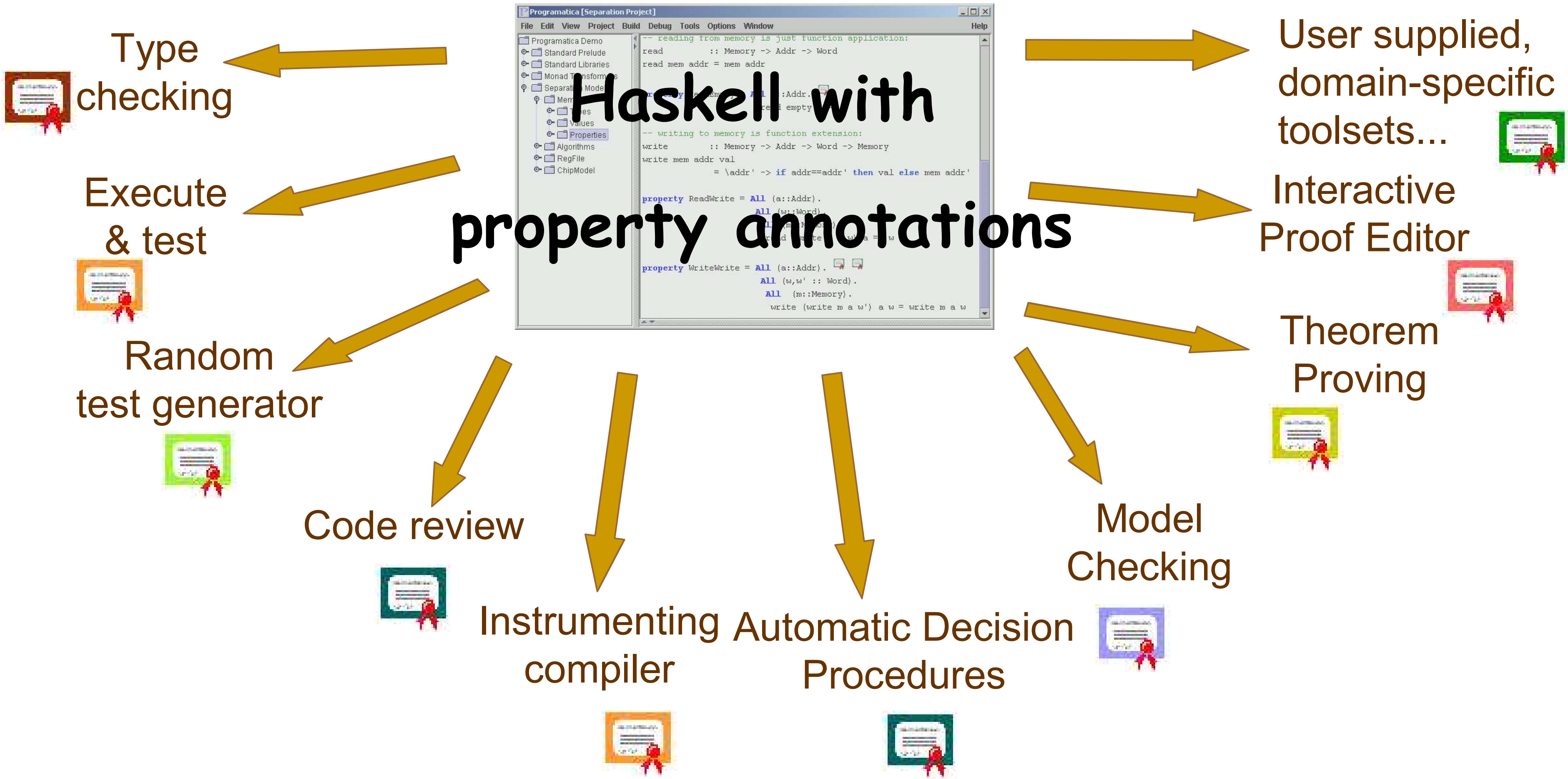
Proving Separation for a Working Microkernel Implementation

Andrew Tolmach
John Matthews
& the Programatica Project Team

PORTLAND STATE UNIVERSITY

OGI SCHOOL OF SCIENCE & ENGINEERING OHSU

The Programatica Vision



"Programming as if properties matter"

Our Ongoing Example Task

- Construct a realistic O/S μ-kernel in Haskell
- Use formal methods to demonstrate very high assurance of separation among user domains
  - esp. using Isabelle theorem prover

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 from Glasgow Haskell Compiler (GHC), including garbage collector, is still coded in C
- Why? Hope to leverage memory safety and increase assurance about key properties

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
- Side-effecting actions are given special typing and labeled as part of the **IO monad**

The 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)
  - Uncontrolled memory access via pointers
- So what level of assurance can we have about programs using the IO Monad ?

The H(ardware) Monad

- Constrained subset of GHC's 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

- Completely memory-safe (well, almost)

Programatica Uses P-Logic

- Extends Haskell with property annotations
- P-Logic used to define properties

```
property Inverses f g =
   ∀x . {f (g x)} === {x} ∧
        {g (f x)} === {x}
```

- and make assertions

```
assert Inverses {\x->x+1} {\x->x-1}
```

- (Tools for property management)

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
```

etc.

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, ctxt :: Context,
                     ticks :: Int, ...}
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))
...
```

Some User-space Properties

- Changing contents of an unmapped physical address cannot affect execution
- 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}
```

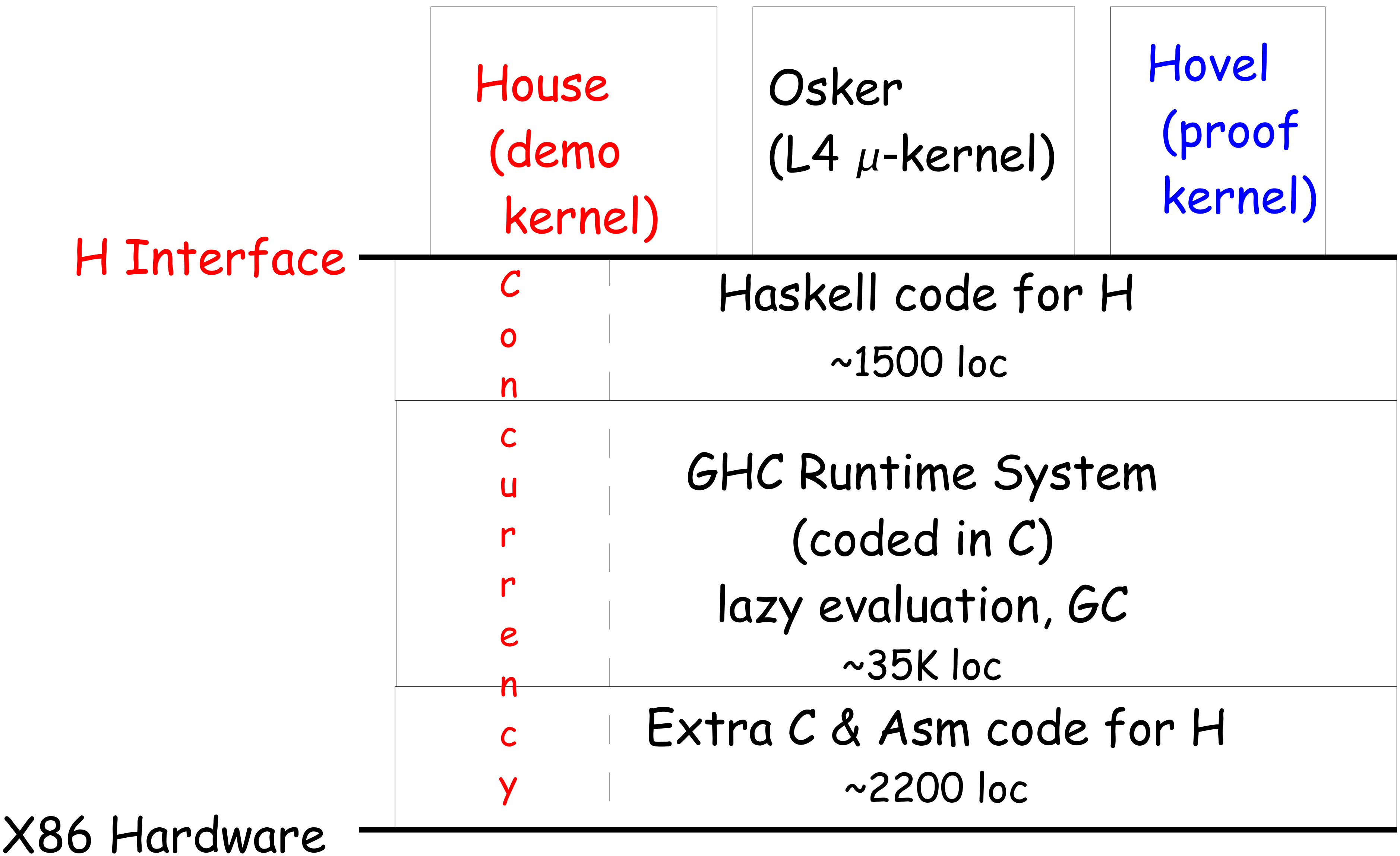
(except for rogue DMA!)

H: I/O Facilities (II)

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

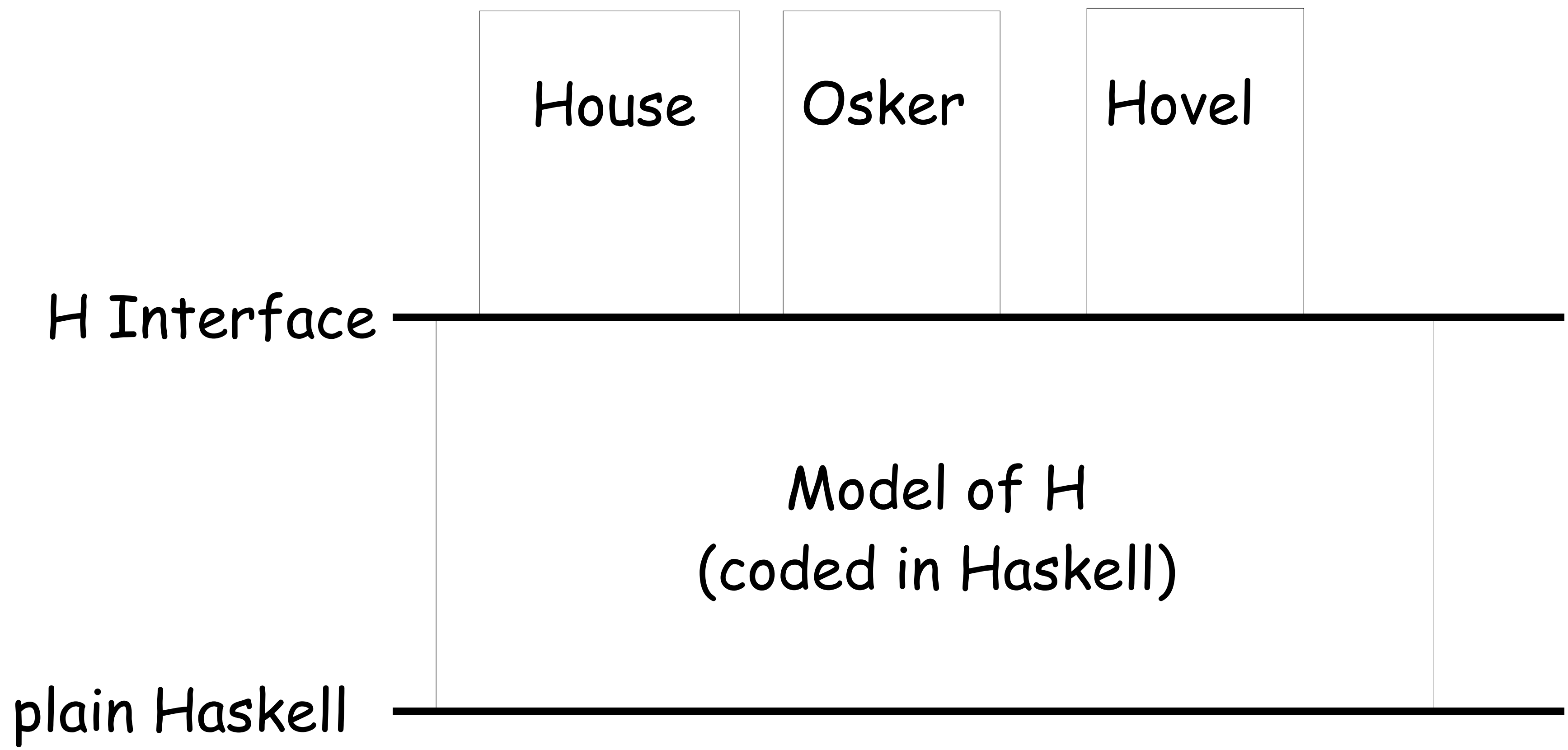
```
data IRQ = IRQ0 | ... | IRQ15
enableIRQ, disableIRQ :: IRQ -> H()
enableInterrupts,disableInterrupts :: H()
endIRQ :: IRQ -> H()
```

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

hello.c

```
#include "stdlib.h"

static char n[] = "HCSS participants";
main () {
  char *c = (char *) malloc(strlen(n+1));
  strcpy(c,n);
  printf("Hello %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);
}
```

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
- Main target for separation proof

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.**<u>*trace*</u>**(schedule [u]) =**

  <u>*trace*</u>**(schedule (u:us))**

Formalizing Traces

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

Modelling H with Traces

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

Monad of state + output

```
type Trace = [String]
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 :: Mem -> Oracle -> H a -> (Trace,a)
```

Separation, More Formally

- A plausible statement of separation:

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

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

Roadmap for rest of the talk

- Proving properties with Isabelle theorem prover
- Haskell→Isabelle translator
- Some example translations
- Current challenges
- Next steps
- Collaboration with Galois Connections

Proving Properties with Isabelle

- Isabelle is an "LCF-style" theorem prover
  - Trusted code base is just a few pages of SML code
  - Can generate explicit proof objects
  - Proof objects could be checked by external tool
- Isabelle has good evidence management support
  - Evidence generated by external oracles
  - Evidence tracked by tagged theorems

Haskell→Isabelle Translator

- Automatically translates:
  - Haskell definitions to Isabelle definitions
  - P-logic assertions to Isabelle predicates
- User then writes a separate proof script file to verify the assertions against the Isabelle definitions
- Motivation
  - Prove security properties of Osker
  - Prove properties of other security-critical Haskell programs
    - Trusted Services Engine
    - ASN.1 compiler

Formal Haskell Semantics

- Most pencil-and-paper Haskell proofs use domain theory
  - Permits equational reasoning about Haskell programs
  - Can define general recursive functions without needing termination proofs
  - Can define higher order and infinite data structures
- So we chose Isabelle/HOLCF

  = Isabelle + HOL+ LCF (Domain Theory)

Haskell Features Currently Supported in Isabelle/HOLCF

- Mutually recursive and infinite datatypes
- Anonymous functions with patterns
- Case-expressions with nested patterns
- Type classes (single-parameter only)
- Haskell modules (but no import/export lists)
- Contravariant datatype recursion (i.e. data Val = Fn (Val → Val) | …)
- Recursive let-bindings with patterns
- Named recursive functions with patterns
- Do-notation (for a fixed monad)

Haskell Syntax in Isabelle

- Isabelle's syntax is user-extensible
- So we added "Haskell brackets" to HOLCF for both terms and types
  - Translated Isabelle definitions now very close to original Haskell code

Example 1: Schedule translation

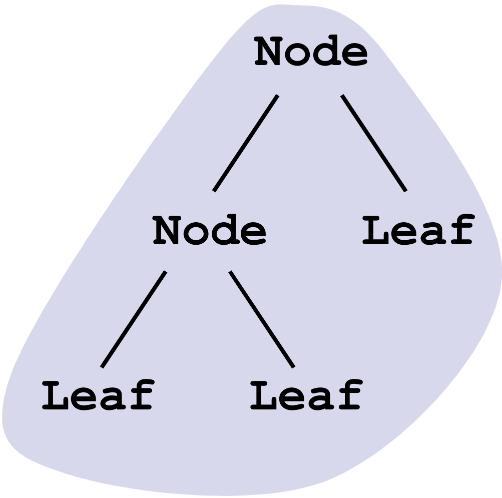
Haskell

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

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

Example 2: Serializing Binary Trees

- Inspired by
  [Slind and Hurd 2003]




```
data Tree = Leaf | Node Tree Tree


encode_Tree :: (Tree, [Bool]) -> [Bool]
encode_Tree (Leaf, bs) = False : bs
encode_Tree (Node l r, bs)
   = True : encode_Tree (l, encode_Tree (r, bs))

decode_Tree :: [Bool] -> (Tree, [Bool])
decode_Tree (False : bs) = (Leaf, bs)
decode_Tree (True : bs) =
  case decode_Tree bs of
    (l, bs1) -> case decode_Tree bs1 of
      (r, bs2) -> (Node l r, bs2)
```

Example 2: Translated HOLCF Theory

```
domain Tree = Leaf | Node (lazy "<<Tree>>") (lazy "<<Tree>>")

consts
  encode_Tree :: "<<(Tree, [Bool]) -> [Bool]>>"
fixrec
  "<<encode_Tree (Leaf, bs) = False : bs>>"
  "<<encode_Tree (Node l r, bs)
       = True : encode_Tree (l, encode_Tree (r, bs))>>"

consts
  decode_Tree :: "<<[Bool] -> (Tree, [Bool])>>"
fixrec
  "<<decode_Tree (False : bs) = (Leaf, bs)>>"
  "<<decode_Tree (True : bs)
        = case decode_Tree bs of
                 {(l, bs1) -> case decode_Tree bs1 of
                      {(r, bs2) -> (Node l r, bs2)}}>>"
```

Example 2: P-logic Assertion

- Serializing a fully-defined binary tree and then deserializing it, results in the original tree.

```
assert DecodeEncode =
  All t . All bs .
    {defined_Tree t} === {()}
    ==>
    {decode_Tree (encode_Tree (t, bs))} === {(t, bs)}
```

Example 2: Translated HOLCF Predicate

```
locale DecodeEncode =
 assumes
     "∀t bs.
      <<defined_Tree t = ()>>
      ==>
      <<decode_Tree (encode_Tree (t, bs)) = (t, bs)>>"
```

Example 2: Proof Script for Translated HOLCF Predicate

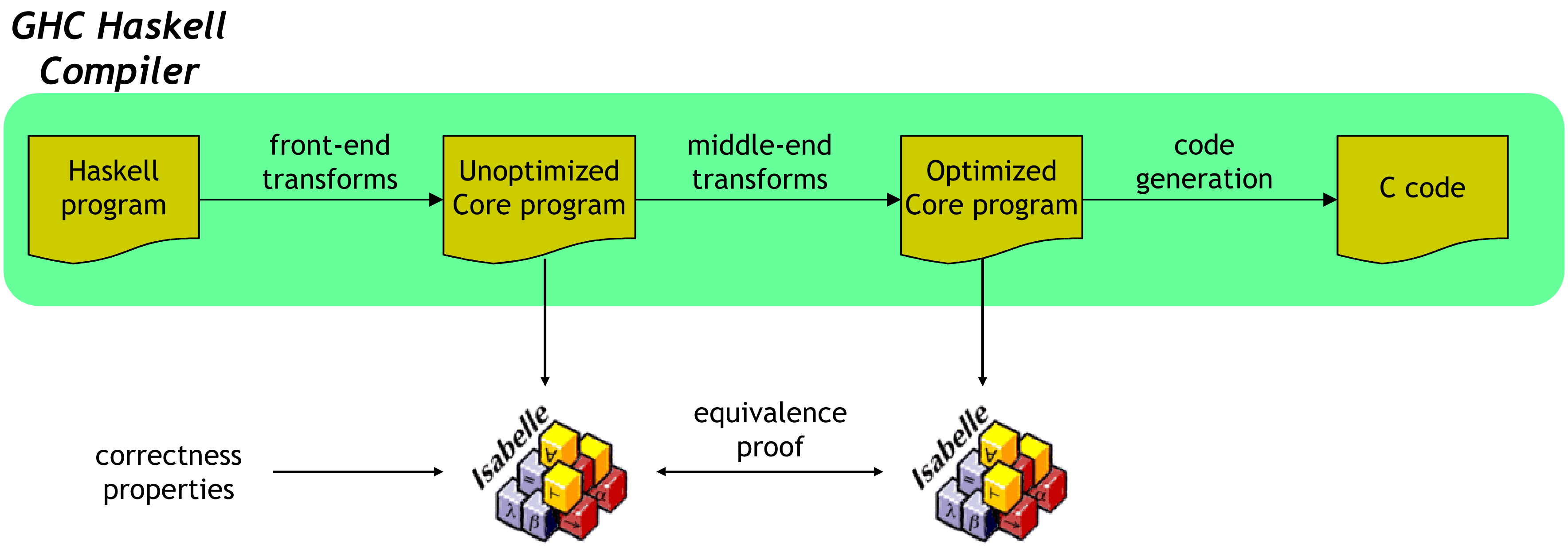
```
theory CoderProofs
imports Coder HaskellLemmas
begin

fixpat defined_Tree_strict[simp]:
  "<<defined_Tree undefined>>"

lemma DecodeEncode: "DecodeEncode"
 apply (rule DecodeEncode.intro)
 apply (rule allI)
 apply (rule_tac x=t in Tree.ind)
 by (auto simp add: Case_hprod_split_eq)

end
```

Current Challenges

1. Does our translation match what the Haskell compiler does?
2. Haskell's type system is much more sophisticated than Isabelle's type system
   - Constructor classes, multi-parameter type classes, higher-rank polymorphism, existential types, etc.
   - Prelude files used by Osker depend on some of these features

Our Plan to Tackle Challenge 1

- Model GHC core in Isabelle
  - Now can model everything GHC can compile
  - Removes Haskell compiler front-end, middle-end from trusted computing base
  - GHC-Core evolves more slowly than GHC Haskell.



Key Issue

- GHC-Core is lower-level than Haskell
  - Solution: Extend Isabelle's pretty-printers to help recover original Haskell syntactic sugar
  - Inspired by similar approach in Cover and Sparkle projects

Our Plan to Tackle Challenge 2

- Translate Haskell types into Partial Equivalence Relations (PERs)
  - PERs also used by [Homeier 2005]
- Allows us to:
  - Translate the full range of Haskell's type system, including existential types, higher-rank polymorphism, unpointed types
  - Precisely model Haskell modules and abstract datatypes
  - Prove that H monad satisfies monad laws

Next steps

- Prove separation properties of Hovel bottom-up
- Start with fixPage separation property:
  - "The page installed by fixPage is globally fresh"
  - Needed to show that no two process's virtual pages are aliased to the same physical page
- Use Isabelle's evidence management to track H monad properties used in proofs

Programatica Collaboration with Galois Connections

- Programatica HOLCF extensions used to model Galois' Cryptol™ semantics

Example: factorial (mod 2^8)

μCryptol

```
fac : B^32 -> B^8;
fac i = facs @@ i
  where {
    rec
      idx : B^8^inf;
      idx = [1] ## [x + 1 | x <- idx];
    and
      facs : B^8^inf;
      facs = [1] ## [x * y | x <- facs
                           | y <- idx]; };
```

```
consts fac :: "nat lift → nat lift"
fixrec
  "fac·i
   = (Letrec
        idx = [:1{{8}}:] ## {env. env·''x'' +{{8}} 1{{8}}
                              | ''x'' ← idx};
        facs = [:1{{8}}:] ## {env. env·''x'' *{{8}} env·''y''
                               | ''x'' ← facs
                               | ''y'' ← idx}
      in facs @@ i)"
```

Summary and Future Work

- H interface encapsulates privileged HW
- Multiple kernels based on H
- Ongoing separation proofs for Hovel, Osker
- Automated translation to Isabelle
- High-assurance RTS (esp. GC)
- Refinements to Haskell

Example 3: fixPage Translation

```
consts
  fixPage :: "<<UProc -> VAddr -> H (Either UProc String)>>"
fixrec
  "<<fixPage uproc vaddr =
       if vaddr `inVRegion` (codeRegion $ aout uproc) then
       do { page <- setupPage uproc vaddr False;
            copyPage page (codeBytes (aout uproc) .
              (+ fromIntegral
                   (pageFloor vaddr -
                      (fst $ (codeRegion $ aout uproc)))));
            return $ Left (ownPage page uproc)}
       else if vaddr `inVRegion` (dataRegion $ aout uproc) then
       do { page <- setupPage uproc vaddr True;
            copyPage page (dataBytes (aout uproc) .
              (+ fromIntegral
                   (pageFloor vaddr -
                     (fst $ (dataRegion $ aout uproc)))));
            return $ Left (ownPage page uproc)}
```

Example 3: fixPage Translation

```
else if vaddr `inVRegion`
        (fst $ (bssRegion $ aout uproc), brk uproc) then
do { page <- setupPage uproc vaddr True;
     zeroPage page;
     return $ Left (ownPage page uproc)}
else if vaddr `inVRegion` stackRegion uproc then
do { page <- setupPage uproc vaddr True;
     zeroPage page;
     return $ Left (ownPage page uproc)}
else return $ Right (''Fatal page fault at '' ++
                     showHex vaddr '''')>>"
```