Operating Systems in Haskell: Implementations, Models, Proofs

Andrew Tolmach

Invited Professor, INRIA Rocquencourt

The Programatica Project

Portland State University

Iavor Diatchki, Thomas Hallgren, Bill Harrison, Jim Hook, Tom Harke, Brian Huffman, Mark Jones, Dick Kieburtz, Rebekah Leslie, John Matthews, Andrew Tolmach, Peter White, ...

An O/S in Haskell?

- Kernel (scheduler,resource management,etc.) written in Haskell
- Does privileged hardware operations (I/O, page table manipulation, etc.) directly
- (Some runtime system support, e.g. garbage collection, is still coded in C)
- Test case for **high-assurance** software development as part of **Programatica** project

Goals of High-Assurance Software Development

- Prevent exploitable bugs
  - e.g. no more buffer overrun errors!
- Match behavioral specifications
  - Requires development of specifications!
- Build systems with new capabilities
  - e.g. **multilevel secure systems** allow military applications at different security classifications to run on single machine with strong assurance of separation

Programatica Project

- High-assurance software by construction, rather than by post-hoc inspection
  - "Programming as if properties matter!"
- Rely on strongly-typed, memory-safe languages (for us, Haskell)
- Apply formal methods where needed
  - "Mostly types, a little theorem proving"
- Keep evaluation methodology in mind
  - Common Criteria for IT Security Evaluation

Structure of this talk

- Review of Haskell IO & monads
- P-Logic properties
- The H(ardware) Interface
- Implementing H on bare metal (with demo!)
- Modeling H within Haskell
- (Proofs)
- Ongoing & Related Work; Some Conclusions

Haskell: Safe & Pure

- Haskell **should** be good for high-assurance development
- Memory safety (via strong typing + garbage collection + runtime checks) rules out many kinds of bugs
- **Pure** computations support simple equational reasoning
- But...what about IO?

Haskell: IO Actions

- Haskell supports IO using **monads**.
- "Pure values" are separated from "worldly actions" in two ways
- Types: An expression with type `IO a` has an associated action, while also returning a value of type `a`
- Terms: The monadic `do` syntax allows multiple actions to be sequenced

IO Monad Example

- Read a character, echo it, and return a Boolean value that says if it was a newline:

```
do c <- getChar
   putChar c
   return (c == '\n')
```

- Makes use of primitive actions

```
getChar :: IO Char
putChar :: Char -> IO ()
return  :: a -> IO a
```

do Typing Details

:: IO ()
(actions without "v <- …" usually have this type)

:: Char

:: IO Char

```
do c <- getChar
   putChar c
   return (c == '\n')
```

:: IO Bool
(the type of the last action also determines the type of the entire do expression)

Building larger Actions

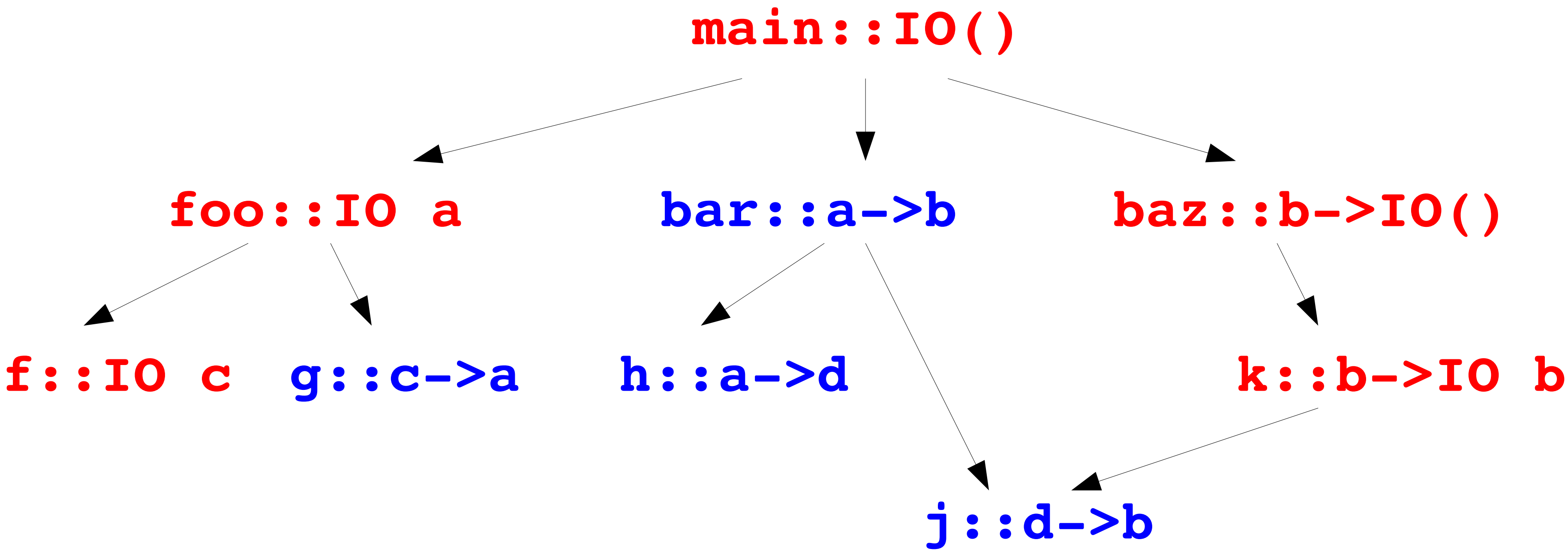
- We can build larger actions out of smaller ones, e.g. using recursion:

```
getLine :: IO String
getLine =
 do c <- getChar          -- get a character
    if c == '\n'          -- if it's a newline
       then return ""     -- then return empty string
       else do l <- getLine  -- otherwise get rest of
                             -- line recursively,
               return (c:l)  -- and return whole line
```

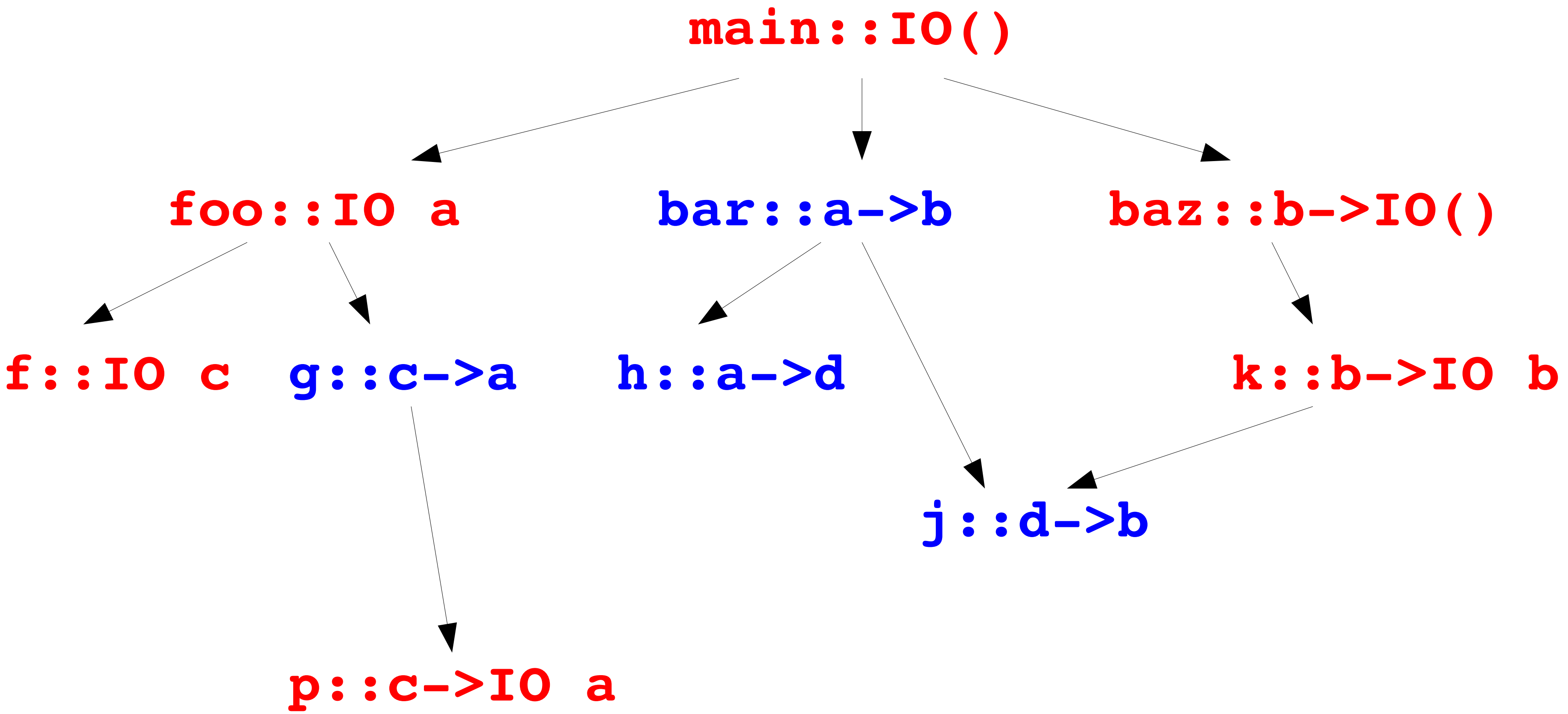
When are IO actions performed?

- A value of type `IO a` is an action, but it is still a value; it will only have an effect when it is **performed**
- In Haskell, a program's value is the value of `main`, which must have type `IO()`. The associated action will be performed when the **whole** program is run
- There is no way to perform an action corresponding to a subprogram by itself

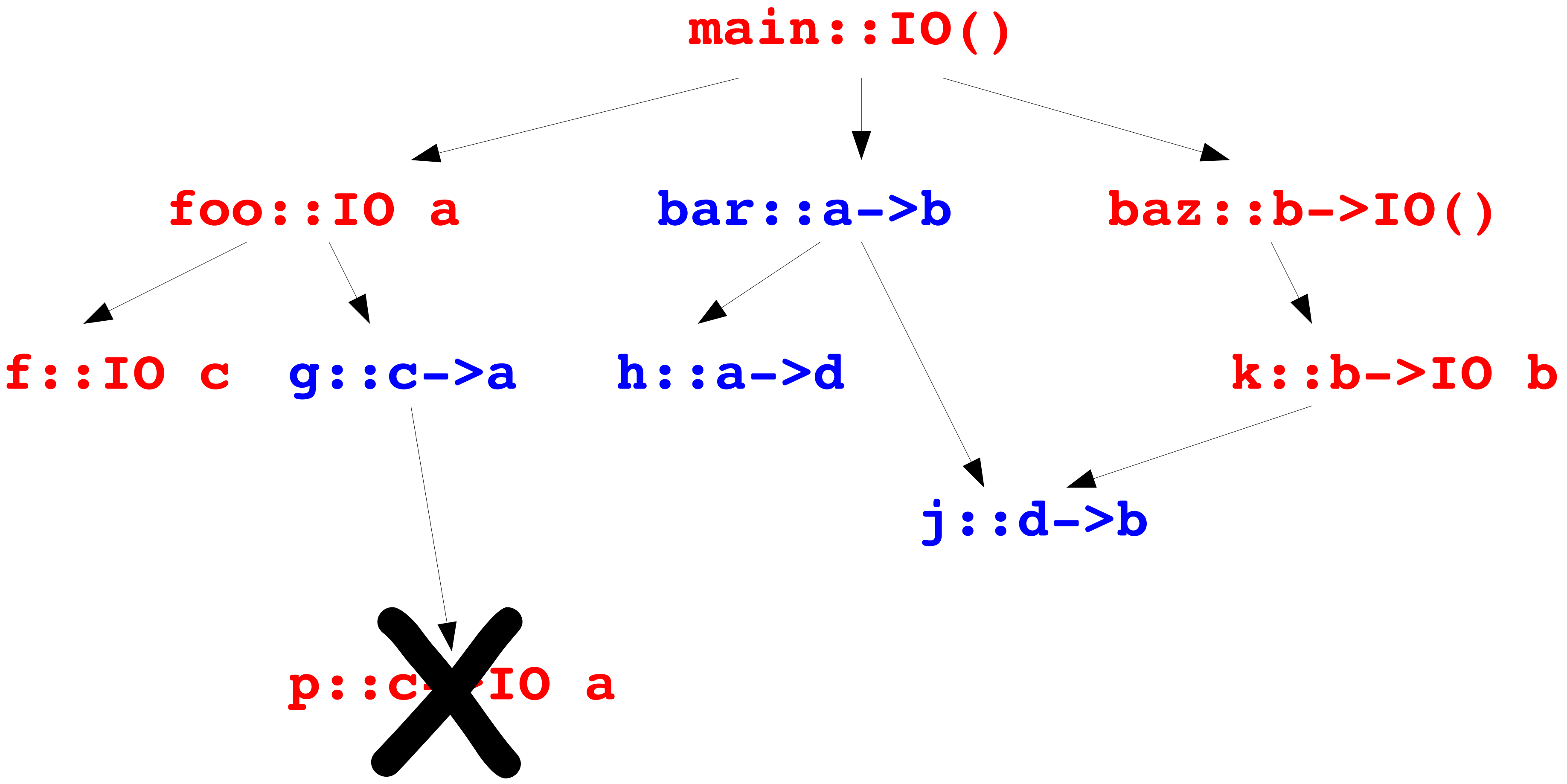
Overall Program Structure



Overall Program Structure



Overall Program Structure



IO Monad Hides Many Sins

- All kinds of impure/non-deterministic ops:
  - Mutable state (references and arrays)
  - Concurrent threads with preemption
  - Exceptions and signals
  - Access to non-Haskell functions using foreign function interface (FFI)

```
foreign import ccall "foo" Int -> IO Int
```

  - Uncontrolled memory access via pointers
- For high-assurance programming, we need to refine this monad

The H(ardware) Monad

- Small, specialized subset of GHC IO monad
- Primitives for privileged IA32 operations
  - Physical & Virtual memory
  - User-mode execution
  - Programmed and memory-mapped I/O
- Partially specified by P-Logic assertions
  - Different sorts of memory are independent
- Memory-safe (almost!)

Programatica Uses P-Logic

- Extend Haskell with type-checked property annotations
- P-Logic for defining properties/assertions, e.g.:

```
property Inverses f g =
   ∀ x . {f (g x)} === {x} ∧
         {g (f x)} === {x}
assert Inverses {\x->x+1} {\x->x-1}
```

- We have built support tools for handling properties and integrating provers, checkers, etc

Independence via Commutativity

```
property Commute f g =
  {do x <- f; y <- g; return (x,y)} ===
  {do y <- g; x <- f; return (x,y)}
property IndSetGet set get =
  ∀x. Commute {set x} {get}
property Independent set get set' get' =
      IndSetGet set get' ∧
      IndSetGet set' get ∧ ...
assert ∀p,p'.(p ≠ p') ⇒
   Independent {poke p} {peek p}
               {poke p'} {peek p'}
```

Summary of H types & operators

Physical memory

- PAddr
- PhysPage
- allocPhysPage
- getPAddr
- setPAddr

Virtual memory

- VAddr
- PageMap
- PageInfo
- allocPageMap
- getPage
- setPage

User-space execution

- Context
- Interrupt
- execContext

Programmed I/O

- Port
- inB/W/L
- outB/W/L

Memory-mapped IO

- MemRegion
- setMemB/W/L
- getMemB/W/L

Interrupts

- IRQ
- enable/disableIRQ
- enable/disableInterrupts
- pollInterrupts

H: Physical memory

- Types:

```
type PAddr = (PhysPage, Word12)
type PhysPage -- instance of Eq
type Word12
    -- unsigned 12-bit machine integers
```

- Operations:

```
allocPhysPage :: H (Maybe PhysPage)
getPAddr :: PAddr -> H Word8
setPAddr :: PAddr -> Word8 -> H()
```

H: Physical Memory Properties

- Each physical address is independent of all other addresses:

```
assert ∀pa,pa'.(pa ≠ pa') ⇒
    Independent {setPAddr pa}
                {getPAddr pa}
                {setPAddr pa'}
                {getPAddr pa'}
```

- (Not valid in Concurrent Haskell)

H: Physical Memory Properties(II)

- Each allocated page is distinct:

```
property Returns x =
   {| m | m === {do m; return x} |}
property Generative f =
 = ∀m.{do x <- f; m; y <- f;
          return (x == y)}
         ::: Returns {False}
assert Generative allocPhysPage
```

H: Virtual Memory

- Types and constants

```
type VAddr = Word32
minVAddr, maxVAddr :: VAddr
type PageMap  -- instance of Eq
data PageInfo =
    PageInfo{ physPage :: PhysPage,
              writable :: Bool,
              dirty :: Bool,
              accessed :: Bool }
```

H: Virtual Memory (II)

- Operations:

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

- Properties:

```
assert Generative allocPageMap
```

H: Virtual Memory Properties

- All page table entries are independent:

```
assert ∀pm,pm',va,va'.
  (pm ≠ pm' ∨ va ≠ va') ⇒
   Independent {setPage pm va}
               {getPage pm va}
               {setPage pm' va'}
               {getPage pm' va'}
```

- Page tables and physical memory are independent

H: User-space Execution

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

data Context =
 Context{eip,ebp,eax,...,eflags::Word32}
data Interrupt =
  I_DivideError | I_NMInterrupt| ... |
  I_PageFault VAddr |
  I_ExternalInterrupt IRQ |
  I_ProgrammedException Word8
```

Using H: A very simple kernel

```
type UProc = UProc { pmap :: PageMap, code :: [Word8],
                     ticks :: Int, ctxt :: Context, ...}
exec uproc =
 do (intrpt,ctxt') <- execContext (pmap uproc) (ctxt uproc)
    case intrpt of
      I_PageFault fAddr ->
         do fixPage uproc fAddr
            exec uproc{ctxt=ctxt'}
      I_ProgrammedException 0x80 ->
         do uproc' <- handleSyscall uproc{ctxt=ctxt'};
            exec uproc'
      I_ExternalInterrupt IRQ0 | ticks uproc > 1 ->
         return (Just uproc{ticks=ticks uproc-1,ctxt=ctxt'})
      _ -> return Nothing
```

Using H: Demand Paging

```
fixPage :: UProc -> VAddr -> H ()
fixPage uproc vaddr | vaddr >= (startCode uproc) &&
                      vaddr < (endCode uproc) =
  do let vbase = pageFloor vaddr
     let codeOffset = vbase - (startCode uproc)
     Just page <- allocPhysPage
     setPage (pmap uproc) vaddr
             (PageInfo {physPage = page, writable = False,
                        dirty = False, accessed = False})
     zipWithM_ setPAddr
               [(page,offset)|offset <- [0..(pageSize-1)]]
               (drop codeOffset (code uproc))
...
```

A User-space Execution Property

- Auxiliary property: conditional independence

```
property PostCommute f g = {| m |
{do m; x <- f; y <- g; return (x,y)} ===
{do m; y <- g; x <- f; return (x,y)} |}
```

- Changing contents of an unmapped physical address cannot affect execution

```
assert ∀pm,pa,c,x,m .
  m ::: NotMapped pm pa ⇒
  m ::: PostCommute {setPAddr pa x}
                    {execContext pm c}
```

Other User-space Properties

- If execution changes the contents of a physical address, that address must be mapped writable at some virtual address whose dirty and access flags are set
- (Execution might set access flag on any mapped page)

H: I/O Facilities

- Programmed I/O

```
type Port = Word16
inB :: Port -> H Word8
outB :: Port -> Word8 -> H()
```

  - and similarly for **Word16** and **Word32**

- Ports and physical memory are distinct

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

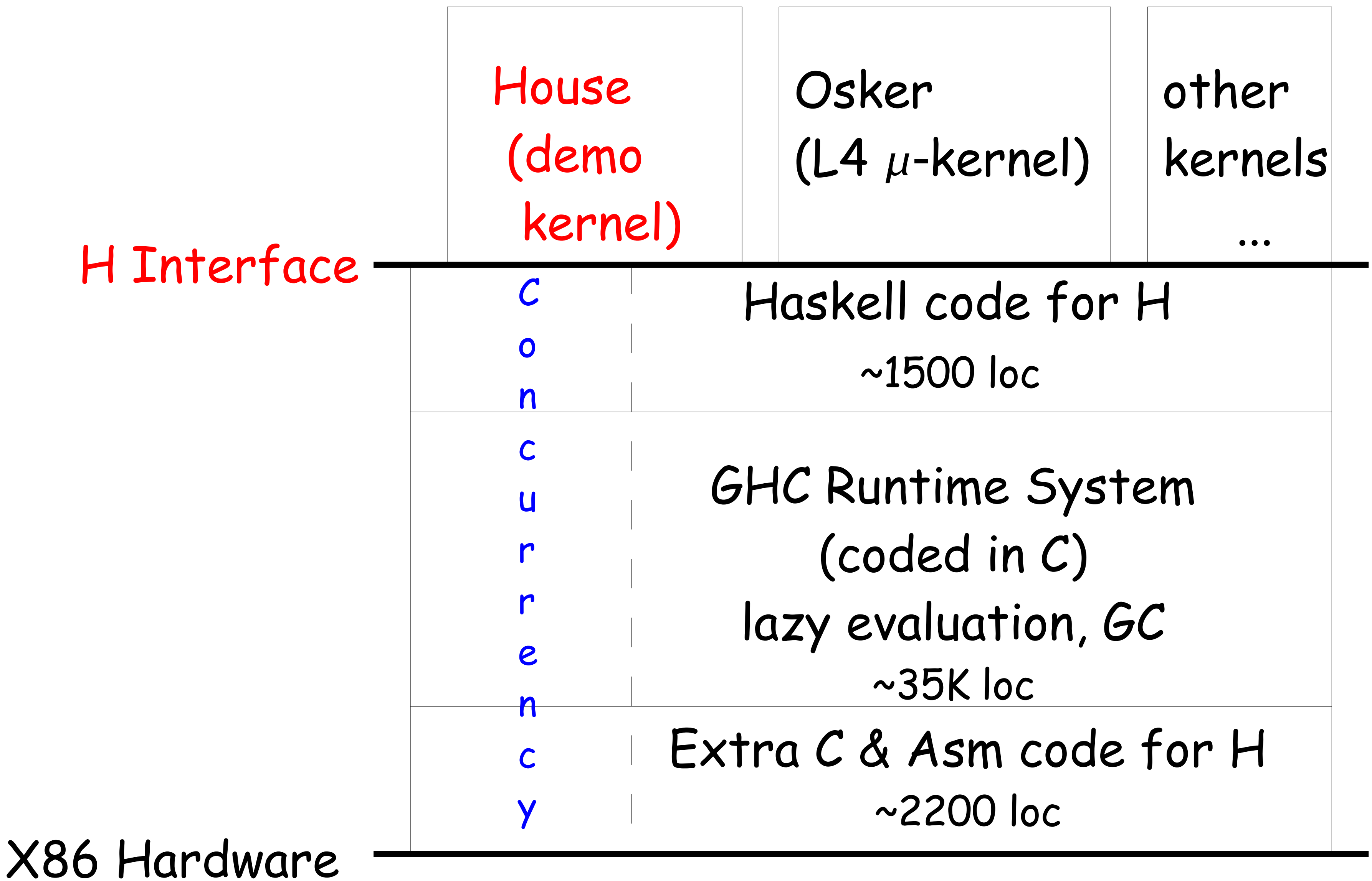
(except for buggy DMA!)

H: I/O Facilities (II)

- Memory-mapped I/O regions
  - Distinct from all other memory
  - Runtime bounds checks on accesses
- Interrupts

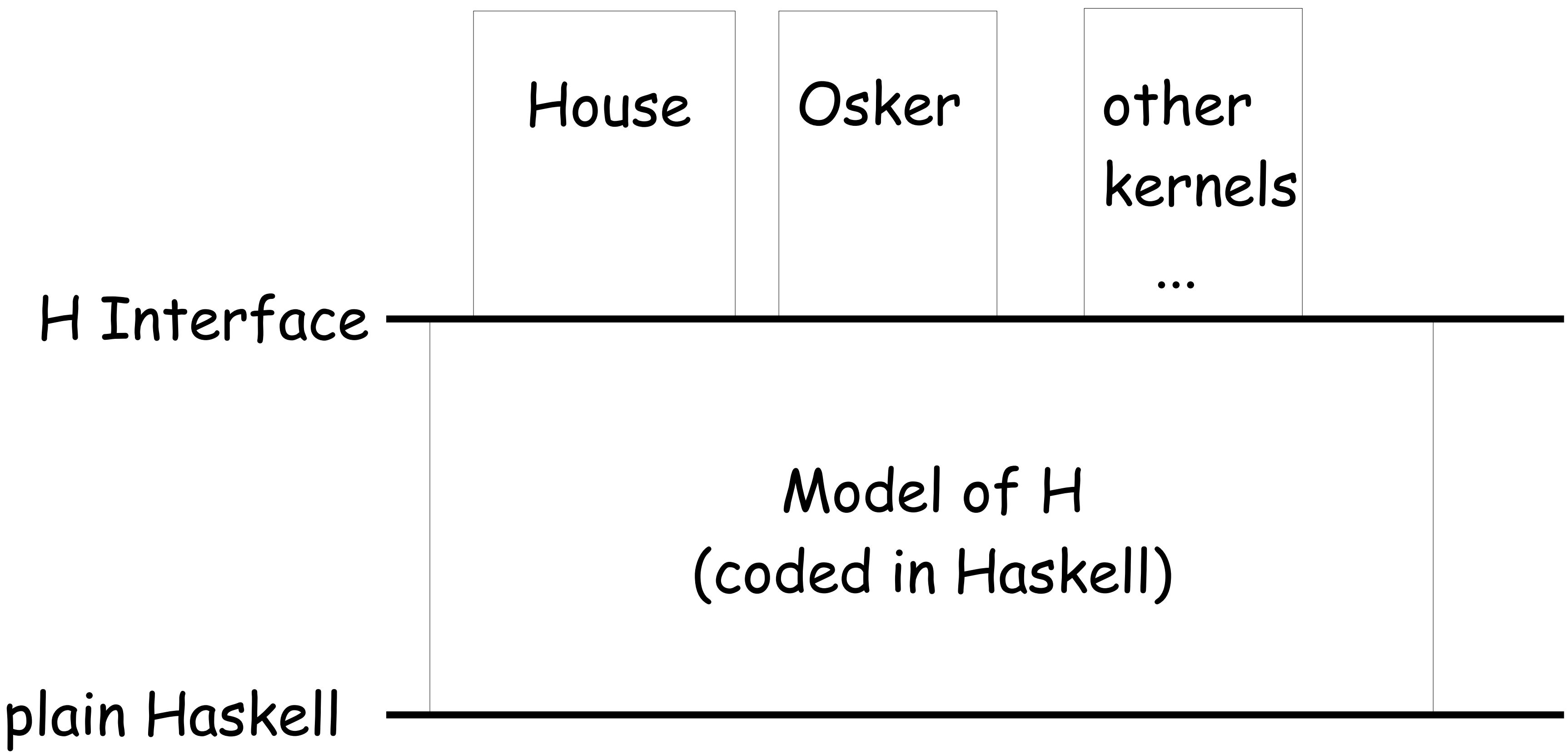
```
data IRQ = IRQ0 | ... | IRQ15
enableIRQ, disableIRQ :: IRQ -> H()
enableInterrupts,disableInterrupts :: H()
endIRQ :: IRQ -> H()
```

H on Real Hardware



H on Modeled Hardware

- Helps develop and check properties



House: A demonstration kernel

- Multiple user processes supported using GHC's Concurrent Haskell primitives
- Haskell device drivers for keyboard, mouse, graphics, network card (some from the **hOp project** [Carlier&Bobbio])
- Simple window system [Noble] and some demo applications, in Concurrent Haskell
- Command shell for running **`a.out`** binaries as protected user-spaces processes

hello.c

```
#include "stdlib.h"

static char n[] = "JFLA 2007";
main () {
  char *c = (char *) malloc(strlen(n+1));
  strcpy(c,n);
  printf("Bonjour %s!\n", c);
  exit(6*7);
}
```

loop.c

```
main () {
  for (;;);
}
```

div.c

```
main () {
  int a = 10 / (fib(5) - fib(5));
}

int fib(int x) {
  if (x < 2) return x;
  else return fib(x-1) + fib(x-2);
}
```

Why "House"?

Environment

Operating

User

Haskell

System

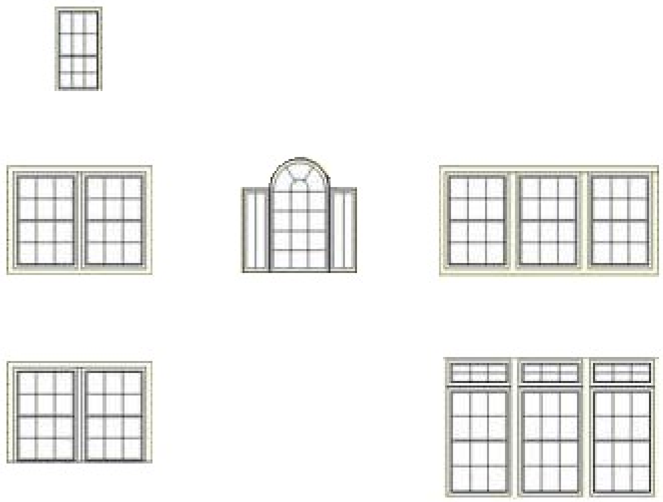
Why "House"?

- You are more secure in a House …

-

Why "House"?

- You are more secure in a House …



- … than if you only have Windows

Osker: A L4-based kernel

- L4 is a "second-generation" μ-kernel design
- Relatively simple, yet realistic
- Well-specified binary interface
- Multiple working implementations exist
- Can use to host multiple, separated versions of Linux
- No use of GHC concurrency in kernel
- Main target for separation proof

(hutte,bouge)

Hovel: A kernel for trying proofs

- Extremely simple, but still executable on real hardware
- Round-robin scheduler

```
schedule :: [UProc] -> H a
schedule [] = schedule []
schedule (u:us) =
  do r <- execUProc u
     case r of
      Just u' -> schedule (us++[u'])
      Nothing -> schedule us
```

Process Separation

- Define observable events

```
trace :: String -> H ()
```

  - outputs to a debug trace channel

- E.g. trace output system calls for a nominated process **u**

- Separation property is roughly

$\forall$**us.*trace*(schedule [u]) =**

**   *trace*(schedule (u:us))**

Formalizing Traces

- What does === mean for H computations?
  - H is a special monad that is not definable within Haskell
- Could take H properties as **axiomatization**
  - Complete? Consistent?
- Could give a separate semantics for H
  - Completely outside Haskell, or
  - **Modelled as an ADT within Haskell**

Modelling H with Traces

```
newtype H a = H (State -> (Trace,State,a))
```

Monad of state + output

```
type Trace = [String]
```

Potentially infinite stream

```
data State = {memory::Mem,
              interrupts::Oracle,...}
type Mem = PAddr -> Byte
type Oracle = [(Int,IRQ)]
```

How many cycles to wait until "delivering" next interrupt (IRQ).

```
runH :: State -> H a -> (Trace,State,a)
```

Using model instead of "real" H

- Instead of treating H in a special way (as ordinary Haskell treats IO), we install an implementation of the model as a monad:

```
instance Monad H where
  bind   = bindH
  return = returnH
```

- Allows us to use the do-notation "for free" :

```
do {x <- e1; e2}
```

is just syntactic sugar for

```
bind e1 (\x -> e2)
```

Defining H Model in Haskell

Cheating a little

```
type H a = State -> (Trace,State,a)

runH s h = h s

returnH x = \s -> ([],s,a)

bindH :: H a -> (a -> H b) -> H b

bindH h k = \s -> let (t1,s1,x1) = h s
                      (t2,s2,x2) = k x s1
                  in (t1 ++ t2,s2,x2)

trace w = \s -> ([w],s,())

allocPhysPage = \s -> ...

execContext pm c = \s -> ...
```

etc, etc...

Separation, More Formally

- Finally, a precise specification of separation:

```
∀state∀us.
    {fst(runH state (sched [u]))}
    ===
    {fst(runH state (sched (u::us)))}
```

- Needs to be guarded with assumptions about independence of **us**, adequate resources, etc.
- Now, how do we prove it...?

Ongoing work: Proof Approaches

- Pencil & paper proof sketch of separation for Hovel
  - Working on automation in Coq
- Automated translation of Haskell code into Isabelle/HOLCF
  - In progress; based on GHC Core
- Do we integrate programming & proving? Not yet!
- Related work for Haskell: Chalmers

Ongoing work: Operating Systems

- Completing the Osker separation kernel
- With Galois Connections: HALVM (Haskell Lightweight Virtual Machine) = GHC on Xen
- With Intel: Haskell modelling of another (proprietary) microkernel
- Other related work: seL4, Coyotos, Singularity, etc.

Ongoing work: Runtime Systems

- Large GHC RTS is big assurance headache
- Working to shrink and modularize RTS
- Current focus: proving correctness of GC
  - In context of Gallium Compcert project
  - Investigating existing systems for proving correctness of imperative pointer programs
- Other big goals: simple concurrency; safe foreign function interface

Which Kernel Concurrency Model?

Implicit House vs. Explicit Osker

Implicit (House)	Explicit (Osker)
(e.g.,using Concurrent Haskell)	
IRQ gets fresh thread	Must poll for IRQs
Natural kernel code	Kernel code all monadic
Simple properties fail	Properties should hold
No scheduler control (maybe being fixed in GHC)	Complete scheduler control
	Doesn't extend to MPs
`installHandler:: IRQ -> H() -> H()`	`pollInterrupts:: H [IRQ]`

Haskell for Systems Programming?

- To a first approximation, runtime efficiency is probably **not** very important for an OS!
- House works in spite of Haskell's limitations
  - Garbage collection any time
  - Laziness causes lots of overhead
  - Very hard to tune time & space performance
- But we **are** planning Systems Haskell dialect
  - Strict evaluation
  - Detailed control over data layout [Diatchki]
  - Related work: Cyclone project

Haskell for Execution & Modeling?

- Monadic ADT framework based on constructor classes works well
  - Easy to swap between “real” and “model” semantics for client code
  - Ability to change meaning of bind is key
- Lack of proper module system is a big problem
  - At the very least, need explicit interfaces

Haskell for Mechanized Proof?

- Haskell was a poor choice
  - Big language; had no formal semantics!
  - Laziness greatly complicates P-Logic
  - Types help but are too static
- But distinguishing pure and impure computations **is** a good idea
  - Related work: "Hoare type theory"
- Distinguishing terminating computations would probably be worthwhile too

Thank you!