\end{array}$$

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

Until now, all examples have focussed on the cstate record which captures information about the invoked system call. Other interesting policies can also be written in terms of the runtime state of the Supervisionary kernel itself. This idea becomes especially interesting if we extend the kernel with new structures recording aspects of a program's execution. By extending Supervisionary to keep a log of all system calls invoked thus far by a user-space program – for example, exposing this log as a field wlog in the wstate record with type wstate ⇒ nat ⇒ option event – we can capture *trace properties* of the executing program. For instance, one may assert that system call invocations must be balanced in some way – exactly one file may be opened at a time, and opening a second file first requires the program close the other, for example – and also deeper properties, including adherence to a protocol.

One common security pattern deployed by software is gradual *jailing*, or shedding of capabilities – for example, OpenBSD's `pledge` system call allows a program to dynamically shed the ability to further invoke particular classes of system call, gradually dropping capabilities during a self-jailing phase. To offer a similar facility for Supervisionary, we need to allow a program to dynamically *strengthen* the prevailing policy being enforced by Supervisionary. Given the prevailing policy p we can allow the user-space program to self-jail by switching to a new policy q if the program can *prove* to Supervisionary that the new policy is more restrictive than the previous one, in the sense that:

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

---

proof-assistants allow, then we need some form of *taint tracking* to ensure that challenges may only be answered by theorems deduced without axioms.

If we view Supervisionary's policies as identifying sets of possible system behaviours, then the user-space program must prove to Supervisionary's satisfaction that the set of permissible behaviours that may occur from now on are a subset of the behaviours that Supervisionary was previously happy to accept. Note here that *quantification* is used in an essential way.

The material in this section has some similarity with an existing idea: *proof-carrying code* [24]. In one model of proof-carrying code the operating system or virtual machine loader is modified to check proof certificates bundled with binaries for adherence to some security or correctness property, for example memory safety, before the binary is executed. Note, however, that these certificates are constructed *up front*, in a separate step, and merely checked by the operating system loader. In contrast, the ideas presented above are more akin to *proof-generating code*, wherein the user-space program and Supervisionary work together 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 Supervisionary kernel and user-space program, respectively.

The ideas above also blur the lines between static and dynamic, or runtime, verification. Supervisionary 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, Supervisionary 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 Supervisionary 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 Supervisionary, proof is no longer the exclusive domain of dedicated programming languages like Agda [26] or Idris [6, 5], but can be extended to any language merely by porting `libsupervisionary`.

4.2 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 Supervisionary 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 Supervisionary, and indeed Wasm was chosen merely as a relatively pain-free way of experimenting with the core ideas behind Supervisionary. Indeed, Supervisionary could have been implemented as real, privileged systems software for an existing instruction set in a relatively straightforward manner.

As a result, the Supervisionary 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 Supervisionary kernel isolates itself, via private memories, is rather "hardware like", and maps nicely onto existing hardware features, and whilst the present Supervisionary system call interface makes extensive use of "pointer-like" handles to refer to kernel objects, on a real hardware implementation these handles could *literally* be pointers into private memories, or similar. Moreover, the system call interface itself is also further carefully designed to avoid arbitrarily large recursive structures, difficult for an instruction set architecture to handle, from being passed across the kernel system call boundary. We could therefore "push Supervisionary down one layer" again, into the underlying instruction set implemented by hardware. With this, the ideas presented in Subsection 4.1 take on a new light, as the system hardware is now capable of expressing, and enforcing, arbitrarily complex security and correctness properties.

5 Conclusions

5.1 Related work

The closest related work to Supervisionary is *VeriML*, an ML-like language extended with limited dependent-types ranging over HOL terms and theorems [34]. 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.

Compared to a typical HOL kernel, 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, Supervisionary "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. (And, as discussed above, this "pushing" of the kernel down through the different layers of abstractions can be taken to its logical conclusion, by pushing the kernel all the way into hardware.) Note, however, that despite the general idea behind the two projects being essentially the same, the two differ markedly in a myriad of design details which have some important consequences: for example, automation in Supervisionary is inherently programming-language agnostic, whereas VeriML is inherently tied to one particular language – VeriML itself.

Interestingly, some of the ideas used in Supervisionary can also be "pushed up one layer" in the hierarchy of abstractions. Specifically, the Separation Logic specifications presented in Subsection 3.5 can be re-interpreted as a series of *local axioms* describing the behaviour of statements or expressions in a programming language for registering, manipulating and querying type-formers, types, and other objects, in a series of heaps secreted from the user, managed by the language's runtime. Interestingly, the natural programming language that one obtains from this exercise is imperative, in contrast to the functional VeriML. (To make a more ergonomic programming language it would make sense for these expressions to be modified so that they manipulate built-in recursive types of the programming language – corresponding to HOL type-formers, types, constants, terms, and theorems – in a similar fashion to VeriML, rather than make use of Supervisionary's handles and its incremental construction of recursive structures.)

In Subsection 4.1 we observed that Supervisionary'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 renaissance, driven by the success of the CHERI capability extensions for MIPS, Arm AArch64, and RISC-V [25]. Capabilities in hardware have a long and storied history – dating at least to the Cambridge CAP machine developed in the 1970s – and capability-based security has also previously been applied to programming languages and software, including systems software like operating systems. Whilst contemporary operating systems like seL4 [19] and other L4 derivatives have a security model built around capabilities, perhaps the best well-known historical example of a capability-based operating system was KeyKOS [29, 13] and its many derivatives, including EROS, the Extremely Reliable Operating System [32]. However, despite this long history, the Supervisionary conception of capabilities differs from other implementations as hardware-based capability systems like CHERI, are relatively inexpressive, merely 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 in the use of unsafe systems programming languages

like the C-language, and derivatives, and must provide an easy "on ramp" allowing existing software to adopt them. Supervisionary's conception of capabilities differs, here, in being more expressive, allowing complex security and correctness properties to be expressed, but 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". Using a Supervisionary capability to open a file may also require unbounded amounts of reasoning first, in order to address the "challenge" posed by Supervisionary, which is not the case with other forms of capability, which act as passive tokens of authority.

Lastly, Supervisionary, as an implementation of HOL, is closely related to several extant systems in the wider HOL family: Isabelle/HOL, HOL4, HOL Light, Candle [1], and so on. The kernels of all of these systems implement very similar logics, albeit with minor modification. However, unlike the aforementioned systems, Supervisionary does not follow the typical LCF-style of system organisation, nor is it written in an ML-derivative.

5.2 On trust

The current Supervisionary proof-checking kernel consists of approximately 6,600 lines of Rust code, compared to approximately 3,100 lines of Standard ML in recent distributions of the Isabelle framework. Whilst comparing linecounts between languages is an imperfect science, the Supervisionary kernel is still clearly larger than one of the most complex extant LCF-style implementations. Largely, this is because Supervisionary implements the HOL Natural Deduction relation in full, providing introduction and elimination rules for all logical constants, as observed in Section 2. Using a bare-bones implementation of HOL, based upon equality, and then deriving introduction and elimination rules for all other logical constants outside of the kernel, would be one way to shrink the kernel line count.

From the point-of-view of a Supervisionary user-space program the kernel is not the only body of code that must be trusted. The implementation of `libsupervisionary` – a mediation layer between kernel and user-space – must also be trusted in much the same way that `libc` must also be implicitly trusted by user-space on Unix-derivatives. A malicious `libsupervisionary` could present as interfacing with the kernel whilst maintaining shadow copies of kernel state, never requesting that the kernel actually check a proof, for example! This threat is not unique to Supervisionary – despite oft-repeated claims that the kernel represents the entire system TCB, users of all interactive theorem proving systems implicitly trust the system's pretty-printer, for example, not to lie about what the system has proved [28], despite this code typically residing outside of the kernel – though the fact that there is a nuanced *threat model* here is perhaps more obvious, and pressing, as a consequence of Supervisionary's dual status as proof-checker and general-purpose virtual machine. Minimising the system kernel size potentially comes at the cost of bloating other code – for example, `libsupervisionary` – that must also be trusted by users wishing to use the system for theorem proving tasks. However, the Supervisionary kernel may also contain functionality – filesystems, timers, network sockets, and similar – related to the Supervisionary general-purpose virtual machine and there is a danger that this code can be used by a malefactor to *exploit* the kernel, perhaps undermining the Supervisionary capability system by somehow deriving a contradiction in an empty context, or similar. On balance – and focussing on security and consistency, rather than efficiency – the kernel size should be minimised if possible, as this prevents security exploits and simply moves code that must be trusted for theorem-proving purposes around, neither helping nor hindering the system's trust story in that respect.

Lastly, the kernel can also be modularised – and is, in the Supervisionary implementation – with all theorem-proving related material isolated inside its own module, all filesystem-related material within its own module, and the two only interacting when strictly needed. Ironically, this re-introduces the idea of protecting key system invariants using programming-language modules and type-abstraction albeit this is never directly exposed to the user.

5.3 Closing remarks

We have presented Supervisionary, a kernel for an implementation of Gordon's HOL. In contrast to most implementations of HOL, Supervisionary is not based on the LCF architectural pattern, but is instead implemented in a style more reminiscent of a typical operating system, making essential use of machine-oriented notions of separation to protect the system kernel from untrusted automation.

The Supervisionary kernel – which is open-source, and developed in the open[9] – is implemented as a host for the Wasmi interpreter[10] for Wasm. Interpretation means that software executing under Supervisionary executes orders of magnitude slower than natively-compiled code. However, the kernel is architected in a layered manner, with all important kernel functionality implemented in a library that is independent of the execution engine used and bound to the execution engine in a thin shim layer sitting between it and the core kernel library. As a result, Supervisionary can be ported to more efficient Wasm execution engines – the Wasmtime just-in-time compiler[11], for example – relatively easily. This porting has already started and will provide a significant increase in system performance, albeit at the cost of bringing a state-of-the-art just-in-time compiler into the kernel.

The design of Supervisionary is interesting in its own right: it completely dispenses with the typical metalanguage associated with an LCF-style proof-assistant, allowing automation to be written in any programming language capable of respecting the Supervisionary kernel binary interface and calling conventions. However, in our view the most interesting aspects of Supervisionary are the consequences of its design, and the possibilities for future work. These include adopting the Supervisionary kernel interface as the foundation of a hardware implementation of HOL – wherein HOL's inference rules are implemented as machine instructions that modify private memories – and the use of Supervisionary as a general-purpose virtual machine that uses its proof-checking abilities to "challenge" user-space programs to explain their adherence to some system-wide security or correctness policy. Notably, by moving this proof-checking capability into the operating system, or other privileged system software, or even hardware, this capability becomes shared by *all* user-space software executing within the system, not just software written in dedicated "verification aware" programming languages. In a sense, mathematical truth becomes just another resource protected by the operating system and system hardware.

References

1 Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. Candle: A verified implementation of HOL light. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 3:1–3:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITP.2022.3.

[9] https://www.github.com/DominicPM/supervisionary
[10] https://github.com/paritytech/wasmi
[11] https://www.wasmtime.dev

2 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.

3 Arm Holdings, Ltd. AArch64 virtual memory system architecture. URL: https://developer.arm.com/documentation/ddi0406/b/System-Level-Architecture/Virtual-Memory-System-Architecture--VMSA-, 2023. Accessed 1st May 2023.

4 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.

5 Edwin C. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.

6 Edwin C. Brady. Idris 2: Quantitative Type Theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECOOP.2021.9.

7 Alonzo Church. A formulation of the Simple Theory of Types. *J. Symb. Log.*, 5(2):56–68, 1940. doi:10.2307/2266170.

8 de Ng Dick Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Studies in logic and the foundations of mathematics*, 133:375–388, 1972.

9 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. doi:10.1145/224056.224076.

10 Michael J. C. Gordon. Introduction to the HOL system. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, August 1991, Davis, California, USA*, pages 2–3. IEEE Computer Society, 1991.

11 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.

12 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.

13 Norman Hardy. KeyKOS architecture. *ACM SIGOPS Oper. Syst. Rev.*, 19(4):8–25, 1985. doi:10.1145/858336.858337.

14 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.

15 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.

16 Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 14–26. ACM, 2001. doi:10.1145/360204.375719.

17 Ralf Jung. *Understanding and evolving the Rust programming language*. PhD thesis, Saarland University, Saarbrücken, Germany, 2020. URL: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647.

18 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.

19 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.

20 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, March 2013. doi:10.1145/2490301.2451167.

21 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 on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2451116.2451167.

22 Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, 1990.

23 Georg Moser and Richard Zach. The epsilon calculus (tutorial). In Matthias Baaz and Johann A. Makowsky, editors, *Computer Science Logic, 17th International Workshop, CSL 2003, 12th Annual Conference of the EACSL, and 8th Kurt Gödel Colloquium, KGC 2003, Vienna, Austria, August 25-30, 2003, Proceedings*, volume 2803 of *Lecture Notes in Computer Science*, page 455. Springer, 2003. doi:10.1007/978-3-540-45220-1_36.

24 George C. Necula. Proof-carrying code. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Ed*, pages 984–986. Springer, 2011. doi:10.1007/978-1-4419-5906-5_864.

25 Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony C. J. Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1003–1020. IEEE, 2020. doi:10.1109/SP40000.2020.00055.

26 Ulf Norell. Interactive programming with dependent types. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA – September 25 – 27, 2013*, pages 1–2. ACM, 2013. doi:10.1145/2500365.2500610.

27 Lawrence C. Paulson, Tobias Nipkow, and Makarius Wenzel. From LCF to Isabelle/HOL. *Form. Asp. Comput.*, 31(6):675–698, December 2019. doi:10.1007/s00165-019-00492-1.

28 Robert Pollack. How to believe a machine-checked proof. In *Twenty Five Years of Constructive Type Theory*. Oxford University Press, October 1998.

29 S. A. Rajunas, Norman Hardy, Allen C. Bomberger, William S. Frantz, and Charles R. Landau. Security in KeyKOS™. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 7-9, 1986*, pages 78–85. IEEE Computer Society, 1986. doi:10.1109/SP.1986.10000.

30 John C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.

31 Amitabha Sanyal and Uday P. Khedker. Garbage collection techniques. In Y. N. Srikant and Priti Shankar, editors, *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*, page 6. CRC Press, 2007.

32 Jonathan S. Shapiro and Norman Hardy. EROS: A principle-driven operating system from the ground up. *IEEE Softw.*, 19(1):26–33, 2002. doi:10.1109/52.976938.

33 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.

34 Antonis Stampoulis and Zhong Shao. VeriML: typed computation of logical terms inside a language with effects. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 333–344. ACM, 2010. doi:10.1145/1863543.1863591.

35 Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems—design and implementation, 3rd Edition.* Pearson Education, 2006.