All watched over by machines of loving grace

Dominic P. Mulligan
Automated Reasoning Group, Amazon Web Services, Cambridge, United Kingdom[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-checkers 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-checker 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*, a prototype programmable proof-checking system for Gordon's HOL that is organized in a manner more reminiscent of operating systems than typical LCF-style proof-checkers. Supervisionary 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 to indirectly manipulate kernel objects managed by the Supervisionary kernel via handles.

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. Indeed, *any* programming language can be used to implement automation for Supervisionary providing the resulting binary respects the Supervisionary kernel calling convention and binary interface, with no risk to system soundness. Moreover, we observe that Supervisionary allows arbitrary programming languages to be endowed with facilities for proof-checking. Indeed, the handles that Supervisionary uses to reference kernel objects under its management may be thought of as a form of *capability*, in the hardware sense. However, unlike typical capabilities, Supervisionary's capabilities are extremely expressive, essentially capturing the full expressive power of HOL, and could be used to enforce fine-grained correctness and security properties 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

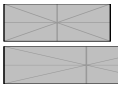
Keywords and phrases Proof assistant design, operating systems, LCF, Supervisionary, capabilities

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

This paper probes the intersection of operating system design and implementations of the foundations of mathematics. Whilst this is, admittedly, a rather moribund research area at the moment, we hope to convince the reader that the intersection of these two areas is of potential great interest by introducing *Supervisionary*, a novel programmable proof-checking system for Gordon's HOL. This system has some interesting properties, and moreover some interesting consequences.

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



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

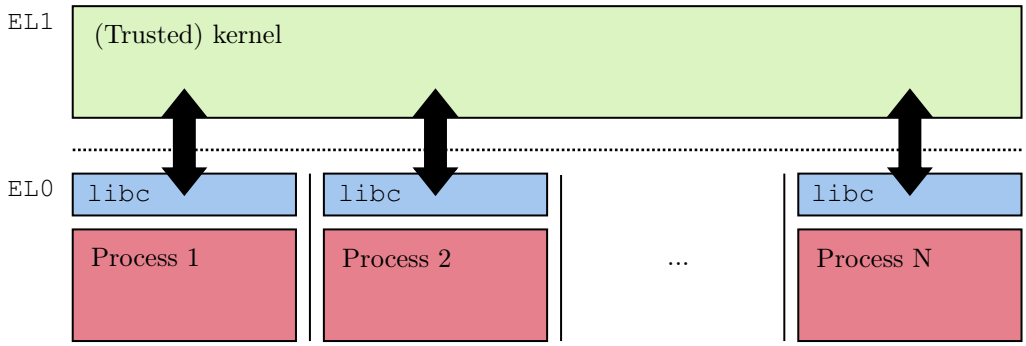


Figure 1 A schematic of the typical system organization of a commodity operating system and its associated user-space. The kernel (in green) executes at a relative level of privilege, enforced by hardware, compared to processes executing in user-space (red)—we follow the Arm convention and show the kernel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

First, however, we begin with a scene-setting overview of common principles in operating system 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*.

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. In addition, the kernel also introduces a process abstraction in user-space and is responsible for ensuring the confidentiality and integrity of processes from other, untrusted processes. 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.

To help the kernel self-isolate modern microprocessors have gradually accreted a number of security features, including *exception levels* or *privilege rings*—as they are variously called, depending on the instruction set architecture—which introduce a notion of *privilege* into the system. Here, software executing at higher-privilege—for example, an operating system kernel—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 ostensibly 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.

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

Moreover, modern microprocessors also provide a form of memory management built around page tables. 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, by presenting a virtual address space to user-space. Essentially, address translation 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 memories, 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 the Figure 1 for a diagrammatic schematic, for example. On Arm platforms, with which the author is most familiar, these instructions induce a type of processor exception, inducing 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 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 and how errors are reported back to user-space, and similar. 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`. However, note that this is generally just a convenience, and user-space software can always invoke system calls directly if wanted by invoking the correct machine instruction and adhering to the appropriate calling convention.[3]

However, crucially, it is *generally* not the case that the operating system kernel and untrusted user-space applications need be written in the same programming language for this all to work. In particular, 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

[3] Note this is the case on Linux but 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.

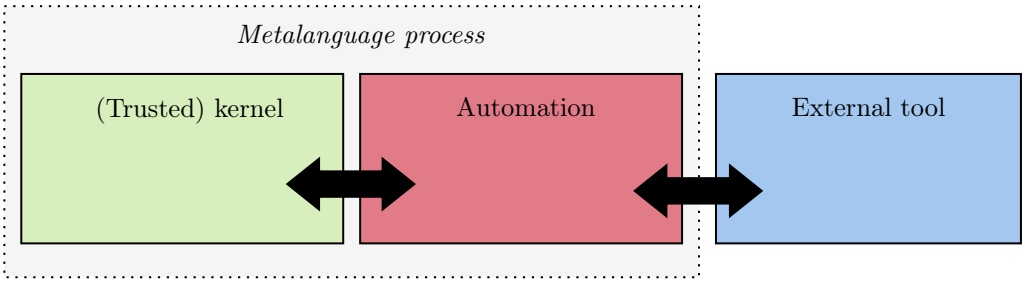


Figure 2 A schematic of the system organisation of a typical LCF-style proof assistant. The trusted kernel (green) is linked against untrusted automation (red) existing within the same metalanguage process (dotted line) and communicate with each other using the kernel's API (black arrow). External tools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

binary interface expected by the kernel. In this respect, for commodity operating systems, the C-language may have prominence as the 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, Coq, Matita, PRL, 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 method of system organisation is known as *the LCF approach* after Milner's eponymous system which introduced it, and is now the most common way of organising proof-checking systems today—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, 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.

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 written in that language, like any other. Whilst external tools, and automation written in other languages, 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

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. 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[4] 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.[5] Whilst the design of this system will be explained in detail in Section **??**, we note here that Supervisionary's system design has more in common with the typical system organisation of an 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 in something akin to Supervisionary's version of "user space". The trusted kernel, and untrusted user space, communicate across a system call boundary, and which must be carefully designed in order to ensure 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 may be thought of as somewhat akin to pointers into Supervisionary's private memories, or analogous to file handles in typical systems software, are used by Supervisionary's user-space to identify kernel objects to manipulate or query.

Remarkably, 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 *unsafe* systems programming language, Rust. 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 and private memories, and not on programming language features. 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

[4] Barring unikernels, or library operating systems, like Mirage, which are in some respects quite similar to LCF-style proof-assistants in this regard, having their kernel modules linked with untrusted "user-space" and separated using programming language features, rather than privilege.

[5] Note that many of the ideas presented henceforth are logic-independent, and though we have chosen to use HOL in our prototype, the ideas presented herein can be applied to a wide variety of other logics with relatively minimal changes.

with the Supervisionary kernel. Supervisionary therefore has no "metalanguage" in the sense understood by users of LCF-style provers, but rather an implementation language, with automation potentially written in multiple languages.

For ease of implementation (and use!) we implement Supervisionary as a WebAssembly (or Wasm, henceforth) host. This allows us to 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. Moreover, we harness work on porting compiler and linker toolchains, allowing our "user space" to be written in any language capable of targeting Wasm.

Lastly—and more speculatively—Supervisionary's handles can be passed around within a program, or even between user-space programs executing concurrently or sequentially under Supervisionary's management. Whilst this is not unique to Supervisionary—values of the abstract type of theorems can also be passed around within an LCF-style system—what *is* unique to Supervisionary is that the objects which these handles denote can themselves be *functions* of the runtime state of the program itself, or of the Supervisionary kernel.

For one example of where this may be useful, consider an extension of Supervisionary with an extended system call interface, including system calls for filesystem manipulation. Prior to opening or modifying a file, Supervisionary could force the user-space program to produce a handle denoting a theorem object whose statement corresponds to some security or correctness policy enforced by Supervisionary. In particular, the statement of these theorems can be parametric in the runtime state of the kernel, or of the arguments passed to the system call (the filename of the file to be opened or modified, for example), or the content to be written to the file, and so on. With this, Supervisionary could enforce a wide range of security and correctness policies, which will further be explored in Section 4.

2 Implemented logic

Supervisionary implements a variant of Gordon's HOL, a classical higher-order logic which can be intuitively understood as Church's Simple Theory of Types 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 set of *type variables* and use α, β, γ, and so on, to range arbitrarily over them. With these, we work with a grammar of *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 their expected number of types 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 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—essentially built-in to the logic itself, and necessary to bootstrap the rest of the logic: bool, the type-former of the Boolean type (and in HOL, also 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,

$$\frac{(r : \tau)}{\Gamma \vdash r = r} \qquad \frac{\Gamma \vdash r = s}{\Gamma \vdash s = r} \qquad \frac{\Gamma \vdash r = s \quad \Gamma \vdash s = t}{\Gamma \vdash r = t} \qquad \frac{(\phi \in \Gamma)}{\Gamma \vdash \phi} \qquad \frac{\Gamma \vdash \bot \quad (\phi : \mathsf{bool})}{\Gamma \vdash \phi}$$

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

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

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

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

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

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

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

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

and 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 we omit here, though we write $r : \tau$ to assert that term r has type τ. We call any term with a type *well-typed*. We will only ever work with well-typed terms in Supervisionary, and call terms with type bool *formulae*. We will use ϕ, ψ, ξ, and so on, to suggestively range over terms that should be understood as being formulae. We work with terms up-to α-equivalence, write $fv(r)$ for the set of *free variables* appearing within 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 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		$(\alpha \Rightarrow \mathsf{bool}) \Rightarrow \alpha$

The majority of the constants above correspond to the usual logical constants and connectives of first- or higher-order logic and are introduced without further explanation. Only the latter

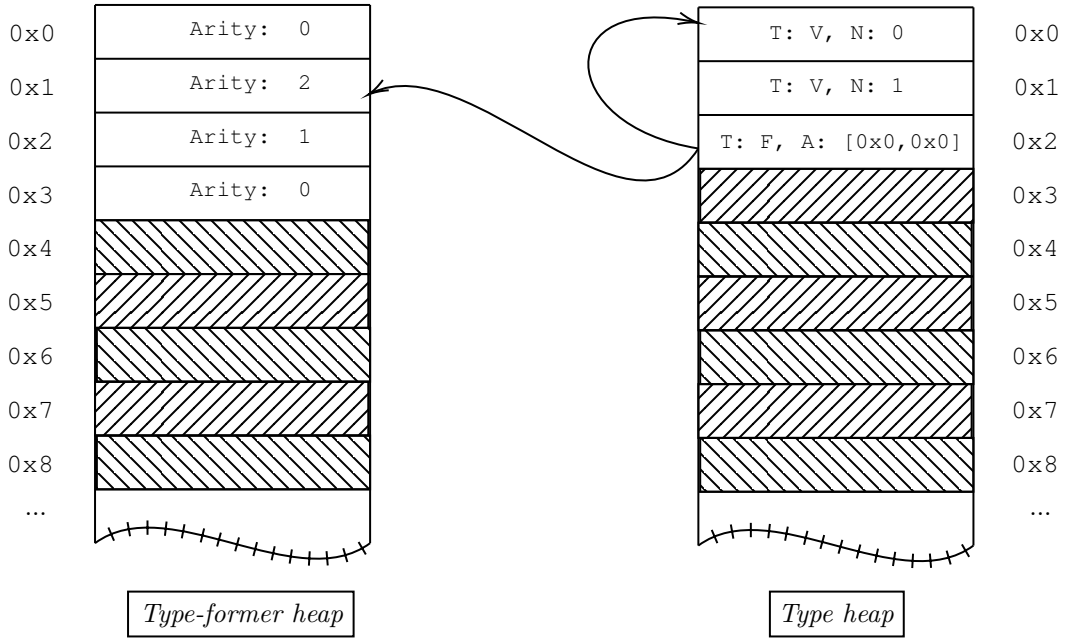


Figure 4 Entries within the Supervisionary kernel's type heap referencing entries within the type-former heap. Cross-hatched heap cells are as-yet unallocated by the kernel. The cell allocated at address `0x2` in the type heap is tagged with the `F` tag, indicating it is a type-former applied to a list 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 $\alpha \Rightarrow \alpha$.

ϵ 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 that all HOL types are inhabited by at least one element. We adopt usual mathematical conventions and precedence levels when writing terms making use of these constants, writing $\phi \longrightarrow \psi$ instead of $(\longrightarrow\ \phi)\psi$, for example. We also suppress explicit type substitutions required to make terms involving polymorphic types well-typed, writing $\forall x_\tau.\phi$ instead of $\forall[\alpha := \tau](\lambda x_\tau.\phi)$, for example.

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 two-place *Natural Deduction relation* between 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. We prefer a more explicit treatment.

3 The kernel state

3.1 The type-former heap

Supervisionary's kernel manages a series of *heaps*, or private memories. These heaps contain different *kernel objects*, of various kinds: type-formers, types, constants, terms, and theorems.

The most foundational of all of the heaps is the heap of type-formers. 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. 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—perhaps to query the arity of the type-former stored at a particular address in Supervisionary's heap, via the `TypeFormer.Arity` system call, for example, which exists for this purpose and takes a handle as input, returning either an arity or a defined error code back to user-space, indicating that the input handle is *dangling*, and cannot be *dereferenced* to obtain an arity.

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 Supervisionary kernel. Yet, there are exceptions to this rule: two primitive type-formers are pre-registered within the type-former heap on system boot: the bool type-former, registered at address `0x0` and 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 Supervisionary system interface, in a similar vein to how the distinguished file handles `stdout` and `stdin` are part of the POSIX system interface, too.

3.2 The type heap

Building atop the heap of type-formers is another heap: the heap of types. Referring back to the grammar of HOL types introduced in Subsection **??**, all allocated entries within this heap are tagged either with a `V`, indicating that they are a type-variable, or with an `F`, indicating that they are a "combination" of a type-former applied to a list of argument types.

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.

Like the type-former heap, 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, $\alpha \Rightarrow \alpha \Rightarrow \mathsf{bool}$. Again, the handles for all of these pre-registered entities must be understood by user-space.

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 (by e.g., modifying the bool type, or the truth constant, $\top$, or similar). 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 Subsection **??**, with larger objects being gradually "built up" out of smaller ones.

Note that the Supervisionary ABI for working with types is much more well-developed than for type-formers. In addition to basic system calls for registering new types with the kernel, Supervisionary also provides a suite of more complex system calls for computing the variables appearing in a type, performing a type-substitution on a type, and similar. This latter call—`Type.Substitute`—is interesting for two reasons: first, as mentioned above, once kernel objects are registered in a heap they are immutable. As a result, this system call allocates a new type, the result of the substitution, in the type heap upon success. Moreover, the substitution to be applied to the type can be arbitrarily large, consisting of an unbound number of type variable-to-type mappings. As a result, this system call takes a pair of base pointers and a pair of lengths, one each for a buffer of variables and a buffer of handles pointing-to types in the type heap. These base pointers point-into a buffer in the calling user-space program's memory space, which the Supervisionary kernel reads to

3.3 Specifying kernel functions

3.4 Programming the kernel

4 Capabilities on steroids

5 Conclusions

5.1 Related work

5.2 Future work

Aside from the inherent amusement in structuring a proof-assistant in this way, Supervisionary also opens up several interesting new lines of future work.

Hardware-accelerated proof-checking

As mentioned earlier, from the perspective of user-space software an operating system's system call interface presents as a suite of particularly CISC-like machine instructions with a rather strange 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 new instructions for proof construction and management.

As a result, note that the Supervisionary system call interface is already particularly well-suited to being implemented as an extension to the Arm instruction set or RISC-V instruction sets, for example. Already, the mechanism through which Supervisionary isolates itself, via private memories, is rather "hardware like", and maps nicely onto existing hardware features. 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. 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 to the kernel. Rather, kernel objects are gradually built up out of smaller, already-existing objects. We believe that this idea maps nicely onto an instruction set architecture, and implementing a hardware-based proof-checker for HOL using the ideas in Supervisionary is future work.

Transferring theorems between systems

Consider compiling another implementation of HOL—for example, HOL Light—into Wasm. The Supervisionary implementation of HOL is sufficiently similar that it should be possible to "rip out" the HOL Light kernel and replace it with a shim layer that exposes the same interface but simply calls into Supervisionary's system calls to implement kernel functionality, with some extra internal book-keeping for handle management. From this, one can immediately "import" the entire HOL Light library directly into Supervisionary, merely by having HOL Light bootstrap itself.

This approach can be used to transfer results from one HOL implementation to another. In particular, consider doing the same kernel shim modification with *another* implementation of HOL, for example, HOL4, written in a different programming language to HOL Light. This too, can also be compiled to Wasm, and can bootstrap itself using Supervisionary. Now, the HOL4 kernel is "sitting on top" of a series of heaps populated already by HOL Light, and can make reference to these existing results, or build on top of them, if desired.

Note that this provides an alternative method of transferring results between implementations of HOL, in contrast to existing approaches like OpenTheory. All results, independent of which system ultimately generated them, are derived within a single, common logical system implemented by Supervisionary. Essentially, the two systems makes use of Supervisionary's heaps as their joint "source of truth" to transfer results.

Enforcing runtime correctness and security properties