A Principled Approach to Operating System Construction in Haskell

Thomas Hallgren Mark P Jones
OGI School of Science & Engineering
Oregon Health & Science University
http://www.cse.ogi.edu/~hallgren/
http://www.cse.ogi.edu/~mpj/

Rebekah Leslie Andrew Tolmach
Department of Computer Science
Portland State University
http://www.cs.pdx.edu/~rebekah/
http://www.cs.pdx.edu/~apt/

Abstract

We describe a monadic interface to low-level hardware features that is a suitable basis for building operating systems in Haskell. The interface includes primitives for controlling memory management hardware, user-mode process execution, and low-level device I/O. The interface enforces memory safety in nearly all circumstances. Its behavior is specified in part by formal assertions written in a programming logic called P-Logic. The interface has been implemented on bare IA32 hardware using the Glasgow Haskell Compiler (GHC) runtime system. We show how a variety of simple O/S kernels can be constructed on top of the interface, including a simple separation kernel and a demonstration system in which the kernel, window system, and all device drivers are written in Haskell.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.4.0 [*Operating Systems*]: Organization and Design; D.4.5 [*Operating Systems*]: Reliability—Verification

General Terms Languages,Design,Verification

Keywords Operating systems, Haskell, hardware interface, monads, programming logic, verification

1. Introduction

Systems software often contains bugs that cause system failures, security violations, or degraded performance. One reason for the high bug rate is that most of this software is written in C or C++, which lack strong static typing and memory safety. For example, many security failures are due to buffer over-runs that could have been avoided simply by using a programming language that enforced type safety and bounds checking.

Writing systems software in a relatively low-level implementation language makes it hard to assure that the software obeys key specifications. For example, we might wish to verify formally that an operating system maintains strict data separation between its processes. If the O/S is written in C, this will be a very challenging task, because reasoning about the program must be performed at a very low level. Again, using a programming language with a cleaner and safer semantics would ease the reasoning task.

Given these goals, Haskell is an attractive language for systems programming. The core of Haskell is type-safe and memory-safe, which prevents many classes of bugs, and also pure, which eases reasoning about program behavior. In addition, Haskell's highly expressive type system can be used to capture important program properties without the need for additional proofs.

Systems software needs to interact directly with machine hardware, which can be accomplished in Haskell by using the built-in IO monad and the Foreign Function Interface (FFI) extensions. Unfortunately, this extended system includes raw pointers and pointer arithmetic, which allow writes to arbitrary memory locations and can corrupt the Haskell heap. It also includes `unsafePerformIO`, which can be used to manufacture unsafe type casts. Of course, just as in C, these problems can be addressed by "careful coding," but we would like a safer infrastructure instead.

In this paper, we describe the design, implementation, and application of a restricted monadic framework that achieves this goal. The monad provides access to hardware facilities needed to build an operating system on the Intel IA32 architecture [13], including virtual memory management, protected execution of arbitrary user binaries, and (low-level) I/O operations. The interface is memory-safe in almost all circumstances; the only possible safety violations are ones that occur via abuse of a device DMA controller. Moreover, unlike the full IO monad, it is small enough that we can enumerate useful properties about it as a basis for reasoning about the behavior of its clients. For example, we can assert that executing a program in user space has no impact on kernel data structures. Such properties can be viewed as part of the specification of the interface. We give them as formulas in P-Logic [18], a programming logic for Haskell that has been developed as part of the Programatica project [22], which is an on-going investigation into using Haskell for high-assurance development.

We are using this "hardware monad" as the basis for some experimental operating system kernels that exploit Haskell's strengths:

- House is a small operating system coded almost entirely in Haskell. It builds on the previous `hOp` project conducted by Sébastien Carlier and Jeremy Bobbio [4]. The system includes device drivers, a simple window system, a network protocol stack, and a command shell window in which `a.out` files can be loaded (via TFTP) and executed in user-mode. To our knowledge, this is the first functional-language operating system that supports execution of arbitrary user binaries (not just programs written in the functional language itself).
- Osker, currently in development, is a microkernel coded in Haskell, based on L4 [25], for which we hope to prove some key

ICFP '05 September 26-28, 2005, Tallinn, Estonia.

security properties, including a strong notion of separation between multiple domains and/or virtual machines that are hosted on a single computer. We expect these proofs to be based on a combination of type-based and ad-hoc reasoning. The hardware monad properties will be key axioms in the ad-hoc proofs. The architecture of this kernel makes heavy use of Haskell's type classes, especially to describe monad transformers.

We have implemented the hardware monad directly on top of IA32 hardware. Our implementation uses a modified version of the Glasgow Haskell Compiler (GHC) runtime system [10], based on that of `hOp`. We retain most services of the runtime system, including the garbage collector and, optionally, the Concurrent Haskell multi-threading primitives. In place of the usual file-based I/O, we provide direct access to the IA32's I/O ports and memory-mapped regions, and we add support for direct access to paging hardware and protection mode switching. In effect, this combines the runtime system and the operating system in a single entity.

In addition to this real implementation, we have developed a model implementation of (most of) the hardware monad entirely in pure Haskell 98. This version is built on an abstract model of modern CPUs that uses a combination of state and continuation monads to implement the mechanisms of virtual memory and protected mode execution [16]. The pure Haskell implementation provides us with a more formal setting in which to validate the consistency of the properties that we have specified for the hardware monad. For example, others in our group are working towards mechanized proofs for many of these properties using Isabelle, and based on the semantics of Haskell instead of informal assumptions about any specific CPU platform. However, we omit further discussion of this work from the current paper.

Related work The idea of applying functional languages to systems programming has a long history. Early examples of operating systems implemented in functional languages include Nebula [17] and the Kent Applicative Operating System [24, 6].

A more recent project, Hello [9], implements an operating system in Standard ML and addresses various language design and efficiency issues, like how to access hardware devices and how to handle interrupts in a garbage-collected language. It builds on the results of the Fox project,where Standard ML was used for systems programming. In particular, it includes FoxNet, an efficient implementation of the TCP/IP protocol stack [2]. Compared to these projects, one new feature of our "hardware monad" is its support for controlling memory management hardware, which allows us to run code written in other languages safely.

Functional languages have also been applied when real-time constraints are an issue. Embedded Gofer [27] has an incremental garbage collector in addition to other language features needed to program embedded controllers [28]. We have not seriously addressed real-time issues in our project yet, but if we do, then we might find it necessary to switch to a version of GHC that uses an incremental garbage collector [5].

Our goal of making systems programming safer is shared by projects using other high-level languages. SPIN [23] takes advantage of the type safety of Modula-3 [3] to ensure safety in an extensible O/S kernel. Cyclone [14, 7] is a language deliberately designed as a safe replacement for C.

Modern PC hardware is complex, and building an operating system to control it can be a daunting task. Abstraction layers that provide reusable, higher-level, simpler and safer interfaces to the hardware can be useful in many contexts. In this sense, our "hardware monad" has something in common with OSKit [21], except that we go a step further by equipping our hardware interface with formal assertions.We expect this to facilitate formal reasoning about software built on top of the interface.

The design of appropriate abstraction layers within operating systems is also currently receiving attention from researchers interested in virtualizing the underlying processor. A good summary of this topic can be found in a recent publication by the Xen group [8].

Outline We assume reading knowledge of Haskell throughout. Section 2 introduces the P-Logic notation used in the remainder of the paper. Section 3 describes the Hardware (`H`) monad interface using type signatures and P-Logic assertions. Section 4 gives the complete code for a simple demand-paging kernel for single user processes built on top of `H`. Section 5 outlines the features of House and Section 6 sketches some of the ideas of the Osker kernel. Section 7 gives some details of the real hardware implementation. Section 8 concludes.

2. The Programatica Approach and P-Logic

Much of the work described in this paper has been carried out in the context of the Programatica project, which, at a high-level, is concerned with general methodologies, tools, and foundations to support development and certification of secure and reliable software systems [22]. As part of this work, we are building Osker, a new microkernel written in Haskell. We are using Osker as a case study to evaluate and demonstrate our tools and techniques. At the same time, to establish the feasibility of building a working operating system in Haskell, we have been working on the House demonstration operating system. Both Osker and House have influenced the design of the Hardware monad interface.

One of our main goals is to prove that Osker satisfies some key security properties. With this in mind, we are developing formal models for Osker and its components, including the Hardware interface. The specification and behavior of the the kernel are captured by annotating the source code with properties that are written using an extension of Haskell syntax called P-Logic [18].

In the remainder of this section, we summarize the fragment of P-Logic that is needed to understand the properties that appear in later sections.

Well-formed formulas The core syntax of P-Logic uses the symbols `/\`, `\/`, `==>`, and `-/` for conjunction (and), disjunction (or), implication, and negation, respectively. (Note that `-/` was chosen as an ASCII rendering of the $\neg$ symbol.) Universal quantification over n variables is written in the form `All` v_1 `,` ... `,` v_n `.`*formula*. Existential quantification uses the `Exist` keyword instead of `All`.

Predicates Predicates are written using expressions for the form $P \; e_1 \; \ldots \; e_n$ where P is a property name and e_1 through e_n are program terms. To avoid potential confusion between the property and the programming language notations, compound Haskell terms must be enclosed in braces when they appear inside a P-Logic formula. For example, the predicate `P {x+1} {y+2}` asserts that the values of the expressions `x+1` and `y+2` are related by the property `P`. Braces are not required around simple Haskell expressions such as variables and literals. For example, we might write `All x, y. P x y ==> P y x` to specify that `P` is symmetric.

Semantic equalities are written using two-place predicates of the form e_1`===`e_2. The syntax e_1`=/=`e_2 is also used as a convenient shorthand for `-/(`e_1`===`e_2`)`. Note that these are different from the standard `==` and `/=` operators, which can only be used inside a Haskell term and not directly in a P-Logic formula.

A Haskell expression e of type `Bool` can be lifted into a P-Logic formula using a predicate of the form `True`$\{e\}$, which asserts that e will evaluate to `True`. Of course, the Haskell `Bool` type includes a bottom value, $\perp$, in addition to `False` and `True`, so the lifting of an expression to a formula of the logic in this way—and, indeed, the general treatment of bottom in P-Logic—requires special care and attention. The details of this can be found elsewhere [18].

Predicates as sets It is often useful to identify a unary predicate with the set of values for which it is true. With this in mind, the syntax of P-Logic allows us to write a predicate $P\ e_1\ \ldots\ e_n\ e$ (with $n+1$ arguments) in the form e `:::` $P\ e_1 \ldots e_n$. For example, given a two place predicate `MultipleOf n m`, which asserts that `m` is a multiple of `n`, we might choose instead to write this in the form `m ::: MultipleOf n` and to think, at least informally, of `MultipleOf n` as a refinement of the `Int` type that contains only the multiples of `n`. In this way, the triple colon symbol, `:::`, is used here in much the same way that the double colon symbol is used in Haskell expressions e`::`T to indicate that e has type T.

P-Logic also provides a comprehension-like notation, `{|` $x_1, \ldots, x_n$ `|` *formula* `|}`, for an n-place predicate over the variables x_1 through x_n. For example, the `MultipleOf n` property mentioned previously can be written as `{| m | Exists k. m==={n*k} |}`.

Continuing the view of predicates as sets, P-Logic syntax overloads `/\` and `\/` as intersection and union operators on (partially applied) predicate expressions. For example, assuming the standard definitions for arithmetic, `MultipleOf 2 /\ MultipleOf 3` is equivalent to `MultipleOf 6`.

Defining new properties The Programatica tools allow new property names to be introduced by including `property` definitions in the source of a Haskell program, each of which takes the form:

`property` $P\ a_1\ \ldots\ a_n$ `=` *predicateExpression*

The variables $a_1, \ldots, a_n$ here represent (optional) arguments to P. Figures 1 and 2 show several examples that will also be used in later sections. The properties in Figure 1 describe the usual behavior that we would expect of `set` and `get` operations in a stateful monad. The `StateGetGet` property, for instance, specifies that multiple reads from the state using `get` should return the same value. Note that the `get` identifier used here is a placeholder for a predicate argument, and not a reference to some globally defined `get` operator. The `Returns` property that appears in Figure 1 specifies that a monadic computation returns a given value. It is defined as follows:

```
property Returns x = {| m | m==={m>>return x} |}
```

Figure 2 defines some independence (or non-interference) properties. For example, `Commute c d` specifies that `c` and `d` can be executed in either order, without a change in behavior. `Commute` is used to define `IndependentSetSet set set'`, which asserts that the two `set` operators do not interfere with each other (because, presumably, they operate on distinct components of the state). The predicate `m ::: PostCommute c d` specifies the weaker property that execution of `c` and `d` will commute if we execute the `m` command first. We often use quantification over `m` to express state-dependent properties. For example, we can specify that `c` and `d` commute in any state where the monadic computation `q` would return `True`, by writing:

```
property Q =
    {| m | {do m; q} ::: Returns {True} |}
assert All m. m ::: Q ==> m ::: PostCommute c d
```

Finally, as illustrated here, our tools support the use of `assert` declarations to indicate that a particular P-Logic formula should be true. Declarations like these can be used to record program specifications and also to capture expectations about program behavior.

3. A Hardware Monad

The raw hardware facilities needed to build an operating system within Haskell can be compactly and conveniently specified as the operations of a monad, which we call `H` (for "Hardware"). `H` should be thought of as a specialized version of the usual Haskell `IO` monad, suitable for supporting programs that run in the machine's privileged (supervisor) mode. Since such programs are usually (though not necessarily) operating systems, we refer to them throughout this paper as *kernels*. The `main` program of a kernel has type `H()`.

Like `IO`, the `H` monad provides primitives for expressing stateful computations and for interacting with the "real world," but at a lower level. For example, `IO` has primitives for allocating, reading, and writing mutable reference cells in the garbage-collected Haskell heap, whereas `H` provides only primitives for reading and writing raw physical memory locations disjoint from the Haskell heap. Similarly, `IO` provides stream I/O operations on terminals and files, whereas `H` provides operations to read and write the machine's programmed I/O ports. Some versions of `IO` have support for asynchronous signal and exception handling; `H` gives direct access to the machine's interrupt mechanism. `H` also provides facilities unlike any found in `IO`; in particular, it provides safe access to the machine's virtual memory mechanisms, and supports safe execution of arbitrary machine binaries in user-mode.

Another important characteristic of `H` is that it is *small*: only a dozen types and about twenty operations. This relatively small size makes it possible for us to formalize many of its properties using P-Logic, which would be extremely difficult for the huge `IO` monad (often described as containing "everything but the kitchen sink.") We are particularly interested in documenting *independence* properties of the interface. `H` supports several kinds of memory-like entities: physical memory for user processes; virtual memory page maps; programmed I/O ports; and memory-mapped I/O locations. These kinds of memory are all essentially distinct. For example, writes to physical memory or to a memory-mapped location do not normally affect reads from the page maps or I/O ports. Many of the P-Logic properties that we introduce below are intended to formalize this intuition. It is difficult to imagine expressing similar independence properties for the operations of the standard `IO` monad.

A key design goal for `H` is *safety*. With the exception of certain I/O operations (as discussed in Section 3.4), no operations in `H` can cause corruption of the Haskell heap.

The remainder of this section describes the types and operators of the `H` interface in detail. Our description is divided into four major areas: physical memory access; virtual memory; user-space process execution; and physical I/O. Much of the interface is essentially machine-independent. When a dependency is exposed, we will assume that the underlying hardware is an IA32 machine. With each operator we provide example properties that help specify its behavior. Note that many of these properties fail in the presence of implicit concurrency; we discuss this further in Section 3.5.

3.1 Physical Pages and Addresses

The type `PAddr` represents byte addresses in the machine's raw physical memory.

```
type PAddr = (PhysPage,POffset)
type PhysPage -- instance of Eq, Show
type POffset = Word12

pageSize = 4096 :: Int
```

It is composed from an abstract type `PhysPage`, representing a physical page, and a numeric `POffset`, representing the offset of a byte within that page. Type `Word12` behaves analogously to the standard unsigned integer types (e.g., `Word8`); thus, arithmetic operations are performed modulo `pageSize`.

`PhysPage`s correspond to *physical pages* that are available for allocation to user-mode processes. New physical pages are allocated from (a subset of) the raw physical memory that is installed in the machine using:

```
property StateGetGet get     = {do x<-get; y<-get; return x} === {do x<-get; y<-get; return y}

property StateSetSet set     = All x, x'. {do set x; set x'} === {do set x'}

property StateSetGet set get = All x. {do set x; get} ::: Returns x

property Stateful    set get = StateGetGet get /\ StateSetSet set /\ StateSetGet set get
```

Figure 1. Stateful properties for set and get operations.

```
property Commute     c d = {do x<-c; y<-d; return (x,y)} === {do y<-d; x<-c; return (x,y)}

property PostCommute c d = {| m | {do m; x<-c; y<-d; return (x,y)} === {do m; y<-d; x<-c; return (x,y)} |}

property IndependentSetSet set set' = All x, x'. Commute {set x} {set x'}

property IndependentSetGet set get  = All x. Commute {set x} {get}

property Independent set get set' get' =
                  IndependentSetSet set set' /\ IndependentSetGet set get' /\ IndependentSetGet set' get
```

Figure 2. Non-interference properties for set and get operations.

```
allocPhysPage :: H (Maybe PhysPage)
```

A call to allocPhysPage returns Nothing if no more pages are available. Each allocated page is distinct:

```
property JustGenerative f =
  All m.
  {do Just x <- f; m; Just y <- f; return(x==y)}
    ::: Returns {False}

assert JustGenerative allocPhysPage
```

There is no explicit mechanism to free PhysPages; this guarantees that any value of type PhysPage is usable at any point. An implementation for H may choose to free physical pages implicitly by using GHC's weak pointers[1] to track when pages are no longer referenced by the kernel. In addition, a kernel can recycle pages by maintaining its own free list.

The contents of individual addresses can be read and written using getPAddr and setPAddr, respectively.

```
getPAddr :: PAddr -> H Word8
setPAddr :: PAddr -> Word8 -> H ()
```

By construction, every value of type PAddr is a valid address, so these functions are total. (We do not attempt to model hardware failures such as parity errors.) Each physical address acts like an independent store element with the usual state-like semantics. Using the auxiliary properties from Figures 1 and 2, we can formalize this intuition with the following assertion:

```
assert
  All pa, pa' . pa =/= pa' ==>
    Stateful {setPAddr pa} {getPAddr pa} /\
    Independent {setPAddr pa}  {getPAddr pa}
                {setPAddr pa'} {getPAddr pa'}
```

3.2 Virtual Memory

The virtual memory facilities in the H monad support writing and reading of *page maps*, which define the translation of *virtual addresses* to physical addresses, with associated access permissions and history. Execution of user space code is performed in the address space defined by a particular page map. Although the raw hardware supports only one *current* page map, the H interface supports multiple simultaneous page maps, each of which typically corresponds to a different user process.

VAddr is a concrete type representing virtual addresses.

```
type VAddr          = Word32
minVAddr, maxVAddr :: VAddr
minVAddr            = 0x10000000
maxVAddr            = 0xffffffff
vaddrRange          = (minVAddr,maxVAddr)
```

It wouldn't make sense to treat virtual addresses as abstract types, because user code often references these addresses directly. Not all 32-bit words are *valid* virtual addresses, because part of every user address space (currently the first 256MB) is reserved for the kernel. The range of valid addresses is given by minVAddr and maxVAddr, but, because user code must already be compiled with the range of valid addresses in mind, these parameters are not very useful to the kernel, except for sanity checking. They do allow us to define a validity property:

```
property ValidVAddr
  = {| v | True{inRange vaddrRange v} |}
```

New page maps, represented by the abstract type PageMap, are obtained using allocPageMap. The number of available page maps may be limited; allocPageMap returns returns Nothing if no more maps are available.

```
type PageMap -- instance of Eq, Show
allocPageMap :: H (Maybe PageMap)
```

Each allocated map is distinct:

```
assert JustGenerative allocPageMap
```

As for PhysPages, there is no explicit mechanism to free PageMaps, but the H implementation may do implicit freeing, and a kernel may keep a free list.

Page map entries are indexed by valid virtual addresses; all addresses on the same page share a single entry. The entry for an

[1] A weak pointer to an object is not traced by the garbage collector and hence does not by itself keep the object alive. It can be tested to see whether the object is still alive or not.

unmapped page contains Nothing; the entry for a mapped page contains a value, Just p, where p is of type PageInfo.

```
data PageInfo
  = PageInfo { physPage::PhysPage,
               writable, dirty, accessed::Bool }
    deriving (Eq,Show)
```

Here, the writable flag indicates whether the user process has write access to the page. Fields physPage and writable are written by the kernel using setPage; they are not changed during user-mode execution. The dirty and accessed flags record the history of page use by the user process. Roughly speaking, dirty indicates that the page has been written; accessed indicates that it has been read or written. These fields are typically initialized to False by the kernel using setPage, subsequently updated during user process execution (in ways we make more precise below), and read back by the kernel using getPage.

```
setPage
  :: PageMap -> VAddr -> Maybe PageInfo ->H ()
getPage
  :: PageMap -> VAddr -> H (Maybe PageInfo)
```

Note that PageMaps are simple, one-level mappings; the IA32 hardware actually uses a two-level paging scheme, but we choose to hide this detail underneath the H interface. Page maps are initially empty, so every valid virtual page address maps to Nothing. Each page map entry behaves like an independent store element.

```
property OnSamePage va va' =
  {va `div` (fromIntegral pageSize)}
  === {va' `div` (fromIntegral pageSize)}

assert
  All pm, pm', va, va'.
  va:::ValidVAddr /\ va':::ValidVAddr /\
  (pm =/= pm' \/ -/ OnSamePage va va') ==>
  Stateful {setPage pm va} {getPage pm va} /\
  Independent {setPage pm  va} {getPage pm va}
              {setPage pm' va'}{getPage pm' va'}
```

Moreover, page map entries and physical addresses are mutually independent.

```
assert All pm, pa, va . va ::: ValidVAddr ==>
     Independent {setPage pm va} {getPage pm va}
                 {setPAddr pa}   {getPAddr pa}
```

3.3 User-space Execution

The operator for executing code in a user address space is

```
execContext
  :: PageMap -> Context -> H (Interrupt,Context)
```

The Context type describes the state of the processor, including the values of program-accessible registers and control flags; it essentially plays the role of a continuation. For the IA32:

```
data Context
  = Context { edi,esi,ebp,esp,ebx,
              edx,ecx,eax,eip,eflags :: Word32 }
```

Here, edi, esi, and ebx through eax are general-purpose registers; esp is the stack pointer; ebp is (typically) the frame pointer; eip is the instruction pointer; and eflags is the control flag register (of which only certain bits are accessible to user programs). The context should really also include the floating point registers and other user-visible registers associated with IA32 extensions such as MMX; our current system omits these for simplicity. Similar context structures could be defined for other processor architectures.

Invoking execContext installs the specified page map and context into the appropriate hardware registers and puts the processor into user mode. User code then executes (starting at the eip recorded in the Context) and can access the physical addresses visible to it through its page map. When user-mode execution is interrupted the processor records the new current context and returns to supervisor mode; execContext then returns with that Context and the nature of the Interrupt. For the IA32, we have

```
data Interrupt
  = I_DivideError | I_NMIInterrupt | ...
  | I_GeneralProtection ErrorCode
  | I_PageFault PageFaultErrorCode VAddr | ...
  | I_ExternalInterrupt IRQ
  | I_ProgrammedException Word8
```

Interrupt constructors (of which only a selection are shown here) correspond to fault vector addresses, and fall into three classes:

- Processor-detected faults, such as I_DivideError and I_PageFault; the latter is parameterized by the kind of fault and by the virtual address where the fault occurred.
- Externally-generated interrupts with an associated IRQ channel number. For example, timer interrupts generate I_ExternalInterrupt IRQ0.
- Software exceptions with an associated interrupt number. For example, executing an INT 0x80 machine instruction, which is often used to enter system calls, generates I_ProgrammedException 0x80.

A similar type could be defined for other processor architectures.

This elegant, continuation-like model for user-space computations supports simple and efficient kernel designs. For example, in Section 4, we will outline a simple demand-paging kernel that can be built using these primitives. In the remainder of this subsection, we provide some simple assertions that capture essential features of execContext. Most of these are independence properties. Clearly, the behavior of execContext will be determined by the way that the page map parameter is configured before the function is called. To begin, we can characterize all of the programs m that leave physical address pa inaccessible in the page map pm using this predicate:

```
property NotMapped pm pa =
  {| m | All va . (va ::: ValidVAddr) ==>
                 {do m; isMappedTo pm va pa}
                 ::: Returns {False} |}
```

Here isMappedTo pm va pa is a simple auxiliary Haskell function (not shown here) that is True iff pm provides read access through the virtual address va to the page containing physical address pa. We can define a similar predicate NotMappedWritableTo to specify that a physical page is inaccessible for writing.

Now we can give some key properties of execContext:

- Changing the contents of an unmapped physical address cannot affect execution.

```
assert All pm, pa, c, x, m.
         m ::: NotMapped pm pa ==>
         m ::: PostCommute {setPAddr pa x}
                           {execContext pm c}
```

- Execution can only change the contents of a physical address that is mapped writable.

```
assert All pm, pa, c, m.
         m ::: NotMappedWritable pm pa ==>
         m ::: PostCommute {getPAddr pa}
                           {execContext pm c}
```

- Changing one page map does not affect execution under another page map:

```
assert All pm, pm', va, c, x.
         pm =/= pm' /\ va ::: validVAddr ==>
         Commute {setPage pm' va x}
                 {execContext pm c}
```

- Executing under a page map does not affect the page mapping or writable status of any entry in that map and has no effect at all on any other page map.

```
assert All pm, va, c. va ::: validVAddr ==>
        Commute {getPageField physPage pm va}
                {execContext pm c}
assert All pm, va, c. va ::: validVAddr ==>
        Commute {getPageField writable pm va}
                {execContext pm c}

getPageField field pm va
  = liftM (fmap field) (getPage pm va)

assert All pm, pm', va, c.
         pm =/= pm' /\ va ::: validVAddr ==>
         Commute {getPage pm' va}
                 {execContext pm c}
```

- If execution under a page map changes some physical address, then that map must contain an entry that maps the page as writable and has its dirty and access flags set.

```
property Changed pa pm c
  = {| m |
      {do m; x <- getPAddr pa;
          execContext pm c; y <- getPAddr pa;
          return (x == y)}
      ::: Returns {False} |}

property Dirty pa pm c
  = {| m | Exist va . va ::: ValidVAddr /\
      {do m; execContext pm c; getPage pm va}
        ::: Returns {Just PageInfo
                       { physPage = fst pa,
                         writable = True,
                         dirty = True,
                         accessed = True}} |}

assert All pa, pm, c, m.
         m ::: Changed pa pm c
         ==> m ::: Dirty pa pm c
```

Note that the last assertion only partially specifies how `dirty` and `access` bits behave; it gives sufficient, but not necessary conditions for them to be set after execution. In particular, we do not specify that `access` bits should be set when a user process reads from a page. In fact, we have no way to state such an assertion, because we (deliberately) don't model the instruction-level behavior of user processes. Fortunately, the kernel properties we are interested in proving do not rely on having accurate `access` bit information.

3.4 I/O and Interrupts

The `H` interface supports the use of IA32 programmed I/O ports, memory-mapped I/O, and external interrupts delivered by a programmable interrupt controller. Inevitably, this part of the `H` interface is highly IA32-specific.

Designing a general-purpose interface to I/O hardware is not straightforward. To allow flexible programming of drivers for a wide range of devices, the abstraction level must be fairly low. This, however, allows interface primitives to be misused. On the other hand, a more restrictive, higher-level interface would necessarily be less flexible about what kinds of devices were supported, and lots of trusted, device-specific code would have to move underneath the interface. Currently, we have implemented a fairly low-level interface that provides no guarantees about correct usage of I/O devices, but does enforce memory safety in almost all circumstances. We may experiment with higher-level interfaces in the future.

Programmed I/O Many PC devices (including timers, the CMOS and real-time clock devices, the keyboard and mouse, network interfaces, etc.) are controlled by reading and writing data and control information to specified *ports* via special `in` and `out` instructions. These are available through `H` via the following functions:

```
type Port = Word16

inB  :: Port -> H Word8
outB :: Port -> Word8 -> H ()
...similar functions for Word16 and Word32...
```

This interface is very flexible, but also easy to abuse:

- The ports for a device are fixed by the hardware or assigned dynamically during system start-up, depending on the device controller's capabilities. Currently, the `H` interface makes no attempt to check that the `Port` argument is valid in any way (e.g., that it corresponds to an installed device).
- Device ports can be used to do very disruptive things, sometimes in surprising ways. For example, writing a certain command word to the keyboard controller can reset the whole computer! The `H` interface makes no attempt to check that meaningful control arguments are passed to a given device, much less that they are constrained to "desirable" behaviors.
- Devices using DMA sometimes specify the address of the in-memory buffer by writing it to a control port. A rogue kernel could use this feature to make the device overwrite arbitrary parts of memory, including the Haskell heap. This is the only potential hole in `H`'s memory safety.

Because of these problems, assertions about "well-behavedness" of `H` can only be valid modulo an assumption that IO device commands are being used "properly." In principle, it might be desirable to formalize this assumption, and make it an explicit proof obligation within the P-Logic framework. Unfortunately, any such formalization will be device-dependent and probably very complicated. In practice, therefore, we believe it is best to treat the assumption informally and implicitly. With this understanding, we can make some assertions about the independence of the port space relative to user physical memory, and to page tables, as in the following examples (for the byte versions):

```
assert All p, pa .
   Independent {outB p} {inB p}
               {setPAddr pa} {getPAddr pa}

assert All p, pm, va .
   va ::: validVAddr ==>
    Independent {outB p} {inB p}
                {setPage pm va} {getPage pm va}
```

Note that the ports do *not* themselves behave like store elements. For example, `outB` is not idempotent, and `inB` will not usually return a value just written with `outB`.

Memory-mapped I/O Some devices make their control and status registers available at special physical memory locations that can be read and written using ordinary load and store instructions. Video cards usually make their frame buffers accessible in the same way. The physical addresses for this pseudo-memory are fixed by the device or negotiated by software executed at system initialization; they can then be mapped to any convenient virtual memory locations. To make access to these devices safe, memory-mapped locations must be kept abstract by the `H` interface, with access only by special-purpose `get` and `set` functions that perform bounds checks.

```
type MemRegion
type Offset
setMemB  :: MemRegion -> Offset -> Word8 -> H ()
getMemB  :: MemRegion -> Offset -> H Word8
validMemB:: MemRegion -> Offset -> Bool
...similar functions for Word16 and Word32...
```

(The `validMem` functions can be used to test validity of an offset; they are primarily used to formulate properties.)

For example, the `H` interface currently includes primitives for accessing the frame buffers for a basic text video mode or a VBE-compliant graphics video mode. Each of these video devices gets its own buffer, and perhaps additional device-specific information.

```
textVideoRegion :: H (Maybe MemRegion)
gfxVideoRegion  :: H (Maybe FrameBufferInfo)

data FrameBufferInfo
   = FBInfo { width, height,
              bitsPerPixel    :: Int,
              frameBuffer     :: MemRegion,
              maskSizes,
              fieldPositions  :: (Int,Int,Int)
            }
```

To perform text video output, individual characters (with graphic attributes) are written using `setMemW textVideoRegion`. To perform graphic video output, individual pixels are written using `setMemB (frameBuffer gfxVideoRegion)`.

Again, we have the expected independence properties with respect to user physical memory and to page tables:

```
assert All mr, off, pa .
 True{validMemB mr off} ==>
 Independent {setMemB mr off} {getMemB mr off}
             {setPAddr pa}     {getPAddr pa}

assert All mr, off, pm, va .
 va ::: ValidVAddr /\ True{validMemB mr off} ==>
 Independent {setMemB mr off} {getMemB mr off}
             {setPage pm va}   {getPage pm va}
```

However, we cannot expect a general property asserting independence of memory-mapped and programmed I/O operations, as many devices support both, with complex, intertwined semantics.

Additional memory-mapped I/O regions can easily be accommodated by extending the interface, and DMA buffers could also be treated in a similar way. The obvious drawback of this scheme, however, is that the interface must be altered for each new device; a more generic mechanism would obviously be desirable.

Interrupts Most devices attached to an IA32 signal interrupts through the programmable interrupt controller, which associates each interrupt source with an interrupt request channel (IRQ). On a PC, some IRQs are statically assigned (e.g., IRQ0 corresponds to the hardware timer, IRQ1 to the keyboard, etc.); others may be dynamically assigned (e.g., to PCI bus devices).

```
data IRQ = IRQ0 | IRQ1 | ... | IRQ15
           deriving (Bounded, Enum)
```

Individual IRQs can be enabled or disabled:

```
enableIRQ, disableIRQ :: IRQ -> H ()
```

Interrupts can also be globally enabled or disabled, independently of the per-IRQ settings.

```
enableInterrupts, disableInterrupts :: H ()
```

Interrupts are handled in two fundamentally different ways, depending on whether they are received while the the processor is in user mode or in supervisor mode. Interrupts occurring in user mode cause a switch back to supervisor mode and an immediate return from `execContext` with a suitable `I_ExternalInterrupt` value. The kernel can then handle the exception however it sees fit. Treatment of interrupts received in supervisor mode is described in the next section.

3.5 Concurrency

Any realistic kernel must simulate concurrency: kernel code, multiple user processes, and pending interrupt handlers must time-share the processor[2] so that they appear to be running "at the same time." We are exploring two basic approaches to achieving this:

- *Implicit Concurrency.* The kernel uses Concurrent Haskell multithreading primitives, as implemented in the GHC library and RTS. For example, each user process can be set to run in a separate thread. (This implies that `H` primitive operations must be thread-safe, i.e., any mutable data structure is protected by a lock.) Interrupts received while in supervisor mode can also be handled in separate threads (see below). Our House implementation (Section 5) uses this approach.
- *Explicit Concurrency.* The kernel simulates concurrency using queues of continuations (`Context` values). If interrupts are allowed in supervisor mode, they must be polled for explicitly by kernel code. Our Osker implementation (Section 6) implements this approach.

In many Haskell implementations, Haskell code can only be paused at certain "safe points"—in GHC, for example, at garbage collection heap-check points. Thus, interrupts occurring in supervisor mode (while the processor is running Haskell code) cannot be handled immediately. Instead we associate these interrupts with a small handler (written in C) that just sets a flag to indicate that a particular `IRQ` was signaled. In the explicit concurrency model, the kernel must poll these flags periodically, using a function

```
pollInterrupts :: H [IRQ]
```

that returns a list of pending interrupts; it can then invoke appropriate (Haskell) handlers if needed. In the implicit model, this polling is done by the Concurrent Haskell RTS code at safe points; we extend `H` with a function

```
installHandler :: IRQ -> H() -> H()
```

that registers a Haskell handler which the RTS is to run (in a separate thread) if the interrupt flag for the given `IRQ` is set.

There are pros and cons to each approach. Using implicit concurrency allows kernel code to be written without explicit attention to timing issues; for example, there is no need to bound the execution time of a pure sub-computation because an interrupt will eventually cause a thread switch if necessary. However, under this approach, many of the state-based assertions given in this section are not really true because another thread may execute in between two apparently consecutive operations. The explicit concurrency approach doesn't have this problem. Moreover, the explicit approach

[2] Of course, multiprocessor systems are truly concurrent; we don't consider them in this paper.

is better for kernels that need to manage thread priorities (e.g., to run interrupt handlers first), because the current GHC implementation does not permit control over the thread scheduling algorithm.

4. A Simple Executive for User Programs

This section shows how the H monad interface can be used to construct a simple demand-paging kernel for executing IA32 user binaries. Binaries are assumed to have a conventional memory layout with code, data, heap, and stack regions. To keep the presentation brief, we allow binaries to use just two system calls: `exit` and `brk`. Hence, they cannot do I/O (except to return a result value at `exit`); to illustrate the use of interrupts and I/O operations, we use an interval timer to track the binary's execution time and abort it after a specified number of milliseconds. As a further simplification, the kernel can run only one binary at a time. More realistic and elaborate kernels are described in Sections 5 and 6.

We describe both static and dynamic characteristics of an executing binary using a `UProc` (user process) record.

```
data UProc
  = UProc { entry                 :: VAddr,
            codeRange, dataRange  :: VRange,
            startBss, brk         :: VAddr,
            stackRange            :: VRange,
            codeBytes, dataBytes  :: [Word8],
            pmap                  :: PageMap,
            ticks                 :: Int
          }

type VRange = (VAddr, VAddr)
```

We assume that the kernel is given a `UProc` with all fields except `pmap` and `ticks` already filled in with a static description of the binary. The field `codeRange` delimits the program's code region; `dataRange` does the same for the data region. The contents of these regions are given by `codeBytes` and `dataBytes`; for simplicity we model these as lists although a real implementation would probably use unboxed arrays. The `entry` field holds the initial instruction pointer. The stack grows down from the end of `stackRange`, the start of which provides a fixed bottom limit. The bss/heap region begins at `startBss` and grows up to `brk`. All these `VAddr`s must be valid in the sense of Section 3.2. The `brk` limit can be arbitrarily altered by a system call (see below) as long as it remains between `startBss` and the start of `stackRange`.

There are no restrictions on the nature of user code; it can (try to) do anything at all! Of course, direct attempts to perform privileged or illegal operations, such as writing to I/O ports or dividing by zero, will cause a fault that immediately returns control to the kernel. User code can request a kernel service by making a system call. To do so, it places the call number (0 for `exit`, 1 for `brk`) in register `eax` and the argument in `ebx`, and issues an `INT 0x80` instruction, which causes a software interrupt. This protocol for making system calls is normally encapsulated in library procedures.

The entry point for the executive is shown in Figure 3. The parameters specify that the binary represented by `uproc` is to be run for a maximum of `msecs` milliseconds. The executive allocates a fresh page map (initially empty), and constructs an appropriate initial `Context` value. After the timer has been initialized (discussed further below), the `UProc` is updated to record the page map and the number of timer ticks to be allowed before terminating the program, and control passes to the `exec` loop, which is the heart of the executive (see Figure 4). The loop is parameterized by the current `uproc` and `context`. At each iteration, control is passed to the user code by `execContext`, which returns only when that code's execution is interrupted in some way. This may be because of a system call, page fault, timer interrupt, or unexpected fault of some kind. After the interrupt is processed, either `exec` is called tail-recursively to re-enter user code at the point where the interrupt occurred, or execution terminates, returning a suitable text message.

```
runUProc :: UProc -> Int -> H String
runUProc uproc msecs
  = do Just pmap <- allocPageMap
       initTimer 10 -- interrupt every 10msec
       exec uproc{pmap=pmap,ticks=msecs ‘div‘ 10}
            context
  where
    context =
      Context {eip=entry uproc,
               esp=snd (stackRange uproc)+1,
               edi=0,esi=0,ebp=0,ebx=0,
               edx=0,ecx=0,eax=0,eflags=0}
```

Figure 3. `runUProc` provides the entry point to our executive.

```
exec :: UProc -> Context -> H String
exec uproc context
 = do (interrupt,context’) <-
        execContext (pmap uproc) context
      case interrupt of
        I_ProgrammedException(0x80) ->
          --- system call ---
          let callnum = eax context’
              arg = ebx context’
          in case callnum of
               0 -> --- exit ---
                    return("Successful completion "
                           ++ "with result "
                           ++ show arg)
               1 -> --- brk ---
                    ...details omitted...
               _ -> exec uproc context’{eax=(-1)}

        I_PageFault _ faultAddr ->
          --- page fault ---
          do fixOK <- fixPage uproc faultAddr
             if fixOK
               then exec uproc context’
               else return ("Fatal page fault at "
                            ++showHex faultAddr "")

        I_ExternalInterrupt 0x00 ->
          --- timer interrupt ---
          do let ticks’ = ticks uproc - 1
                 uproc’ = uproc{ticks=ticks’}
             if ticks’ > 0
               then exec uproc’ context’
               else return ("Time exhausted")

        _ ->
          return ("Unexpected Fault or Interrupt: "
                  ++ show interrupt)
```

Figure 4. The heart of a simple demand-paging executive.

System calls are handled by dispatching on the call number. The only non-trivial call is `brk`, which alters the `brk` field in the `uproc`; we omit the details here. If an invalid system call number is requested, the error code -1 is returned (by placing it in `eax`).

```
fixPage :: UProc -> VAddr -> H Bool
fixPage uproc vaddr
  | inRange (codeRange uproc) vaddr
  = do setupPage uproc vbase
                 (drop offset (codeBytes uproc))
                 False
       return True
    where
      vbase = pageFloor vaddr
      offset =
        fromEnum (vbase - fst (codeRange uproc))
fixPage uproc vaddr
  | inRange (dataRange uproc) vaddr
  = ...similar...
fixPage uproc vaddr
  | vaddr >= startBss uproc && vaddr < brk uproc
    || inRange (stackRange uproc) vaddr
  = do setupPage uproc vbase (repeat 0) True
       return True
    where vbase = pageFloor vaddr
fixPage uproc vaddr
  | otherwise = return False

setupPage:: UProc -> VAddr -> [Word8] -> Bool ->H()
setupPage uproc vbase src writable
  = do page <- newPhysPage
       let pi = PageInfo {physPage=page,
                          writable=writable,
                          dirty=False,
                          accessed=False}
       setPage (pmap uproc) vbase (Just pi)
       zipWithM_ (curry setPAddr page)
                 [0..pageSize-1]
                 src
```

Figure 5. Auxiliary functions `fixPage` and `setupPage`

When the timer interrupts, the kernel executes a miniature handler which decrements the number of allowed ticks remaining for the user code, and terminates execution if the tick count is zero.

Page faults are dispatched to auxiliary function `fixPage` (see Figure 5), which returns `True` iff the missing page has been successfully loaded. Note that no pages are mapped at all when execution begins; the system relies on page faults to force them in when needed. In particular, the very first call to `execContext` will always cause a fault on the page containing the initial `eip` address.

`fixPage` analyzes the faulting virtual address and attempts to set up the missing page appropriately according to the region it belongs to. `pageFloor` returns the nearest aligned virtual address below its argument. The contents of code pages are obtained from `codeBytes` and they are marked read-only; similarly, data pages are obtained from `dataBytes` and are marked writable. Stack and heap pages are zeroed. If the fault address is outside any region, `fixPage` does nothing and returns `False`. The auxiliary routine `setupPage` is used to obtain a fresh physical page, install it in the page map, and load its contents with the requested access rights.

Finally, if `execContext` returns any other kind of interrupt, user code execution is terminated and an appropriate error message is returned.

Before the user code runs, it is important to initialize the programmable interrupt timer; otherwise the kernel may never get control back. This is done by the call to `initTimer`, which writes an appropriate value to the timer's control port and enables its interrupt line.

```
initTimer :: Int -> H ()
initTimer msPerTick
  = do -- set timer interrupt frequency...
       outB timerPort (ticks .&. 0xff)
       -- ... in two steps:
       outB timerPort (ticks `shiftR` 8)
       enableIRQ timerIRQ
  where
    ticks = fromIntegral
              (1193190 * msPerTick `div` 1000)
    timerIRQ = IRQ0
    timerPort = 0x40 :: Port
```

5. House: A Working Demonstration System

Having a safe, high-level language like Haskell at our disposal, it is natural to explore the possibility of using it to build an entire system, including kernel, system programs, and applications. Building on the work of the `hOp` project and some other existing projects, we have constructed a prototype for such a system, which we call House. The system includes the following features, all written in Haskell and built on `H`:

- Device drivers, including interrupt handlers, for some useful PC hardware devices: keyboard, mouse, graphics and a network card. (The keyboard and text mode video drivers were provided by `hOp`.)
- A simple window system, including some demo applications. (This is the Gadgets system [20], originally implemented in Component Gofer, and ported by us to Concurrent Haskell.)
- A network protocol stack with basic support for the following protocols: Ethernet, ARP, IPv4, ICMP, UDP and TFTP.
- A command shell window, where separate `a.out` binaries can be loaded via TFTP or by GRUB and run as protected user-space processes.
- Support for multiple simultaneous user processes and interrupt handling, using Concurrent Haskell extensions (threads, channels, `MVars`)

The system is freely available for download [12].

Networking The largest House component we have developed ourselves is a simple network protocol stack. We use a uniform representation of interfaces at each network protocol layer, with parsing/unparsing combinators to convert between uninterpreted and structured representations of packets. This may turn out to be a bad idea from a performance perspective; we don't yet have any performance data.

We have implemented a driver for NE2000 compatible network cards, because this is the card model supported by our development emulator, QEMU [1]. (At present, we don't have a physical card to drive.) The network driver is implemented as a single thread that handles the communication with both the network device and the client. We install a simple interrupt handler that just sends a message to the handler thread when the card has finished receiving an incoming packet or transmitting an outgoing packet.

The overall structure of our network protocol stack implementation is fairly conventional. We have aimed to make it easy to add support for more protocols (e.g., IPv6 and TCP) and link layers (e.g., PPP).

Graphics and the Gadgets window system The Gadgets implementation [20] relies on a few simple graphics primitives that we have implemented in Haskell by writing to the linear frame buffer interface through the VBE [26] interface (see Section 3.4). This solution is simple, portable and efficient enough for demonstration purposes. In the future, we might, of course, want to take advantage of hardware accelerated graphics routines.

6. Towards Osker: Modeling a Separation Kernel

We are also using the `H` interface as a basis for the implementation of a kernel called Osker (the "Oregon Separation Kernel") that is designed to support concurrent execution of multiple user processes with controlled interactions between them. Unlike House, the implementation of Osker adopts the explicit approach to concurrency that was described in Section 3.5. Specifically, the kernel maintains Haskell data structures that describe the set of active processes in the system at any given time, and it uses these to make scheduling decisions between the available threads. Osker also supports IPC (interprocess communication), which provides a mechanism for synchronous message passing between threads; its design is closely based on the well-known L4 microkernel [25].

Our long-term goal is to establish strong guarantees of separation for Osker by proving, formally, that concurrent processes running on the same system can only interfere with one another to the degree permitted by an explicit system security policy. This is work in progress, but it seems clear that such a proof will depend on the independence properties of `H` described in Section 3.

In the hopes of simplifying the verification effort, we are leveraging compile-time type checking in Haskell to ensure that key system invariants are maintained. We are also using Haskell type classes and monad transformers to support modular construction of the kernel and, we hope, to facilitate a correspondingly modular proof of separation. The specific techniques that we are using here are similar to those previously applied in the construction of modular interpreters [15, 19].

The remainder of this section describes some of these aspects of our current prototype in more detail and show hows the `H` interface is used in this context.

Domains and Threads In the terminology of Osker, a system is organized as a collection of *domains*, each of which has a distinct user-mode address space that is shared by multiple threads.

```
type System = [Domain]

data Domain
  = Domain { uproc    :: UProc,
             runnable :: [Thread Runnable],
             blocked  :: [Thread Blocked] }
```

The `uproc` field holds information about the address space and user code, just as in Section 4. The remaining fields store information about threads belonging to the domain: `runnable` is a queue of threads waiting to execute (maintained in priority order), and `blocked` is a list of threads waiting to send or receive a message.

Every thread has an identifier, an associated scheduling priority, and details reflecting its current `state`:

```
data Thread s = Thread { threadId :: ThreadId,
                         priority :: Int,
                         state    :: s }
```

The information that we need for a thread that is runnable (i.e., awaiting execution in a scheduling queue) is different from the details that are needed for a thread that is blocked (i.e., waiting to transfer a message). We reflect this by making the state type a parameter `s` of `Thread`, and by instantiating it, as appropriate, to one of the following types:

```
data Running  = Running
data Runnable = Runnable { ctxt :: Context }
data Blocked  = Sending   ThreadId Context
              | Receiving ThreadId Context
```

A running thread has no state-specific data, so the `Running` type does not carry any extra information. For a runnable or blocked thread, the kernel must record the user-space continuation from the thread's last execution, represented by a `Context`. `Blocked` also carries information about the pending message transfer, which the kernel uses to detect when a thread should be unblocked.

Parameterizing the `Thread` types with a state has many useful consequences. For example, with this approach, the type system will prevent us from inadvertently scheduling a `Blocked` thread or from placing a `Runnable` thread in the `blocked` list of a domain.

Tracking Effects At runtime, the kernel and the hardware maintain several state components, including, for example, a list of all the current domains. Some parts of the system, however, do not require access to all of these components. For example, one system call might only require access to the virtual memory operations of `H`, while another requires access to the state of the running domain, but nothing else. If we can capture and limit the impact of such dependencies automatically using types, then we can also hope to limit the number of side-effects we must reason about in corresponding sections of our proof.

In general terms, for example, the code for handling each of the system calls in Osker is described by a function of type:

```
  Thread Running -> Context -> Handler ()
```

The two parameters describe the thread that made the system call and the associated context. The `Handler` type is a monad that is constructed on top of the `H` interface, using a state monad transformer called `StateT` [15, 19].

```
type Kernel  = StateT System H
type Handler = StateT Domain Kernel
```

The resulting monad includes state components for the domain of the current thread and for the list of active domains in the system. For completeness, the following code fragments show some details of the implementation of `StateT`, and also a generic `StateMonad` type class that encapsulates the notion of a stateful monad.

```
newtype StateT s m a = StateT (s -> m (a, s))

instance Monad m => Monad (StateT s m)
  where ...

class StateMonad s m where
  get    :: m s
  set    :: s -> m ()
  update :: (s -> s) -> m ()

instance (Monad m) => StateMonad s (StateT s m)
  where ...
```

Suppose now that we add a simple system call, `yield`, which the calling thread can use to yield the remainder of its timeslice. The implementation of `yield` does not modify the list of active domains or use operations from the `H` interface; it only modifies the data structure of the running domain. To capture this, we define the handler for `yield` as a function of the following type, without explicitly mentioning the `Handler` monad:

```
(StateMonad Domain m) =>
  Thread Running -> Context -> m ()
```

This type uses a predicate `(StateMonad Domain m)` to specify a constraint on the choice of the monad `m` (specifically, that it must have a state component of type `Domain`). We can also see that the implementation of `yield` does not require any features from the `H` interface because there is no mention of `H` in its type.

As a second example, consider the implementation of a spawn system call that takes a start address and an identifier for a new thread in the current domain as parameters. In this case, the handler has type:

```
(HMonad m, StateMonad Domain m) =>
  VAddr -> ThreadId ->
    Thread Running -> Context -> m ()
```

The implementation of spawn requires operations from the H interface to allocate thread-local space for the new thread. This is reflected in the type by the use of the HMonad class, which is a wrapper that provides access to all of the operations of H and allows them to be lifted implicitly and automatically through arbitrary layers of monad transformers.

```
class Monad h => HMonad h where
  allocPhysPage :: h (Maybe PhysPage)
  getPAddr      :: PAddr -> h Word8
  setPAddr      :: PAddr -> Word8 -> h ()
  ...

instance HMonad H where ...
instance HMonad m => HMonad (StateT s m)
  where ...
```

Note that the declaration of the HMonad class includes a signature for each of the operations in the H interface, but replaces each use of the H type constructor with the type variable h. The two instance declarations describe how these functions are implemented for H (by binding the overloaded names to the non-overloaded primitives introduced earlier) and for transformed versions of the monad like Kernel and Handler (by applying suitable liftings).

Although we have omitted some details, these examples illustrate the use of type classes and monads to provide a form of effects analysis that can capture dependencies explicitly in inferred types.

Scheduling and Exception Handling Osker implements round-robin scheduling for domains and priority-based scheduling for threads. The domain scheduler selects a domain to run from the list of active domains and passes it to the thread scheduler, called tScheduler, which selects the thread to be run.

```
tScheduler :: (HMonad m, StateMonad System m)
                => Domain -> m ()
tScheduler dom
 = case runnable dom of
    []     -> return ()
    (t:ts) -> do dom' <-
                   execThread (uproc dom) t
                    `runStateTs` dom{runnable=ts}
                 update (insertDomain dom')
```

If the queue is empty, the domain is finished or deadlocked, and is not rescheduled. Otherwise, the thread at the head of the queue is selected and executed using execThread. The runStateTs operator used here is part of the library that defines StateT:

```
runStateTs :: Monad m
               => StateT s m a -> s -> m s
runStateTs (StateT c) s = do (a,s') <- c s
                             return s'
```

In this specific case, runStateTs is used to embed an execThread computation from the Handler monad into a computation in the Kernel monad by passing in an additional Domain parameter. The thread scheduler uses the update call to insert the modified version of the current domain back into the list of active domains. At this point, tScheduler returns to the domain scheduler, which will loop back to execute a thread from the next active domain.

The execThread function replaces the execution loop, exec, of the kernel in Section 4. Notice how the type of execThread ensures that we can only execute Runnable threads.

```
execThread :: (HMonad m, StateMonad Domain m,
               StateMonad System m)
             => UProc -> Thread Runnable -> m ()
execThread u t
  = do (intr, ctxt') <-
         execContext (pmap u) (ctxt (state t))
       handle t{state=Running} ctxt' intr
```

This definition uses the execContext function from the H interface to run the continuation stored in the ctxt field in the state of thread t. The result is an Interrupt value and a modified Context. To service the interrupt, we change the thread state to Running and call the function handle. This works in a similar way to the case expression in the definition of exec (Figure 4) and passes control to our system call handlers where necessary.

7. Implementing the Hardware Monad

We have implemented the H interface directly on top of real IA32 hardware. The implementation consists of about fifteen small modules, amounting to about 1200 lines of Haskell, supported by about 250 lines of C code (with a few lines of embedded assembler) accessed via the Foreign Function Interface (FFI). An additional 500 lines of C and 150 lines of assembler were added to version 6.2 of the standard Glasgow Haskell Compiler (GHC) runtime system (RTS) to support booting on a bare machine and to set up paging. Our current implementation can safely be used with Concurrent Haskell threads, which are specific to GHC; in particular, we use QSems to protect critical sections. Except for this, our Haskell code could probably be ported to a different, FFI-compliant Haskell 98 implementation without too much difficulty.

The lower levels of our implementation are based heavily on the hOp Haskell microkernel [4]. hOp can be viewed as a port of the GHC RTS to a bare IA32-based PC. The standard RTS start-up code is extended to initialize the bare machine. The remainder of the RTS is scrubbed to remove any dependencies on an underlying operating system. The standard C library is replaced by a minimal version that suffices to support compiled Haskell code and output to the text-mode display. There is a small library of device drivers written in Haskell, using the IO monad and FFI facilities to access the hardware (this is straightforward since the entire system runs in privileged mode). The drivers make heavy use of GHC's Concurrent Haskell thread mechanism for intra-kernel communication and for interrupt handling. A simple Haskell main program exercises the drivers. All the Haskell code, including relevant parts of the Haskell standard library, is compiled and linked by GHC in the ordinary way. The resulting microkernel is then converted to a boot image that can be loaded and started by the GRUB boot loader [11].

To implement H, we have modified and extended the hOp work by adding support for virtual memory and user-mode process execution in a protected address space. We have also implemented a simpler and more flexible approach to interrupt handling, reduced the dependence of the system on Concurrent Haskell threads, and added support for memory-mapped I/O (e.g., for graphics/video). The remainder of this section gives some details of these features.

Physical Memory Figure 6 shows a typical physical memory layout, assuming (for simplicity) a total installed memory size of 64MB. The C stack and heap are used by the code in the GHC RTS, our initialization code, and in the foreign functions called as part of our H implementation. The kernel (main program) executable image contains all compiled Haskell code, the GHC RTS, and our C and assembler extensions. (Although 7MB is set

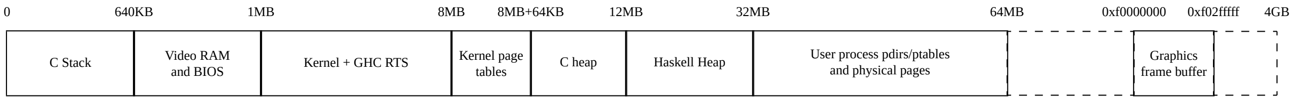

Figure 6. Physical Memory Layout (not to scale!)

aside for this image, our current implementations use much less; for example, the complete House system described in Section 5 occupies less than 3MB.) The Haskell heap is used to store Haskell data structures in the usual way; it is managed by the standard GHC garbage collector. Kernel page tables describe the page mapping used by kernel code; this is incorporated as part of every user-mode page map too (see below). User page directories, page tables, and physical pages holding code and data are allocated out of a single large memory pool.

In addition to the regions we allocate, certain regions of physical address space are associated with specific devices to support memory-mapped I/O. The low-memory region at 640KB contains a text video buffer and BIOS RAM on all PC's. The graphics frame buffer in high memory is shown as an example: a typical system will have one or more such buffers at device-specific addresses. Recall that both buffers are exported via the `H` interface.

Virtual Memory Page Maps The IA32 supports a fairly straightforward two-level paging scheme. (It also has a separate, mostly orthogonal memory protection scheme based on *segments*; like most modern operating systems, we ignore segments almost entirely.) A *page table* contains entries mapping individual virtual pages to physical pages. A *page directory* maps contiguous ranges of virtual pages to page tables. Crucially, both page tables and (especially) page directories can be sparsely populated: a zero entry means the corresponding virtual addresses are unmapped. The `H` interface's `PageMap` corresponds to a page directory and its subsidiary page tables. Virtual-to-physical translation is performed using the *current directory*, which is pointed to by processor control register `CR3`. Page directories for user processes are installed into `CR3` by the `execContext` routine.

All the virtual address spaces have the same general layout. Assuming the 64MB memory size shown in Figure 6, the first 64MB of virtual addresses are mapped directly to identical physical addresses, and marked as accessible only to supervisor mode; i.e., user-mode processes cannot see them. Any memory-mapped I/O regions living at high physical memory addresses (e.g. the graphics frame buffer) are mapped to virtual addresses immediately above 64MB, again marked for supervisor-only access. Virtual addresses starting at 256MB (`0x100000000`) are mapped for access by user processes according to calls made to `setPageMap`. This scheme permits kernel code to access all physical memory (and any memory-mapped I/O regions) regardless of which page directory is current, so there is no need to switch back to a special page map when returning from user to supervisor mode. It is efficient to implement because page tables can be shared between multiple page directories; in this case, we arrange for every page directory to map the supervisor-accessible addresses to the *same* set of page tables.

Function `allocPageMap` simply grabs a page from the user-process page pool, zeroes it, and returns a pointer to it. Function `setPageMap` is a bit more complicated because it must take account of the two-level page map structure. If the virtual address being mapped lives in a range that is already mapped to a page table, then the corresponding entry in that table is written. If there is no appropriate page table, a fresh page must be taken from the pool, initialized, and entered in the page directory first. User page tables are never shared, as this would expose the two-level structure above the `H` interface. Changes to a page map will normally become visible the next time the page directory is set into `CR3`. However, if the target page directory of `setPageMap` is already installed in `CR3`, the function issues a `invlpg` instruction to invalidate (just) the changed mapping in the TLB.

The `H` interface deliberately lacks functions for freeing `PageMaps`; instead, we use GHC's weak pointers to detect when a `PageMap` is no longer referenced by the kernel. The page directory and all its subsidiary page tables can then be returned to the pool. (However, care must be taken *not* to free a page directory that is still installed in `CR3`!) Individual page tables are also reclaimed when they no longer contain any mappings. Similar techniques are used to allocate `PhysPages` from the user-process page pool, and to free them automatically when they are no longer referenced.

Executing Code in User Space Each call to `execContext` causes a transfer from supervisor to user mode and (eventually) back again. Implementing these mode transfers is an arcane affair; we give only a high-level description here. To transfer control to user mode, `execContext` begins by copying the fields of the `Context` record parameter into a C-level *context structure* (whose layout is dictated by the hardware) and setting `CR3` to point to the specified page directory (if different from the current one). It then saves its stack pointer in memory, sets `esp` to point to the context structure, and executes a return-from-interrupt (`rti`) instruction, which loads the user process state from the context structure and jumps to the `eip` recorded in that context. Control returns to the kernel when the user process is interrupted by a hardware fault, an external hardware interrupt, or a software exception instruction; all these cause the hardware to dump the current process state into the context structure and transfer control to a C exception handler. This handler copies the context structure to a new Haskell `Context` record, restores the saved stack pointer from memory, and returns from `execContext`.

I/O Implementation Because most of the I/O facilities in the `H` interface are relatively low-level, their implementation is quite simple. For example, the various functions to read and write I/O ports are implemented as single IA32 instructions.

As noted in Section 3.5, special processing is needed if an interrupt occurs while the processor is in supervisor mode. If `installHandler` has been used to specify a (Haskell) handler for the relevant IRQ, we must arrange for this handler to be run as soon as possible. To achieve this, we make use of an existing GHC RTS mechanism for delivering signals. The RTS maintains a queue of signal handlers to be activated (in fresh Concurrent Haskell threads) the next time that the garbage collector gains control because a heap limit check has failed. Our C handler simply inserts the registered Haskell interrupt handler into this queue. If no Haskell handler is registered, the C handler just sets an IRQ-specific flag that can be read by `pollInterrupts`.

Memory-mapped I/O regions and DMA buffer regions must be allocated as part of the system initialization process. For example, our current implementation uses information obtained from the BIOS via the GRUB boot loader to determine the physical memory address frame buffer, which it then remaps to a convenient virtual address and exposes (along with other graphics device information) through the `H` interface.

8. Conclusion and Future Work

We have successfully designed and implemented a monadic interface to the low-level hardware features of the IA32 architecture, and used it to build preliminary versions of a new, all-Haskell operating system (House) and also of an L4-compatible microkernel (Osker). This paper has focused on the challenges of building a real implementation of the interface. As another part of our ongoing research, we are also building formal models of the interface. These two threads of work will eventually come together as we build a validated Osker system. But even now, at an early stage in our project, they are already having significant influence on each another. The long term goal of formal validation has encouraged us to think carefully about the interfaces that we build. This had led us to consider, not just the types and the operations that should be provided, but also the properties that we expect or require them to satisfy. We have used P-Logic assertions of these properties to give a rich and precise definition of the interface.

Although we believe our current set of assertions to be both valid and useful, it is likely to be expanded and refined as we continue work on both sides of the `H` interface. Completing the formal model of `H` will allow us to verify the consistency of assertions and will suggest new ones. As we continue to develop Osker code, we will also be building proofs of key properties such as separation; this process will undoubtedly lead to further refinement of `H`'s properties as well.

Our experience using Haskell as a systems implementation language has been largely positive so far. We have found its type system and support for monads to be especially useful. The speed of compiled code has proved adequate for the purposes of the House system. However, key issues about performance remain to be answered. In particular, we have not yet determined whether devices requiring high bandwidth and low latency (e.g., network cards) can be adequately serviced in a Haskell runtime environment—particularly in the presence of garbage collection.

Haskell's type-based separation between pure and effectful computations is useful for documenting function interfaces, and we expect it to be helpful in simplifying proofs. Otherwise, there is no particular reason why this work could not be done in a different functional language. We have made no essential use of laziness in our code, so an eager language such as ML might seem a more natural choice. On the other hand, laziness does not appear to be costing us much in terms of time or space performance. A more compelling advantage of ML, especially for the specification work reported in this paper, may be that it supports proper module interfaces in the form of signatures.

Because our ultimate goal is high-assurance systems software, we need to be able to validate every component of our system, including the compiler and runtime system. We currently rely on the GHC compiler and RTS, which are large and complex, making such validation challenging. Because `H` (if used without implicit concurrency) makes no essential use of GHC-specific features, we expect to be able to port to a much simpler RTS (e.g., one providing just garbage collection). We are actively investigating mechanisms for building high-assurance collectors.

Acknowledgments

Sébastien Carlier and Jeremy Bobbio implemented hOp, which was one of our main inspirations. The code for Gadgets is due to Rob Noble. Iavor Diatchki wrote the NE2000 Ethernet and mouse drivers, and made many other helpful contributions. Other members of the Programatica team and the anonymous reviewers made useful suggestions about the presentation of this paper.

References

[1] F. Bellard. QEMU. `http://fabrice.bellard.free.fr/qemu/`.

[2] E. Biagioni, R. Harper, and P. Lee. A Network Protocol Stack in Standard ML. *Higher Order Symbol. Comput.*, 14(4):309–356, 2001.

[3] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3: Language definition. `http://www.research.compaq.com/SRC/m3defn/html/complete.html`.

[4] S. Carlier and J. Bobbio. hOp. `http://www.macs.hw.ac.uk/~sebc/hOp/`, 2004.

[5] A. Cheadle, T. Field, S. Marlow, S. Peyton Jones, and L. While. Exploring the Barrier to Entry: Incremental Generational Garbage Collection for Haskell. In *Int. Symp. on Memory Management*, pages 163–174, 2004.

[6] J. Cupitt. A brief walk thourgh KAOS. Techical Report 58, Computing Laboratory, Univ. of Kent at Canterbury, February 1989.

[7] Cyclone. `http://www.research.att.com/projects/cyclone/`.

[8] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proc. OASIS ASPLOS Workshop*, 2004.

[9] G. Fu. Design and implementation of an operating system in Standard ML. Master's thesis, University of Hawaii, August 1999.

[10] Glasgow haskell compiler. `http://www.haskell.org/ghc`.

[11] Grub. `http://www.gnu.org/software/grub/`.

[12] T. Hallgren. The House web page. `http://www.cse.ogi.edu/~hallgren/House/`, 2004.

[13] Intel Corp. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2004.

[14] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, June 2002.

[15] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, 1st Int. Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136, London, UK, 1995. Springer-Verlag.

[16] M. P. Jones. Bare Metal: A Programatica model of hardware. In *High Confidence Software and Systems Conference*, Baltimore, MD, March 2005.

[17] K. Karlsson. Nebula: A functional operating system. Technical report, Programing Methodology Group, University of Göteborg and Chalmers University of Technology, 1981.

[18] R. B. Kieburtz. P-logic: Property verification for Haskell programs. `ftp://ftp.cse.ogi.edu/pub/pacsoft/papers/Plogic.pdf`, August 2002.

[19] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL '95: Proc. 22nd ACM Symp. on Principles of programming languages*, pages 333–343, 1995.

[20] R. Noble. *Lazy Functional Components for Graphical User Interface*. PhD thesis, University of York, November 1995.

[21] The OSKit Project. `http://www.cs.utah.edu/flux/oskit/`.

[22] The Programatica Project home page. `www.cse.ogi.edu/PacSoft/projects/programatica/`, 2002.

[23] The SPIN project. `http://www.cs.washington.edu/research/projects/spin/www/`, 1997.

[24] W. Stoye. Message-based functional operating systems. *Science of Computer Programming*, 6:291–311, 1986.

[25] L. Team. *L4 eXperimental Kernel Reference Manual*, January 2005.

[26] Video Electronics Standards Association. *VESA BIOS EXTENSION (VBE) – Core Functions Standard, Version: 3.0*, September 1998. `www.vesa.org`.

[27] M. Wallace. *Functional Programming and Embedded Systems*. PhD thesis, Dept of Computer Science, Univ. of York, UK, January 1995.

[28] M. Wallace and C. Runciman. Lambdas in the Liftshaft – Functional Programming and an Embedded Architecture. In *FPCA '95: Proc. 7th Int. Conf. on Functional Programming Languages and Computer Architecture*, pages 249–258, 1995.