Experience Report: seL4

Formally Verifying a High-Performance Microkernel

Gerwin Klein
NICTA and University of NSW
gerwin.klein@nicta.com.au

Philip Derrin
NICTA
philip.derrin@nicta.com.au

Kevin Elphinstone
NICTA and University of NSW
kevin.elphinstone@nicta.com.au

Abstract

We report on our experience using Haskell as an executable specification language in the formal verification of the seL4 microkernel. The verification connects an abstract operational specification in the theorem prover Isabelle/HOL to a C implementation of the microkernel. We describe how this project differs from other efforts, and examine the effect of using Haskell in a large-scale formal verification. The kernel comprises 8,700 lines of C code; the verification more than 150,000 lines of proof script.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.1.1 [*Programming Techniques*]: Functional Programming; D.4.5 [*Operating Systems*]: Reliability—Verification

General Terms Verification, Design, Languages

Keywords Haskell, seL4, microkernel, Isabelle/HOL

1. Introduction

We report on our experience using the functional programming language Haskell in the formal verification of the seL4 microkernel (Elphinstone et al. 2007). The seL4 kernel is an evolution of the high-performance L4 microkernel family (Liedtke 1995) for secure, embedded devices. It provides essential operating system services such as threads, inter-process communication, virtual memory, interrupts, and authorisation via capabilities. In earlier work (Derrin et al. 2006), we reported on our experience with Haskell as a specification language for seL4. In this paper, we concentrate on the effect our choice of Haskell had on the formal verification of the kernel, from abstract operational specification down to high-performance C code. To our knowledge this is the first large-scale formal verification project that employs Haskell (or any other functional programming language) in this way.[1]

We found that working with Haskell decreased our kernel design time, enabled an iterative prototyping process in an area where usually only top-down and bottom-up approaches are advocated, and made formal verification towards abstract and concrete levels substantially easier and faster than they would have been otherwise.

[1] The ACL2 prover uses LISP as its formal language. Our use of Haskell differs in the sense that our executable kernel prototype in Haskell is an independent program that can stand on its own without theorem prover involvement.

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-332-7/09/08. . . $5.00

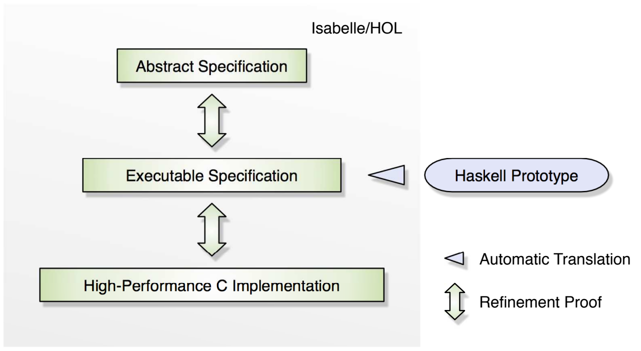

Figure 1. Specification layers in the L4.verified project.

The basic structure of the verification project is shown in Fig. 1. The left-hand side follows the classic pattern of a traditional refinement. There is an abstract specification at the top, an executable specification in the middle, and a C implementation on the bottom. Elkaduwe et al. (2008) have also created an even more abstract security model with security proof that would be placed above the abstract specification, but it has not yet been formally connected with the rest of the stack. In the setting of a commercial Common Criteria evaluation, the abstract specification is the high-level design and the executable specification is the low-level design. Cock et al. (2008) present details on the proof between abstract and executable level; the proof between executable specification and implementation will appear elsewhere (Winwood et al. 2009).

While neither the main property (functional correctness) nor the main proof methodology (refinement) are unusual, the size and scope of the project are. The verification does not stop at a specification, but descends to the implementation level: 8,700 lines of high-performance, manually tuned C code close to hardware. All proofs in this project are machine-checked in the interactive theorem prover Isabelle/HOL (Nipkow et al. 2002).

The project is also unusual in the approach it takes to kernel design and implementation. Two teams were involved: a kernel design team with an operating systems background, and a verification team with a formal methods background. The right-hand side of Fig. 1 indicates that the executable specification of seL4 is produced from a kernel prototype written in Haskell. We have implemented an automatic translator that converts our subset of Haskell into Isabelle/HOL. The Haskell prototype is written and maintained by the design team. It is the principal embodiment of their design decisions. It also became, after automatic translation, the starting point for the verification effort. The traditional model for greenfield projects is to work top-down from a high-level specification, and for existing implementations to work bottom-up from the implementation level; beginning verification with an executable specification is unusual. The effects of this choice on the verification are discussed below.

The abstract specification and the C implementation were developed manually; both were started before the design — and therefore the executable specification — was completely stable. Both activities fed changes back into the Haskell prototype, but they did not supersede it. The Haskell prototype remained the central reference model throughout the duration of the project.

2. Executable and Abstract Specification

Microkernels in the L4 family share a number of basic design principles, set out by Liedtke (1995). They provide only the abstractions that are essential for performance or security — primarily virtual memory, threads, and inter-process communication. They are designed with an emphasis on IPC performance, which is critical to the overall performance of microkernel-based systems. They have, historically, provided only minimal support for managing kernel resources and for controlling access to communication.

The seL4 project set out to design a new microkernel based on the same basic principles, but taking a new approach with capability-based resource management and access control. Since this entailed designing a new API that was significantly different to that of previous L4 kernels, there was a large design space to explore.

We decided, at the outset, that implementing a new kernel in C from scratch when the design was still uncertain was a risky proposition; much time would be wasted on rewrites and low-level, hardware-dependent debugging before a final design emerged. On the other hand, designing a kernel formally on paper before implementing it might result in an API with limited application to real-world use. We wanted to be able to execute user-level programs to test our proposed designs. Also, since we intended to formally verify safety properties of the kernel (Tuch et al. 2005), we desired a precise specification with well-defined semantics.

Thus, we developed an executable model of the design in Haskell. This model was gradually developed into a complete prototype, exploring various design alternatives in the process (see Derrin et al. 2006). We were able to exercise user-level programs on the binary level by attaching a processor and platform simulation to the Haskell prototype. At the same time, we formalised the Haskell prototype by translating it into Isabelle/HOL; the translation was initially performed by hand, but was later automated.

When, after a number of iterations, the kernel design in Haskell had begun to stabilise, we constructed an abstract, operational specification with fewer data structure details, and with features like scheduling underspecified so that different implementation choices could be explored in later versions of the kernel. The abstract specification is meant to specify what the kernel does; the executable specification gives details on how it is done. This initial abstract formalisation process provided immediate feedback on correctness and safety to the design team. The feedback increased when we started the refinement proof between the two layers (see Sect. 4).

Isabelle/HOL is based on lambda calculus, and can be seen as a functional programming language extended with logical operators. It is less expressive than Haskell in some ways: every function must terminate, which limits use of laziness; type classes cannot have multiple parameters, and there are no constructor type class, so there is no built-in Monad typeclass, nor was there initially do-syntax for lists of sequential operations. However, Isabelle's syntax is easily extensible, and we were able to define our own do-syntax for the specific monads used by our abstract and executable models. Unsurprisingly, the Isabelle/HOL standard library is geared towards theorem proving and felt therefore limited for implementing a large-scale functional program. Again, this was easily extended. We were able to implement all of the monad and list functions that we used of the Haskell library in Isabelle.

Proving termination for every function was less difficult than we anticipated — for primitive recursion and for functions with a simple lexicographic termination measure the proofs are straightforward and in many cases entirely automatic. However, we avoided complex recursion patterns, such as nested mutual recursion, because they would have been more difficult to translate. This was also desirable because the microkernel has to operate on strictly limited and known stack depth, so recursion ultimately had to be implemented by loops in C anyway.

We used only a constrained subset of Haskell that could be translated to Isabelle/HOL. Besides forgoing the use of laziness in any essential way, we made limited use of type classes in Haskell (in particular using only three specific instances of MONAD m), and avoided most GHC extensions.

The use of Haskell at this stage had two main effects on the verification component of the project. First, it removed the need to interpret vague and inaccurate natural language design specifications, user manuals, or incomprehensible optimised C code. Second, it constrained the design team to a subset of Haskell that could be handled by the automatic translation, leading them to instinctively favour designs suitable for formal specification. Since extending the subset had an obvious cost in terms of modifications to the translator, there was a natural counter-force to increasing this subset too much. The question *How would we write this in Haskell?*, and therefore *How can it be formalised?*, was a topic in design team discussions. In a normal kernel design process, it would not have been.

3. High Performance C Implementation

The goals of the Haskell prototype were twofold: predictable behaviour to provide an easy path to formalisation, and enough detail to provide an easy path to a C implementation.

The second requirement, especially, led to an imperative-style Haskell program with extensive use of the StateT and ErrorT monads, including an explicit model of kernel memory addressed by typed pointers. An explicit hardware interface made it easier to connect the prototype to different simulators (M5, qemu, and our own ARMv6 instruction simulator). This interface also became the machine interface of the C kernel.

As performance tuning is essential for microkernels, we did not attempt to generate C code from the model, but implemented the kernel manually, following the structure of the executable specification closely. The direct C implementation work was roughly 2 person months in effort, which is insignificant compared to the 20 person years spent on the complete project. The extremely rapid manual implementation was possible thanks to the precise executable specification. Not many implementation choices had to be made, and the structure of the program was clearly laid out already. D. Wheeler's SLOCCount estimates that the effort for implementing the kernel directly in C would have been 4 person years. The effort for designing, writing and documenting the Haskell prototype was *ca* 2 person years. Based on this estimate, the use of Haskell reduced the implementation effort by 50%.

In the first implementation pass, we did not pay any attention to performance. The result of this initial pass was therefore unsurprisingly slow (on the order of the Mach microkernel), a factor of 3 slower than comparable operations in existing L4 kernels. After a first round of manual optimisations, seL4 IPC performance is now comparable to OKL4 2.1 (2008) on ARMv6.

Another consequence of using a functional language as the design source was the structured use of tagged unions in the C code. Verification of unstructured unions in C is unpleasant. Since unions and structs were used in a principled way, we managed to avoid this additional verification burden entirely. Moreover, we did not trust the compiler to translate C bitfields correctly and with the fine-grained control we required; instead, we generated the C code for these structures and tagged unions automatically from a separate specification language (Cock 2008). We also generated the corresponding Isabelle/HOL proof of code correctness.

Our verification framework treats a large, true subset of C99. The main restrictions are: we do not allow the address-of (&) operator on local variables, because the stack is modelled separately from the heap; we do not allow function pointers and goto statements; we make some expressions such as `x++` statement forms; and we allow at most one side-effecting sub-expression in any assignment, because execution order is arbitrary otherwise. The function pointer prohibition implies that we did not make heavy use of higher-order functions in Haskell apart from some specific functions that are used to emulate C control structures such as *mapM_*, *zipWithM*, *catchError*, and so on. This prohibition could be lifted fairly easily from the C translation, but we found it advantageous to strive for simplicity over features when possible in our Hoare logic framework for C.

We translate C types precisely into Isabelle/HOL, including pointers, address arithmetic, finite integers, structs, and padding in structs (Tuch et al. 2007). Our target architecture is ARMv6; the compiler is GCC 4.2.2. Strictly speaking, Fig. 1 is inaccurate in that we do not reason on C directly, but on a translation of C into Isabelle/HOL. In contrast to the Haskell/Isabelle translation, this is a comparatively small translation step, with explicit care taken to map the semantics of C precisely into the theorem prover.

Although our experience was in general favourable, efficient kernel code does not always translate well from Haskell. For example, the executable model's error handling code contains a function that loads a message into a user-level context (*setMRs*), which is applied to the results of one of several functions that generate messages (as lists of machine words) from various error types. Translating this directly to C leads to implicit allocation of memory to temporarily hold the message, and double copying of the message's contents; in order to keep the C code efficient, we manually unfolded the definition of *setMRs* and fused it with the message generation functions before translating. More generally, we found that Haskell at times encouraged coding practices that are inefficient in our subset of C if translated naively: passing large structures as function arguments, throwing and catching exceptions for error handling, and function composition that depends on laziness to be efficient.

An interesting observation on the C implementation was that the C program was in parts less verbose than the explicit memory model we used in Haskell, because load-check-modify-store idioms are simply written as pointer accesses and updates in C. The more verbose style for this part in Haskell did not hinder verification. On the contrary, pointers made up a large part of the hidden complexity of the C program that was dealt with explicitly in the executable and abstract models. The additional verbosity was local only. In total, the Haskell prototype comes to 5,700 LOC compared to 8,700 LOC in C (numbers according to SLOCCount).

To summarise, we restricted our use of Haskell to a suitable subset and were able to manually implement a high-performance C version of the kernel in very little time. The Haskell and C versions have almost identical data and code structures. We exploited this fact heavily in the verification.

4. Formal Verification

As mentioned in the introduction, the formal verification of seL4 consisted of two major refinement steps: between abstract and executable specification, and between executable specification and implementation. Our embedding of Haskell into Isabelle is shallow, the embedding of C into Isabelle is deep for statements and shallow for expressions.

The main statement we proved in each of the two steps is formal refinement, reduced to forward simulation: if the initial states are in a system-global state relation R, and the concrete level takes one step, then the abstract level must be able to take a corresponding step such that the resulting states again are in the relation R. Cock et al. (2008) extend this classic notion to state monads, integrating the aspects of failure, non-determinism and exceptions needed in the kernel specifications. The notion implies, and we have proved in Isabelle/HOL, that all Hoare triples that are true on the abstract level are also true on the concrete level, modulo the state relation R.

4.1 First Refinement Step

Refinement step one in the verification took *ca* 8 person years in total and manually produced 117,000 lines of Isabelle/HOL proof script. This step contains the conceptually interesting part of the proof, reasoning about the design aspects of execution safety and correctness. We cannot go into the details of this proof here for space reasons, but the simpler and higher-level data structures of the abstract specification require invariants on their more detailed counterparts on the executable level to show correspondence. Basic preconditions of the correspondence proof are that each operation is well defined, that memory accesses are correctly typed, that assertions do not fail, that objects that are read from do exist, and that partially defined functions (e.g. those with incomplete patterns in Haskell) are used only within their domain. These preconditions for safe execution spawned a number of complex invariants on how the kernel works, how it explicitly re-uses memory, and how it prevents dangling references to deleted objects in any part of the kernel (including all of memory). Reasoning on this level included explicit decoding of binary system call arguments read from user registers and full argument checking to ensure safe operation for any kernel input, be it benign, maliciously crafted, or simply garbage.

The effect our use of Haskell had on this proof can be summarised as: the ability to exploit structural similarities, an increased use of library functions, initial increased technical friction in working with generated definitions, and different proof style. We explain each of these in more detail in the following paragraphs.

The Haskell prototype existed first, and therefore the abstract specification was inspired by it in structure. We were able to exploit this structural similarity to make the proof easier. Being inspired by the executable level also means that our abstract specification is probably more concrete than it may have been without this input. A higher-level abstract specification would have meant more distance for the refinement proof to the executable level, but possibly less distance to further layers above. For showing specific properties of the kernel that turn out to be too complex for the complete abstract operational specification, we would add a further, more abstract layer to the stack that is specialised to the property under consideration, such as we are currently exploring with the abstract access control model of seL4 (Elkaduwe et al. 2008).

Haskell being a fully-featured programming language led the design team to make more extensive use of library functions like mapM and zipWithM than they otherwise may have. At the time, there were no Isabelle versions of these functions. Introducing them saved verification work because we avoided repeating proofs over many similar recursion patterns.

On the less positive side, we observed more technical friction in the proofs that were concerned with definitions generated from Haskell than in those that were written in Isabelle directly. This was expected. Programming idioms did not always match up with how rules were phrased in the Isabelle library. The executable specification was generated Isabelle code that was not as concise as the Haskell source, and not always nice to read. This turned out to be an initial problem only. The new idioms became manageable once the verification team were used to them, and had built up a library of matching rules. The generated code could often be rewritten trivially with associativity and other simple, general state-monad laws to read more nicely.

Due to the monadic, imperative style of the Haskell prototype and therefore the executable specification, the majority of the proof took the form of Hoare triples, weakest precondition reasoning, and correspondence calculus reasoning. Apart from the rewrites mentioned above we used only little of the algebraic reasoning that would usually be associated with verifying functional programs. This proof structure is mainly an artefact of the application area and of having C as a target implementation language. We did use induction where recursion was involved in the Haskell program. In C, these were replaced by loops.

This first proof lead to around 200 changes in the Haskell prototype and 300 changes in the abstract specification. Less than half of these were genuine bugs or design defects. Most changes were for proof convenience: reshuffling functions to match up more closely, adding assertions to transport information across levels, and adding local checks or re-arranging code to make properties more obviously true. The majority of the bugs we found during verification were mundane: simple typos and some copy & paste errors. We did also find more subtle problems in the initial design like missing argument checking, potential security violations etc. Of course, it does not matter if the defect is mundane or not: the kernel will happily crash or allow a security attack either way. It is interesting to note that the actual discovery of defects does not necessarily occur when they leap from the screen in the form of a counter example or unprovable lemma (although that did happen). Instead, many defects were found when new invariants became necessary and the verification and design teams discussed what these should be, whether they would hold, and, if so, why. One answer a verification team should be wary of is *this is never done anywhere in the kernel*. This answer usually means proving a new fact about the whole kernel instead of quick local reasoning.

It is not clear if the use of Haskell would have been beneficial in just the proof of the first refinement step in isolation. Nicer, more elegant reasoning might have been possible in a more abstract, still executable setting with definitions written directly in Isabelle/HOL. The detailed executable model, and its use as a prototype running user binaries, injected a sometimes unpleasant dose of realism into the proof — forcing us to consider implementation details that are necessary for an *efficient* kernel, rather than one that simply functions correctly. For example, one part of the first refinement proof that was particularly challenging was the relation between the abstract and executable models' versions of the *capability derivation tree*. This is one of the two main metadata structures in the seL4 implementation; it is conceptually a forest of directed trees, and is represented in the abstract model by simple functions that encode a binary relation between pointers to the nodes. The executable model represents it the way a real kernel implementation would: as a set of doubly linked lists, each corresponding to a pre-order traversal of a tree. The depth information is implicitly encoded in the nodes, and is available only by comparing two nodes; the depth comparison function requires that its arguments are in the stored order. Furthermore, the lists are not represented by Haskell's standard list type, but by pairs of pointers stored in separate node objects in the modelled physical memory — as they would be in C. Naturally, the invariants that must be maintained by operations on these structures are complex, and therefore so is the refinement proof for those operations.

This realism paid off in the second refinement step.

4.2 Second Refinement Step

One important observation about the first refinement step is that we spent roughly 80% of the proof effort on showing invariants of the abstract and executable levels and only 20% on the correspondence itself. The invariants were necessary preconditions for the correspondence, but they also carry a large amount of information on how precisely the kernel works and why its internal data structures are safe to use.

Because the Haskell and C implementations share almost identical data and code structures, we were able to avoid these 80% for the second step. The important invariants had already been proved on the executable specification level, and no complex semantic reasoning was necessary on the C level. The most complex new relationships we had to show on the C level were the implementation of Haskell or Isabelle lists as doubly linked lists, some of which were encoded in existing data structures. The C verification did lead us to prove new invariants on the level of the executable specification, but far fewer than we needed in the first step. They were mainly due to optimisations in C that made use of conditions known to be true over kernel execution.

The main challenges in the second step were dealing with C language semantics and data structure encodings, but without complex data refinement. Having to prove at the same time that the C code maintains complex invariants as we had shown in step one would have made this proof much harder. At the time of writing, the proof on the C level is completed for 474 functions out of 518 and we have so far spent 26 person months on this part of the verification. The speed of verification on this level was 3–4 functions per person per week with 3–5 persons working on this body of proofs concurrently.

Even though the kernel implementation had in the meantime been used in a number of small student projects, had been ported to the x86 architecture, and had been run through static analysis tools, we still found 97 defects in C during the verification. We had not attempted to test the implementation in great detail, because formal verification was scheduled anyway. For each of the defects we could have found a test case that demonstrated it, but of course the question is whether we would have thought of these beforehand. Unsurprisingly, the defects were concentrated in parts of the kernel that were less used in the student projects and that were complicated to use. Most of them were simple translation errors and typos in the implementation step from Haskell to C, fewer were defects in new data encodings and optimisation. We also observed compiler specific errors: for instance, some functions that we had annotated with GCC's pure and const attributes to enable optimisations were not in fact pure or const. The compiler did not check the attributes, and neither did we initially in the verification. This lead to unexpected execution behaviour in otherwise already verified code. We have updated the verification framework in the meantime to include such compiler hints and make them proof obligations. The pure and const attributes are now checked automatically.

As in the first refinement step, it was crucial for the verification that we were able to change the C code as well as the Haskell source for proof convenience instead of having to prove complex reordering theorems. For some optimisations, we changed the observable behaviour of both the abstract and the executable specifications. For instance, we changed the order in which data was stored in global data structures, or the order in which arguments were checked (and therefore which error messages would be reported first).

In summary, the verification of seL4 proceeded in two main steps. Step one dealt with mostly semantic content in a shallow embedding; step two was more syntactic and dealt with C, its memory model and specific optimisations. We were able to avoid a large part of the proof in the second step, because of the structural similarity between the C and the Haskell implementations.

5. Conclusion

We have presented our experience in using Haskell in the verification of the seL4 microkernel. The aspects of the verification that are specific to this project are its size, the implementation level it descends to, and its iterative development cycle. As there is not sufficient space to survey related work in this article we refer to the comprehensive overview by Klein (2009).

We consider it important for the success of this project that the kernel was designed by a team with an OS background, not by the verification team. The verification team believes it would have designed a much more elegant, but much more useless microkernel. The connection between the two teams was the Haskell prototype. All of the verification and implementation activities fed back into this central reference specification of the project.

We could have created the executable specification in Isabelle directly, but that would have left the design team out of the loop. We could also have chosen another functional language, such as ML, rather than Haskell; the primary motivation for our choice was the local availability of experienced Haskell programmers at UNSW, where Haskell is used in several research projects and was used as the introductory undergraduate programming language at the time. In addition, we consider the extensive tool-chain support for Haskell (compiler, foreign function interface, literate Haskell) an important contributor to the success of the Haskell source as a simultaneous binary-compatible prototype, design document, and formal executable specification. The part of the OS team that actively wrote the Haskell code had previous experience with Haskell. The part of the OS team that did not have extensive experience with Haskell was comfortable with the new language after less than one month.

The culture shock between the Formal Methods and Operating Systems groups was smaller than expected and greatly alleviated by team members who had gone through advanced courses in both areas.

Will this approach work for everything? We believe that for high assurance on the level of kernel and systems code where performance and hardware interaction are important, the approach will work well. On the application level, it might be sufficient to stop at the level of an executable specification, possibly in Haskell or ML, if the compiler, runtime, and translation into Isabelle can be trusted.

Acknowledgments

We thank Timothy Bourke, Michael Norrish, and Thomas Sewell for reading and discussing drafts of this article.

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

D. Cock. Bitfields and tagged unions in C: Verification through automatic generation. In B. Beckert and G. Klein, editors, *Proceedings of the 5th International Verification Workshop (VERIFY'08)*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Sydney, Australia, Aug 2008.

D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182, Montreal, Canada, Aug 2008. Springer.

P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *ACM SIGPLAN Haskell WS*, Portland, OR, USA, Sep 2006.

D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In J. Woodcock and N. Shankar, editors, *VSTTE 2008 — Verified Softw.: Theories, Tools & Experiments*, volume 5295 of *LNCS*, pages 99–114, Toronto, Canada, 2008. Springer.

K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *11th HotOS*, pages 117–122, 2007.

G. Klein. Operating system verification — an overview. *Sādhanā*, 34(1): 27–69, Feb 2009.

J. Liedtke. On μ-kernel construction. In *15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.

T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

Open Kernel Labs. OKL4 v2.1. `http://www.ok-labs.com`, 2008.

H. Tuch, G. Klein, and G. Heiser. OS verification — now! In *10th HotOS*, pages 7–12, Santa Fe, NM, USA, Jun 2005. USENIX.

H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *34th POPL*, pages 97–108, 2007.

S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap: A verification framework for low-level C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, volume 5674 of *LNCS*. Springer, 2009. To appear.