Formalising a High-Performance Microkernel

Kevin Elphinstone Gerwin Klein Rafal Kolanski

National ICT Australia *, Sydney, Australia
School of Computer Science and Engineering, UNSW, Sydney, Australia
{kevin.elphinstone|gerwin.klein|rafal.kolanski}@nicta.com.au

Abstract

This paper argues that a pragmatic approach is needed for integrating design and formalisation of complex systems. We report on our approach to designing the seL4 operating system microkernel API and its formalisation in Isabelle/HOL. The formalisation consists of the systematic translation of significant parts of the functional programming language Haskell into Isabelle/HOL, including monad-based code. We give an account of the experience, decisions and outcomes in this translation as well as the technical problems we encountered together with our solutions. The longer-term goal is to demonstrate that formalisation and verification of a large, complex, OS-level code base is feasible with current tools and methods and is in the order of magnitude of traditional development cost.

1. Introduction

Sometimes an incomplete engineering approach is better than a complete, precise mathematical solution. As for normal software, so also for formalisation and verification, seeking the perfect solution to a problem is at odds with the reality of limited development costs.

The overall aim of our project is to design and verify a microkernel-based operating system (Sect. 2). In this paper we argue that a pragmatic approach is essential for large-scale projects such as operating system (OS) verification. We aim to be pragmatic in the sense that we are using a method that on first sight is not suitable, because it will not work in general, because it does not provide a complete solution to the problem, and because it is not fully automatic where in theory it could be. Instead it is semi-automated, systematic, cheap, easy to employ, and still gives the desired result.

Another of our overall goals is to demonstrate that formalisation and verification of a large, complex OS-level code base is feasible with current tools and methods and that the cost of this is in the same order of magnitude as traditional development cost.

Our methodology is to develop an executable OS prototype in the high-level programming language Haskell (done by the OS team), then translate it to a formal specification (done by the theorem proving team). The result will be the basis for refinement into a high-performance C implementation of the OS, as well as the basis for a further abstraction to verify security properties.

This paper focuses on the techniques employed to achieve a practical translation process, the observations and lessons learnt. Its main contributions are:

- a simple, pragmatic method for making the benefits of formal verification and specification available to system architects in traditional system design. It retains traditional means for testing and validation while avoiding the need for unfamiliar specification languages (Sect. 2).
- a practical method for translating complex, real-life, monad-based Haskell [17] code to Isabelle/HOL, detailed in Sect. 3.
- experience from conducting a large scale verification project (Sect. 4), in particular creating a large, complex specification within a short time and with few resources, as would be common in an industrial setting.

At this early stage we can report that the methodology has been successful and beneficial so far. The formalisation cost was significantly lower than the implementation and testing cost, which is a significant improvement to our earlier experience on formalising and verifying parts of the C implementation of the L4 microkernel. The methodology resulted in a fully working microkernel prototype, implemented in Haskell, formalised in Isabelle/HOL, running normal ARM binaries through a simulator. The formalisation has uncovered a number of problems with the high-level language prototype, including a potentially unbounded operation (Sect. 4).

Although we have only concluded the formalisation stage so far, and have not proceeded to verification on that part of the project, the formalisation already implies one theorem: all system calls terminate.

2. seL4

A microkernel is an OS kernel designed to be minimal in code size and concepts. The kernel is the part of the OS that runs in the privileged mode of the hardware. L4 is a widely deployed second generation microkernel [20] providing the improved reliability and flexibility of the microkernel approach while overcoming the performance limitations of its predecessors. The current L4 implementation is on the order of 10,000 lines of C++ and assembler code.

The seL4 project is a descendant of L4, and aims to provide a secure foundation for high-end embedded systems development (e.g. mobile phones or PDAs). The security goals address two general areas that are lacking in the existing L4 API: communication control between applications, and kernel physical memory management. Control of communication is critical for both providing isolation guarantees between subsystems, and providing confinement guarantees of information possessed by an application. Control of physical memory consumed by the kernel is critical for providing availability guarantees for kernel services, and also for the predictability of their execution times.

On embarking on the seL4 project, we wanted an approach that had the following properties, while enabling the exploration of the design space of potential API solutions that address the issues outlined above

* National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council

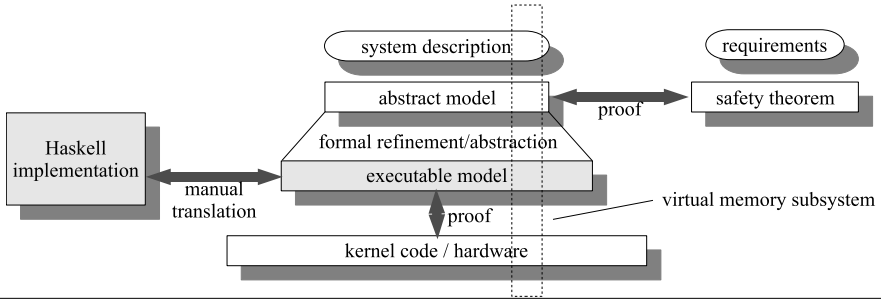

Figure 1. Overview

- The resulting API specification must be precise. Natural language manual-like descriptions are ambiguous and unsatisfactory.
- The approach must expose enough of the implementation details to allow the experimenter to be convinced a high performance implementation is possible.
- It should provide a method for gaining experience with the API by allowing construction of higher-level systems on top.
- It must be readily amenable to formalisation.
- The approach must be usable by kernel programmers who are not adept in formal methods.

The approach taken was to use literate Haskell [17] to specify and implement the seL4 API. Haskell, as a functional programming language, is not a large paradigm shift for typical kernel programmers. This might sound surprising as OS kernels are not usually developed in functional languages, but the proposition came from the OS group, not the verification group. Haskell is side-effect free, and at the same time allows us to explore implementation details of the kernel if desired.

This methodology is in part born out of a pilot project on L4 that we conducted to investigate two aspects of the feasibility of kernel verification: a formalisation of the L4 API using the B Method (topmost horizontal layer in Fig. 1) [19], taking about 6 person months, and a full refinement proof for a non-trivial subsystem (virtual memory) of L4 using Isabelle/HOL (vertical slice in Fig. 1) [26], taking about 18 person months. In the current work, the Haskell prototype serves as a solid, validated basis for the design, formalisation and verification of seL4. The formalisation in this paper occupies the horizontal layer between the most abstract model and the C code.

For validation, to enable the API to be used without requiring a real kernel implementation together with all the complexities of managing real hardware, we created a simulator that implements the ARM processor user-level instruction set and that transfers control to the Haskell kernel for hardware events like page faults and system calls. The simulator enables normal ARM application binaries, compiled for instance from C, to be executed on our kernel prototype.

At the present time, we have an initial seL4 API in Haskell together with the simulator executing ARM binaries. We have formalised this implementation using the mechanism described in Sect. 3 and are now further validating the API by attempting to prove the first security properties and at the same time porting our high-level application environment [15] to the new platform.

As observed in the design iterations so far, we expect to be able to readily adapt the API as we gain more experience in its use for building systems on top of the kernel, without the time consuming debugging usually associated with kernel programming and without large time investments for tracking changes in the formalisation.

3. Translating Haskell

This section gives an overview of our translation from Haskell to Isabelle/HOL. Although some of the solutions below are tailored to our specific problem, they are more generally useful and they demonstrate that interactive specification and theorem proving tools like Isabelle/HOL are suitable for pragmatic, large scale projects.

After discussing our choice of logic in Sect. 3.1, we describe the translation process itself in Sect. 3.2. Our main aim is to keep the translated code readable for interactive verification. Correctness, in particular with respect to partiality, is not our main concern. What will be implemented in the end is a refinement of the verified formal construct in Isabelle. The original Haskell code serves to validate the API design, not the implementation. Danielsson et al [5] show why partiality does not matter in this translation if the program is shown to terminate.

The last three subsections focus on particularly interesting parts of the translation: termination (Sect. 3.3), monads (Sect. 3.4), and the `Dynamic` extension of GHC (Sect. 3.5).

3.1 HOL or HOLCF?

Based on experience in our aforementioned pilot project, the verification tool of choice was the generic interactive theorem prover Isabelle which provides two logic instantiations that might be suitable: HOL and HOLCF.

As the name suggests, HOLCF is an implementation of Scott's logic of computable functions on top of Isabelle/HOL. It is well suited for faithfully describing features of Haskell such as partial functions and lazy evaluation. The Programatica project [9] attempts to automatically translate Haskell into Isabelle/HOLCF. Even though automatic translation would be ideal, we chose HOL over HOLCF, because Programatica at the time of writing was not able to parse our code base, because partial functions and lazy data structures do no play a major role in our code, and because HOLCF as a logic is more heavyweight than HOL, introducing reasoning about domains and continuous functions.

As the pilot study clearly showed that most of the effort will be spent in the later stages of the project, we made the trade-off towards more work in the specification phase instead of more complex reasoning later.

3.2 Types and Terms

The translation proceeded by creating one Isabelle theory per Haskell module. In the case of circular module dependencies which are possible with GHC, we created two Isabelle theories for one module — one with type and constant declarations only, the other with the corresponding definitions.

The bulk of the translation from Haskell to Isabelle/HOL consists of straightforward purely syntactic transformations, some of them just symbol replacements like converting Haskell's `->` function arrow into Isabelle's $\Rightarrow$, and Haskell's prefix notation for type constructors into postfix notation in Isabelle/HOL. The frequent simple case of algebraic data type declarations like `data D a = C1 a | C2 a Int` is trivial to translate; the general case with labelled fields like

```
data D a b = C1 { f1:a } | C2 {f1:a } { f2:Int }
```

has no direct counterpart in Isabelle, but can be simulated by defining separate field selector and update functions. If the appropriate naming and type conventions are respected, the update functions automatically become available in Isabelle as record update syntax.

For many basic terms no translation is required (Haskell and Isabelle syntax and semantics coincide apart from laziness), for most, simple token replacements already do the trick.

Among the more interesting constructs are `let`, where Isabelle does not allow recursive references. These would have to be lifted out and declared in a separate function, although this did not occur in our application. The list comprehensions `[e | pat <- xs, P pat ]` that can be translated directly into `[pat∈xs . P pat]` are only a small subset of the Haskell98 standard, but again were sufficient for our application. For more complex list comprehension expressions a separate function declaration might be necessary. Patterns in lambda, `case`, `let` and list comprehension expressions are restricted in Isabelle. Isabelle allows basic tuple patterns in `let` and lambda, and non-nested constructor patterns only for `case` and primitive recursion. More deeply nested patterns were translated to selectors. For example, for the `option` datatype with the constructors `Some 'a` and `None` and the selector `the (Some x) = x`, the expression `let Some x = f` would become `let x = the f`. More complex patterns in case construct are translated into one or more predicates combined with a let statement for name binding. For example, `case x of p1 -> t1; ...` is translated into `if is_p1 x then let p1 = x in t1 else ...` This is similar to the translation the Haskell98 [17] report gives into the Haskell core language. The difference here is that we use this expansion only when strictly necessary to keep the translated code as close as possible to the original.

Incomplete pattern matches are mapped to the value `undefined` in Haskell which is semantically equivalent to $\bot$, a non-terminating program. Compilers typically abort the program with an error message when `undefined` or `error`, which takes a message as its argument, are evaluated. We handle incomplete pattern matches in the standard Isabelle/HOL way: they are mapped to the value `arbitrary` which exists for every type, but is left unspecified. Since HOL is a logic of total functions, this value exists, but nothing is known about it. We also map explicit calls to `undefined` and `error` to `arbitrary`. This corresponds neatly with Haskell's lazy evaluation. In Haskell the error is only raised when `undefined` is evaluated, i.e. when it contributes to the result of a function. In Isabelle, proofs about the result of the same function only fail due to `arbitrary` when the constant contributes to the result.

Our first instance of this translation process was almost completely manual and still only took a small fraction of the original implementation cost in terms of effort. In the meantime, we have automated most of this process. Our tool is highly incomplete and manages an estimated 90% of the overall translation work automatically with the remaining 10% supplied manually as stubs or sections of translated code. The main lesson from this is that although Haskell is a very rich language and a complete, fully faithful translation is a sizeable project on its own, the easy, incomplete solution does work in practice. It is sufficient even for complex, real implementations of software on the OS level to be able to translate the part of Haskell with a relatively straightforward correspondence to HOL.

A trivial, but important detail was maintaining, as far as possible, a 1:1 correspondence with the Haskell code in both naming and the visual layout of functions. Since the translation was manual anyway, we had initially started out to translate concepts instead of syntax, along the way introducing slight abstractions. It quickly became clear that the better way for ease of translation and tracking change to the original was to translate purely syntactically first, and develop abstractions by proofs later if necessary.

As mentioned above, our application does not use lazy data structures and lazy evaluation as an essential feature. Laziness occurred in expressions like `zip xs [1..]` which could easily be translated to `zip xs [1..length xs]`.

3.3 Termination

Using HOL instead of HOLCF introduces the problem that all functions must be total. As mentioned before, this is our intention anyway, but it introduces an additional proof burden when writing the specification.

Fortunately termination for the bulk of the kernel code is obvious as it does not contain recursion. These parts can be handled by Isabelle's **constdef** which introduces a new constant as an abbreviation of existing constants. Apart from one instance, all recursion occurring in the code was easy to handle using Isabelle's primitive and well-founded recursion constructs. The one difficult instance concerns the one long-running operation of the seL4 kernel: revoking a capability. This is reflected in the code in a mutual recursion over four different functions that traverse the so-called mapping database which keeps track of capability derivations. Isabelle/HOL does support well-founded recursion in one argument, but it currently does not directly support arbitrary mutual recursion. To define these functions, we instead build one recursive function that takes the union of the original parameters together with an additional parameter that determines which of the branches is to be executed. This momentarily introduces more complexity and large, ugly terms, but the original function definitions as they appear in Haskell can then be easily derived as lemmas. For validation and proofs we use these lemmas, not the large construct containing all recursive branches. This technique was documented in detail by Slind [23].

Termination of this function still was nontrivial. The algorithm follows pointers in a data structure that models physical machine memory. We have shown a similar mechanism to terminate in the pilot study [18], but, since we still expect changes from the ongoing validation of the seL4 API in real systems, we would at this stage ideally like to avoid deep proofs that might be obsoleted faster than they were produced. A very simple method of at least guaranteeing that termination depends on the pointer parameter only is the observation that the set of machine words is finite and that traversing the tree will visit each pointer at most once. This termination criterion is easily accepted by Isabelle.

3.4 Monads

Since microkernels are inherently state-based, the Haskell implementation of seL4 uses monads [21] heavily to encapsulate this state. On the one hand this explicit state representation is much closer to HOL than for instance ML's implicit program state. On the other hand, faithfully representing Haskell monads in Isabelle/HOL is problematic.

The main difficulty is that Haskell uses type constructor classes for describing monads abstractly. A monad is a structure of type `'a m` where `'a` is a type variable and `m` a type constructor (like `list` or `option`), implementing two functions `return :: 'a ⇒ 'a m`

and `bind :: 'a m ⇒ ('a ⇒ 'b m) ⇒ 'b m`, written `_ >>= _`. Although there is no way to enforce this in Haskell, to form a monad, the two operations additionally have to satisfy the three monad laws.

Isabelle does provide single parameter axiomatic type classes, but it does not provide constructor classes, and can hence not express monads in the same abstract fashion. There is, however, nothing stopping us from defining concrete monads in Isabelle. The seL4 implementation uses three monads: a state transformer, a state transformer with an exception (`ErrorT`) monad on top, and a state transformer with two exception monads on top. They are easily formalised:

```
('s,'a) state_monad = 's ⇒ 'a × 's
return a ≡ λs. (a, s)
f >>= g ≡ λs. let (v, s') = f s in g v s'
gets f ≡ λs. (f s, s)
modify f ≡ λs. ((), f s)
('s,'a,'b) error_monad = ('s, 'a + 'b) state_monad
returnOk ≡ return ∘ Inr
throwError ≡ return ∘ Inl
lift f v ≡ case v of Inl e ⇒ throwError e
           | Inr v' ⇒ f v'
f >>=E g ≡ f >>= lift g
```

Note that we did not formalise the monad transformer `ErrorT`, but instead the result, an `ErrorT StateMonad`. The functions `Inl` and `Inr` are the projections into the sum type. We leave out the formal definition of the `ErrorT (ErrorT StateMonad)`, it is analogous and introduces another sum type in the result.

We initially defined `('s,'a) state_monad` in the usual way as a data type with the only constructor `State 's ⇒ 'a × 's`. This ensures that `state_monad` is different from the function space `'s ⇒ 'a × 's` and makes conversions between them explicit. Later, for reasoning about the state monad, these explicit conversions got in the way. Apart from purely algebraic reasoning, we often showed equality of two state monads by extensionality (being equal if they yield the same results for all start states). Stating this without explicit conversions was more natural and provided smoother automation.

It was easy to show that the three monad laws hold for all of the instantiations, and it was also not hard to provide a slightly modified `do`-notation where `do x ← f; g x od` stands for `bind f (λx. g)`.

We opted not to use Isabelle's constant overloading, but instead chose different names for each `bind` and `return` implementation with corresponding `do`-notation (`doE`, `doEE`). The reason is again the absence of constructor classes. To express the type of `return` for all implementations, we would have to generalise it to `'a ⇒ 'b` which in turn dilutes the value of type checking the specification. That means we traded off a small notational overhead against higher assurance through type checking. In fact, we found the notational overhead made the specification clearer than the original Haskell code because in its nested `do`-blocks it was often not obvious in which monad the operations are performed.

Fig. 2 shows a typical example of translated monadic code and demonstrates how more complex case patterns are resolved.

For specification purposes, this concrete treatment of monads proved fully adequate. The main disadvantage is that we cannot reason abstractly about monads just in term of monad laws, which could lead to duplication of theorems. So far this did not turn out to be a problem. We mostly had to reason about the behaviour of the state monad, which involved lemmas specific to state monads, not lemmas about monads in general. Scalability was not a problem. For some programs it might turn out inconvenient to not have monad transformers available as such, but only their results. Applying significantly more than three transformers (as in our case) is unlikely to occur in practice, though.

Haskell:

```
activateThread = do
  thread <- getCurThread
  state <- getWaitState thread
    case state of
      NotWaiting -> return ()
      WaitingToSend { pendingReceiveCap = Nothing } ->
        doIPCTransfer thread (waitingIPCPartner state)
      WaitingToReceive {} ->
        doIPCTransfer (waitingIPCPartner state) thread
      _ -> error "Current thread is blocked"
```

Isabelle/HOL:

```
activateThread ≡
do thread ← getCurThread;
   state ← getWaitState thread;
   case state of
     NotWaiting ⇒ return ()
   | WaitingToSend eptr badge fault cap ⇒
       if cap = None then
         doIPCTransfer thread (waitingIPCPartner state)
       else arbitrary
   | WaitingToReceive eptr ⇒
       doIPCTransfer (waitingIPCPartner state) thread
   | _ ⇒ arbitrary
od
```

Figure 2. Typical monad code translation

Although the method described above is very lightweight and proved adequate so far, it would be more satisfactory and scalable to be able to directly emulate constructor classes in Isabelle.

Dawson [6] shows that abstract reasoning about monads is possible in Isabelle/HOL by declaring a new type `'a m` that encodes type constructor application. This makes the types of `return` and `bind` and the corresponding laws directly expressible. Unfortunately, this technique prohibits instantiation.

Lüth et al [3, 16] have extended Isabelle with parameterised theories. They show how this mechanism enables an abstract treatment of monads together with a convenient instantiation mechanism. The only drawback of the method is scalability of another kind: it relies on Isabelle's proof terms to produce the required instantiations. Proof terms currently consume a significant amount of additional resources (mainly memory). For small to medium-sized developments this does not pose a problem, but we expect the size of our proofs to go beyond the limits of current ML systems if proof terms are switched on.

Huffman [14] uses a method similar to Dawson's for modelling monads in Isabelle/HOLCF which he recently extended to Isabelle/HOL [13]. Instead of declaring a new type for all type constructors, he creates an axiomatic class of type constructors and defines a new type for each specific type constructor. For example, instead of showing that `'a option` is of class `monad`, one declares a new type `Option` and instead shows that this is of class `monad`. It can then be used as `'a · Option` where `·` is a new operator for applying type constructors to types. The argument type `'a` is restricted to the class of representable types which are basically types whose values can be enumerated. The approach is flexible, allows abstract reasoning, generic `do`-notation, monad transformers, and instantiation. It is, however, cumbersome to use because it requires explicit conversion between e.g. `'a option` and `'a · Option` that are not present in the Haskell code.

3.5 Dynamic

The `Dynamic` extension of GHC to Haskell98 allows a limited form of type casting: automatic conversion of monomorphic types to the type `Dynamic` and back.

This extension is used in the kernel implementation to model physical memory as a map from addresses (machine words) to a tuple of dynamic objects and their size:

```
psMap :: Map (PPtr w) (Int, Dynamic)
```

Kernel objects belong to the `Storable` type class and implement operations which among others allow their storage to (`storeObject`) and from (`loadObject`) this physical memory.

The question therefore is how to define this type class storable and how to represent `Dynamic` in Isabelle/HOL. These two points are related: had we not wanted to define a type class, the naïve solution of modelling `Dynamic` as the union (a sum or datatype) of all types we possibly might want to store would work. Because some of the types to be stored have parameters, so would the union. Since the class `storable` already describes at least one type variable, this conflicts with the fact that Isabelle supports single parameter type classes only.

We therefore chose a concrete type that is large enough to support an injection of all storable objects: `word8 list` where `word8` is the type of 8 bit machine words. This choice was arbitrary, we could just as well have chosen natural numbers or anything else large enough. We picked `word8 list`, because we already had some of the infrastructure for encoding/decoding other types into it available from our work on a memory model for C pointers [27]. What is new here is lifting these encodings to more complex data structures by using parser combinators.

The axiomatic type class `storable` is built up as follows in Isabelle/HOL.

```
axclass to_from_byte < type
to_byte :: 'a::to_from_byte ⇒ word8 list
from_byte :: word8 list ⇒
             ('a::to_from_byte × word8 list) option

axclass storable < to_from_byte
from_byte (to_byte x @ xs) = Some (x, xs)
```

We use the class `to_from_byte`, a subclass of Isabelle's default `type`, to restrict the type of the two overloaded constants `to_byte` and `from_byte`. The subclass `storable` introduces the defining axiom. The constant `from_byte` has a slightly more complex type than might be expected, because we are interested in what remains of the stream when we have read an object (`@` is the append operator).

We can now define a combinator and an extractor:

```
(f1 -- f2) bs ≡ let res1 = f1 bs;
                    res2 =
                      case res1 of None ⇒ None
                      | Some (obj, rem) ⇒ f2 rem
                in case res2 of None ⇒ None
                   | Some (obj2, rem2) ⇒
                       Some
                        ((fst (the res1), obj2), rem2)

x ▷ f ≡ case x of None ⇒ None
        | Some (y, rem) ⇒ Some (f y, rem)
```

This allows us to build more complex types from existing ones. For example, if we have already proved that `bool::storable`, the datatype used to model capability rights is introduced easily:

```
datatype cap_rights = CapRights bool bool bool bool

to_byte (CapRights b1 b2 b3 b4) =
to_byte b1 @ to_byte b2 @ to_byte b3 @ to_byte b4

from_byte bs ≡ (from_byte --
                 from_byte -- from_byte -- from_byte)
                 bs ▷
               (λ(b1, b2, b3, b4).
                   CapRights b1 b2 b3 b4)
```

Type inference and overloading saves us from specifying which `to_byte` and `from_byte` are to be used. We only need to give the structure of the encoding.

After showing that arbitrarily sized machine words are storable, we encoded natural numbers using repeated modulo/division by 255, with 255 itself as the terminator of the stream. Boolean values were stored as byte values of 0 or 1; similarly for the `option` type, but with the encapsulated object following it in the stream. Lists were encoded as their length (a natural number) followed by their contents. Functions can be encoded as long as their domains can be shown to be finite enumerations; this is done by iterating over the domain and encoding only the range. All other components were based upon these primitives.

We found this approach to scale well beyond primitive types; once these were defined, the build-up of all other storable data types and records was swift, and the instantiation proofs automatic.

4. Experience

As mentioned above, the overall project goal is not to show that microkernels can be verified in principle. The goal is to show that and how it can be done with time and resources in the order of traditional system implementation (within maybe a factor of 2 or 3). The goal is not to verify a toy implementation or simplified abstraction, but a high-performance, binary compatible version of seL4 that can directly be used on embedded devices such as mobile phones.

The basic philosophy in planning and conducting this project is not much different from conducting a software development project. We are aiming to be pragmatic, to use existing tools as far as possible, to employ automation when possible, and to use systematic methods if not.

Translating Haskell in part manually instead of fully automatically was a pragmatic decision. The cost of manual translation was with about 1 person month significantly below that of implementing the Haskell kernel in the first place. The total development cost (including API design ca. 10 person months) was less than our formalisations in the pilot project had suggested. Change tracking using dependency tags in the source files together with normal version control and shell scripts to pick out changes automatically proved to be effective and again well below the cost of the implementing the changes in Haskell in the first place. The overall cost of creating a fully automated tool would have been significantly higher.

So far, the design and formalisation task has gone smoothly and largely according to expectation.

This process of designing a new OS kernel API and formalising it at the same time was highly interactive and interwoven with many iterations and it has not concluded yet. The translation to Isabelle/HOL started relatively early, when the Haskell API was nearing a first stable point and first user-level binaries could be run through the machine simulator on top of seL4. Already during the translation process, we found and fixed a number of problems, for example an unintentionally unbounded runtime of the IPC send operation. It was discovered because Isabelle demanded termina-

tion proofs for operations that were supposed to execute in constant time.

5. Related Work

We have already mentioned work related to the translation of Haskell [9, 14] and the treatment of monads in Isabelle [6, 14, 13, 3, 16] in Sect. 3.

Thompson [25] translates Miranda to Isabelle, but uses first order logic as base and does not treat advanced features like monads or `Dynamic`. Abel et al [1] give an automatic translation of the GHC core language (into which Haskell is transformed for compilation) into the type-theory based Agda system and into first order logic. Some of our manual translations to Isabelle/HOL are similar to those of Haskell to GHC core. Harrison and Kieburtz [11] present P-logic, a program logic for Haskell that focuses on the strict and lazy aspects of the Haskell semantics.

Hallgren et al [10] also produced a microkernel in Haskell. The difference to our work is that they are interested in providing a production kernel in Haskell running directly on the hardware. We are producing a series of design prototypes that are later to result in a high-performance C implementation.

Earlier work on OS verification includes PSOS [22] and UCLA Secure Unix [30]. Later, KIT [2] describes verification of process isolation properties down to object code level, but for an idealised kernel with far simpler and less general abstractions than modern microkernels. A number of case studies [7, 4, 29] describe the IPC and scheduling subsystems of microkernels in PROMELA and verify them with the SPIN model checker. Manually constructed, these abstractions are not necessarily sound, and so while useful for discovering concurrency bugs, they cannot provide guarantees of correctness. The VFiasco project [12] is verifying parts of the L4-based Fiasco micro kernel directly on the implementation level without a more abstract specification of its behaviour. The VeriSoft project [8] is attempting to verify a whole system stack, including hardware, compiler, applications, and a simplified microkernel called VAMOS.

Spies [24] uses the high level specification language *FOCUS* to describe the behaviour of an operating system. The difference to our approach is that we use a language that is foremost a programming language and that can being used as such by the kernel design team without expert knowledge in formal methods. It is one contribution of this work to show that this choice does not sacrifice the ability to arrive at an exact formalisation with little effort.

Our approach occupies the middle ground between two extremes: the a priori approach where the kernel is designed formally from the start, and the a posteriori approach where a traditional (C/C++) implementation is created first and formalised later. Both can be found in the literature, e.g. the formal design process of PSOS [22] and implementation verifications such as [7, 4, 29, 12].

In our setting, the a priori approach would design the kernel directly in the theorem prover and extract a program to be used for validation. This requires that the OS designers are intimately familiar with the formal specification language, which they are usually not. They also would be restricted in their use of the language by the executable fragment of HOL, since validation of low-level design decisions is necessary to distinguish between those designs that can possibly be implemented efficiently and those that cannot. This restriction is significant, because even full Isabelle/HOL, while perfectly suited for specification, is not a comfortable programming language yet, certainly not one for rapid development, testing and prototyping of sizeable, low-level, and largely imperative systems.

The a posteriori approach would create a traditional C implementation first. Folklore says and our own experience [28] shows that the effort for formalisation here is significantly higher and correspondence to the prototype much less obvious. Additionally, the effort for implementation is significantly higher as well — we estimate the effort for creating a micro-kernel prototype the traditional way in our OS group to be about 1 person year. This does not include the numerous iterative changes to the API that we went through in our process.

Our approach lies in between. Compared to the a priori method, we enjoy the richness and expressiveness of a full functional programming language and keep the intricacies of formalisation from the OS designers. Compared to the a posteriori method, we arrive at a precise formalisation very quickly and easily. We also significantly speed up development and make an iterative prototyping process possible that in a few months has gone through more API changes than what would otherwise have taken years to implement.

6. Conclusion

In this paper, we have shown how formalisation can be included in the creation of an OS microkernel while maintaining OS design concerns as the main driving factors. We are convinced that this approach is amenable to spreading software verification wider into industry use, because it makes it easier to achieve scale, and to achieve formalisations quickly and systematically. Note that the formalisation activity can be carried out by a separate group of people — there is no need to replace an established design team with people who are trained in formal methods. This is an important difference to other techniques such as creating design prototypes in the theorem prover directly.

We have shown how a significant part of Haskell98, including a number of common GHC extensions that occur in practice can be systematically translated into Isabelle/HOL. It was not our aim to provide a complete translation for all language features, as our target language HOL is not suited to this task. The programs that are likely to work well with our method are those that terminate and do not make essential use of laziness. Programs that are likely to be problematic are those that make heavy use of laziness and advanced type system features like multi-parameter type classes. As we have shown, though, some of these problems can be solved, at least for specific applications.

We have received feedback on earlier version of this paper in both directions: on the one hand that it is obvious that one would want to be pragmatic in this way and on the other hand that it will not work in the longer term, because the lack of automation will introduce inconsistencies. With longer experience in tracking change and staying in synchronisation with Haskell over many prototype iterations we can safely disagree with the latter. The former criticism is harder to rebut. It is, of course, obvious that you want to be pragmatic when you know with hindsight that the approach works. The main message of this paper is that it does work, and that it does work well. One can in fact formalise large (5.000 loc literate Haskell), real-world programs at a low cost, even though the translation is not fully automatic, HOL and Haskell do not quite match, and the method we use is incomplete. This fact in our view is far from obvious.

We have additionally created (but not shown here) a more abstract specification that is suitable for proofs on security and invariants on kernel data structures. We are currently proving that it is indeed a formal abstraction of the result of the Haskell translation. Again, starting this process has already helped uncover problems in the Haskell implementation that slipped through code reviews and tests (such as an incomplete test in the revoke-capability operation).

This shows that formalisation and the use of theorem proving tools is beneficial even if full verification is not yet performed or is not even planned. In our setting the formalisation cost so far has been significantly lower than the implementation and testing cost, while the design team did not have to switch to completely new methods or notations.

We currently have a fully working microkernel, implemented in Haskell, running normal ARM binaries through a simulator. We have fully formalised this microkernel in Isabelle/HOL by systematic translation. Next to continuing validation and development of the kernel, this formalisation is the basis for future verification of properties of the system (which we have already started), and a formal refinement of the formalisation down to high-performance C code.

Acknowledgements We thank Jeremy Dawson and Brian Huffman for discussions on their monad formalisations. We are also grateful to Manuel Chakravarty, Michael Norrish, and Kai Engelhardt for reading earlier drafts of this paper.

References

[1] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell. Verifying Haskell programs using constructive type theory. In *Haskell'05, Tallinn, Estonia*, 2005.

[2] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[3] E. Broch Johnsen and C. Lüth. Theorem reuse by proof term transformation. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *International Conference on Theorem Proving in Higher-Order Logics TPHOLs 2004*, volume 3223 of *LNCS*, pages 152–167. Springer, Sept. 2004.

[4] T. Cattel. Modelization and verification of a multiprocessor realtime OS kernel. In *Proceedings of FORTE '94, Bern, Switzerland*, October 1994.

[5] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In J. G. Morrisett and S. L. P. Jones, editors, *POPL*, pages 206–217. ACM, 2006.

[6] J. Dawson. Compound monads and the kleisli category. `http://users.rsise.anu.edu.au/~jeremy/pubs/cmkc/`, 2006. Draft.

[7] G. Duval and J. Julliand. Modelling and verification of the RUBIS μ-kernel with SPIN. In *SPIN95 Workshop Proceedings*, 1995.

[8] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In *Proc. 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, pages 1–16, Oxford, UK, 2005.

[9] T. Hallgren, J. Hook, M. P. Jones, and R. B. Kieburtz. An overview of the programatica toolset. High Confidence Software and Systems Conference, HCSS04, `http://www.cse.ogi.edu/~hallgren/Programatica/HCSS04`, 2004.

[10] T. Hallgren, M. P. Jones, R. Leslie, and A. P. Tolmach. A principled approach to operating system construction in haskell. In O. Danvy and B. C. Pierce, editors, *ICFP*, pages 116–128. ACM, 2005.

[11] W. L. Harrison and R. B. Kieburtz. The logic of demand in Haskell. *Journal of Functional Programming*, 15(6):837–891, 2005.

[12] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März, TU Dresden, 2002.

[13] B. Huffman. Constructor classes in Isabelle/HOL. `http://www.csee.ogi.edu/~brianh/`, 2006. Formal Proof Development.

[14] B. Huffman, J. Matthews, and P. White. Axiomatic constructor classes in Isabelle/HOLCF. In J. Hurd and T. F. Melham, editors, *TPHOLs*, volume 3603 of *LNCS*, pages 147–162. Springer, 2005.

[15] The Iguana operating system. `http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/`, 2006.

[16] E. B. Johnsen, C. Lüth, and M. Bortin. Formal software development with Isabelle. `http://www.tzi.de/~cxl/awe/sfd.pdf`, 2006.

[17] S. P. Jones. Haskell98 language and libraries, revised report, Dec 2002.

[18] G. Klein and H. Tuch. Towards verified virtual memory in L4. In K. Slind, editor, *TPHOLs Emerging Trends '04*, Park City, Utah, USA, 2004.

[19] R. Kolanski and G. Klein. Formalising the L4 microkernel API. In B. Jay and J. Gudmundsson, editors, *Computing: The Australasian Theory Symposium (CATS 06)*, volume 51 of *Conferences in Research and Practice in Information Technology*, pages 53–68, Hobart, Australia, Jan. 2006.

[20] J. Liedtke. Towards real μ-kernels. *CACM*, 39(9):70–77, 1996.

[21] E. Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[22] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International, 1980.

[23] K. Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Technische Universität München, Institut für Informatik, 1999.

[24] K. Spies. *Eine Methode zur formalen Modellierung von Betriebssystemkonzepten*. Phdrep, Technische Univerität München, 1998.

[25] S. Thompson. A Logic for Miranda, Revisited. *Formal Aspects of Computing*, (7), March 1995.

[26] H. Tuch and G. Klein. Verifying the L4 virtual memory subsystem. In *Proc. NICTA FM Workshop on OS Verification*, pages 73–97. Technical Report 0401005T-1, National ICT Australia, 2004.

[27] H. Tuch and G. Klein. A unified memory model for pointers. In G. Sutcliffe and A. Voronkov, editors, *12th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-12)*, volume 3835 of *LNCS*, pages 474–488, Jamaica, Dec. 2005.

[28] H. Tuch, G. Klein, and G. Heiser. OS verification — now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pages 7–12, Santa Fe, NM, USA, June 2005. USENIX.

[29] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 20–25, 1997.

[30] B. Walker, R. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.