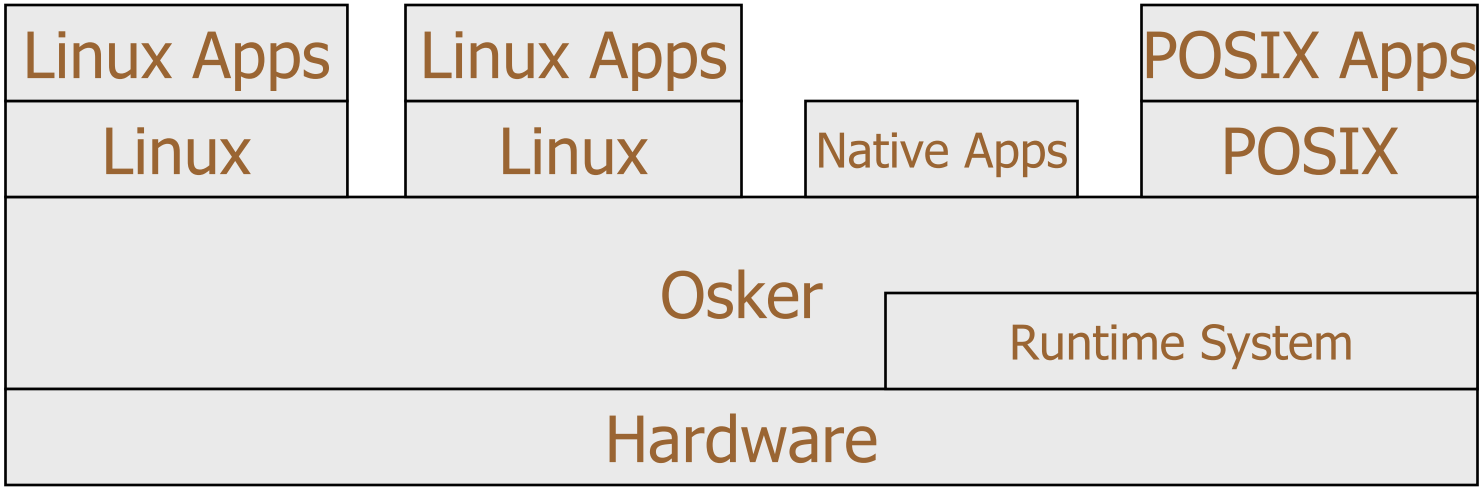
Building a High-Assurance Separation Kernel using Programatica

Mark P Jones

High Confidence Software & Systems
March 2005

OGI SCHOOL OF SCIENCE & ENGINEERING OHSU
PORTLAND STATE UNIVERSITY
UNIVERSITY OF MISSOURI-COLUMBIA

The Oregon Separation Kernel:



Linux Apps
Linux Apps
POSIX Apps
Linux
Linux
Native Apps
POSIX
Osker
Runtime System
Hardware

A working μ-kernel implementation with very high assurance of separation between domains

(interference only as permitted by explicit policy)

How do you build things like this?

- Good design, good architecture
- Sound engineering, informal reasoning
- Cope with an abundance of small details
- Rely on behavior of underlying platform
- The Programatica Approach:

  “Programming as if properties matter”

How do we build things like this?

- Good design, good architecture
  - Reuse the good ideas!
  - Monads, ADTs: “separation by construction”

How do we build things like this?

- Good design, good architecture

- Sound engineering, informal reasoning
  - Capture specifications/programmer expectations as embedded properties (“Extreme Formal Methods”)
  - Integrate with (formal & informal) validation tools

How do we build things like this?

- Good design, good architecture
- Sound engineering, informal reasoning
- Cope with an abundance of small details
  - Raise the level of abstraction
  - Leverage types: “mostly types, a little proving”

How do we build things like this?

- Good design, good architecture
- Sound engineering, informal reasoning
- Cope with an abundance of small details
- Rely on behavior of underlying platform
  - "Trusted hardware" … but trusted to do what?
  - Formalize and document assumptions

Ingredients:

- **Programatica:**
  Certification
- **Haskell:**
  Modeling, implementation, tractable reasoning
- **L4:**
  Keeping it real
- **House:**
  Feasibility, prototyping

Programatica

Programatica Goals:

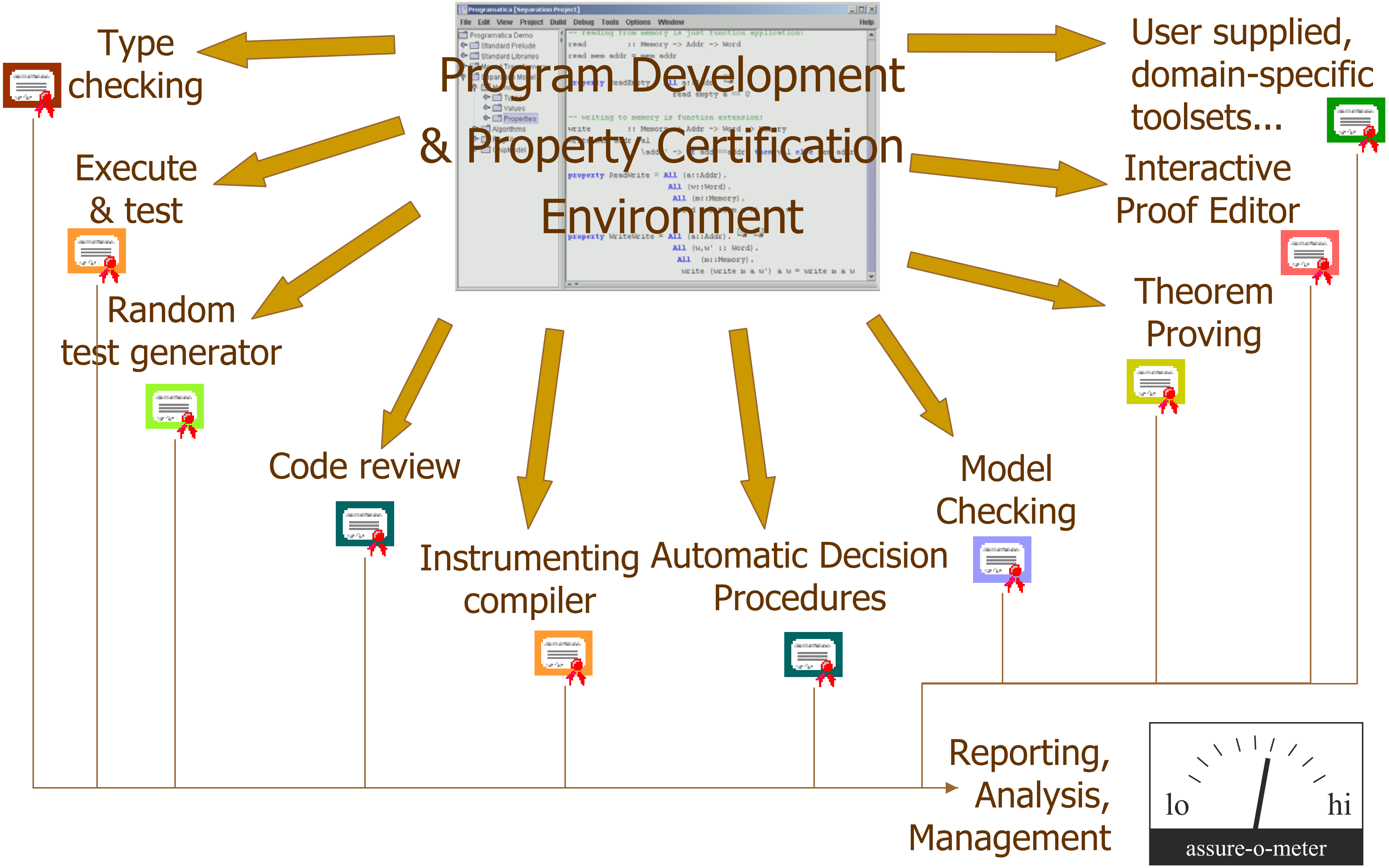
- **Develop** methodologies, tools, and foundations to support the construction and certification of high-assurance systems

- **Integrate** a broad and open spectrum of assurance techniques (code review, testing, formal methods, …)

- **Support** evolving code, evidence, and assurance requirements (e.g., track dependencies, revalidate, …)

- **Apply** to assurance of security properties in complex software artifacts of engineering significance

The Programatica Vision:



```
-- reading from memory is just function application:
read          :: Memory -> Addr -> Word
read mem addr = mem addr

property ReadEmpty [illegible]
                      read empty a == 0

-- writing to memory is function extension:
write         :: Memory -> Addr -> Word -> Memory
write mem addr val
              \addr' -> [illegible] addr'==addr then val else mem addr'

property ReadWrite = All (a::Addr).
                     All (w::Word).
                      All (m::Memory).
                       [illegible]

property WriteWrite = All (a::Addr).
                       All (w,w' :: Word).
                        All (m::Memory).
                         write (write m a w') a w = write m a w
```

Program Development
& Property Certification
Environment

Type
checking

Execute
& test

Random
test generator

Code review

Instrumenting
compiler

Automatic Decision
Procedures

Model
Checking

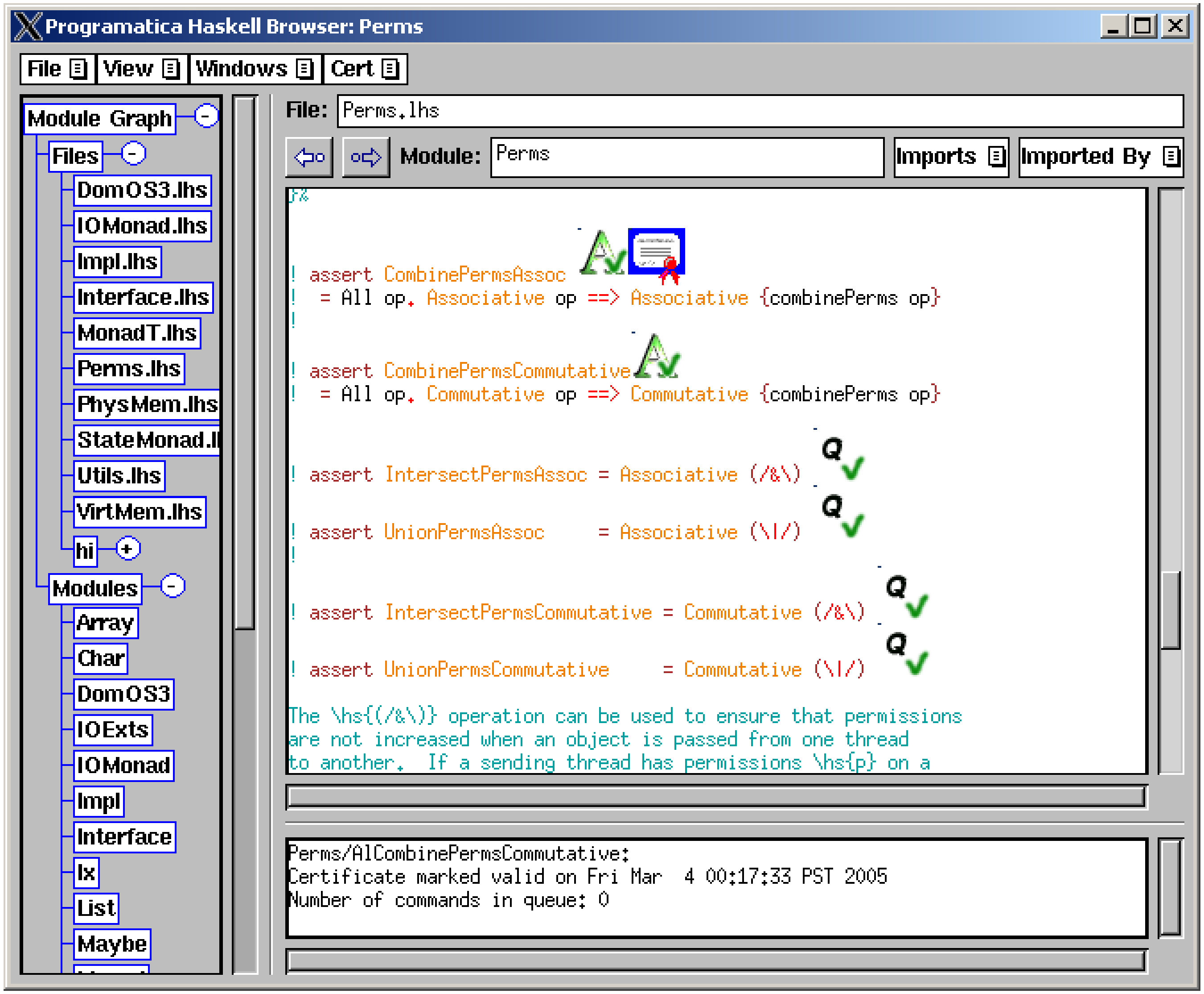
Theorem
Proving

Interactive
Proof Editor

User supplied,
domain-specific
toolsets...

Reporting,
Analysis,
Management

The Programatica Browser:



File: Perms.lhs

Module: Perms

```
}%

! assert CombinePermsAssoc
!  = All op. Associative op ==> Associative {combinePerms op}
!

! assert CombinePermsCommutative
!  = All op. Commutative op ==> Commutative {combinePerms op}


! assert IntersectPermsAssoc = Associative (/&\)

! assert UnionPermsAssoc     = Associative (\|/)
!

! assert IntersectPermsCommutative = Commutative (/&\)

! assert UnionPermsCommutative     = Commutative (\|/)

The \hs{(/&\)} operation can be used to ensure that permissions
are not increased when an object is passed from one thread
to another.  If a sending thread has permissions \hs{p} on a
```

```
Perms/AlCombinePermsCommutative:
Certificate marked valid on Fri Mar  4 00:17:33 PST 2005
Number of commands in queue: 0
```

Programatica Servers:

- "I say so"
  - A person signs their name by an assertion
- Testcases
  - Individual test cases / regression testing
- QuickCheck
  - Random testing
- Plover
  - The P-logic verifier
- Alfa
  - Interactive proof editor based on type theory

implemented,
automated,
maturing

- Isabelle
  - Logical framework, tactic-based theorem prover

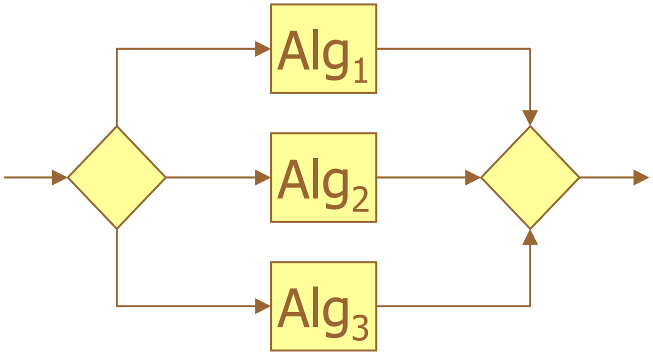
hand
translation

- Model Checking of Monadic Code

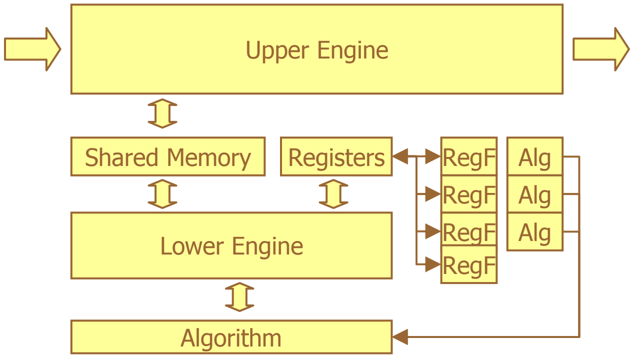
new
development

Early Case Study:

- Based on a Hypothetical Crypto Chip Design



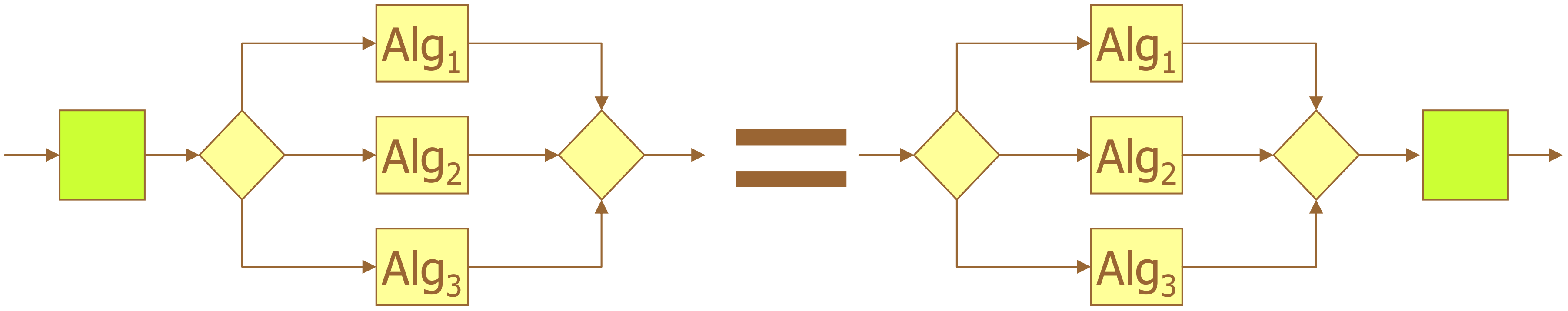
Conceptual View



Implementation

- Modeled in Haskell (~260 LOC)
- GUARANTEED separation between channels

The Separation Property:



```
assert Separation
    = All algs :: Algs.
          All select :: (ChannelId → Bool).
              { filter (select . fst) . chip algs }
                 ===
              { chip algs . filter (select . fst) }
```

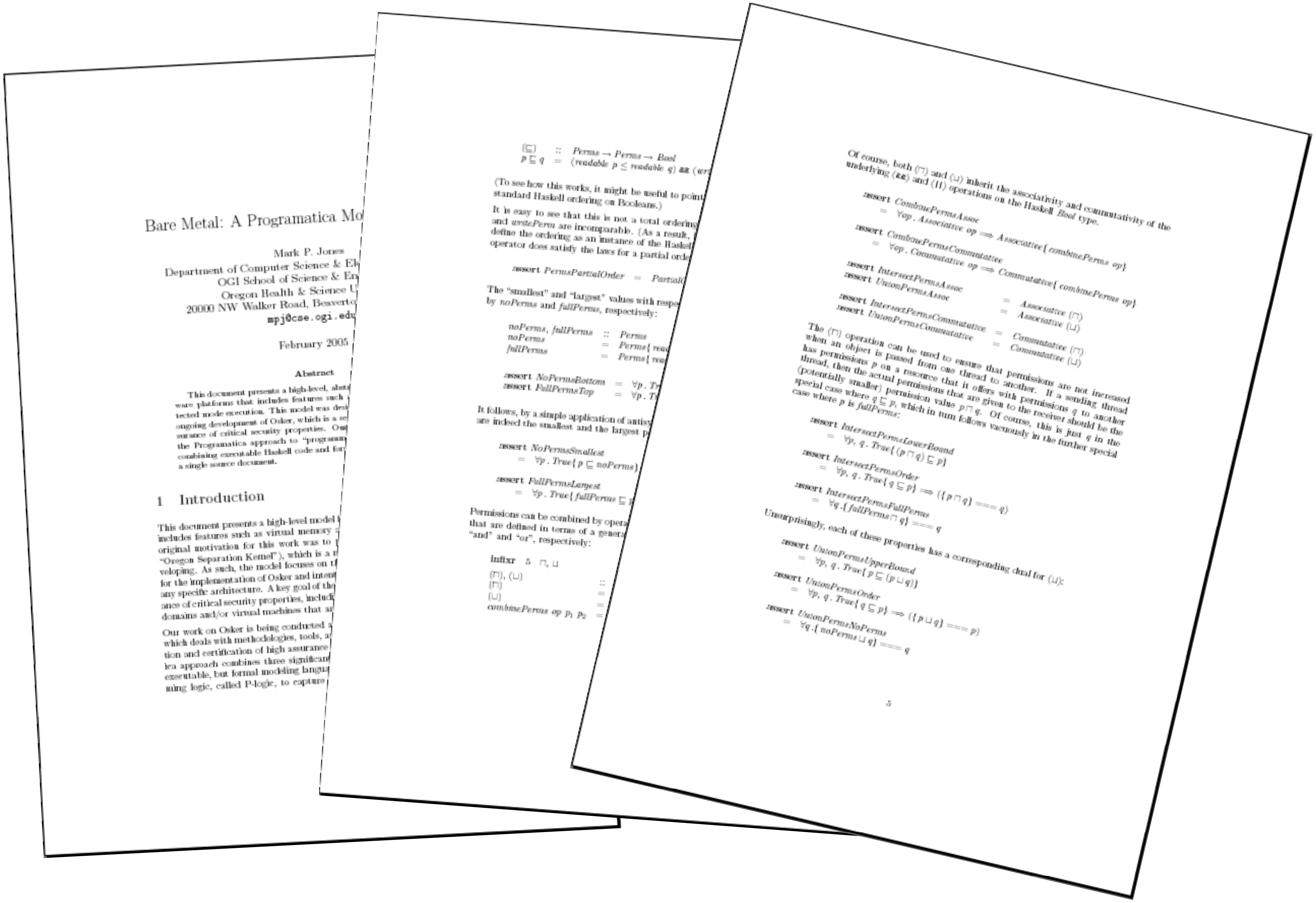
- Concluded with formal proof using the Alfa server

Haskell

Haskell:

- An expressive, purely functional programming language
- A semantically rich, formal modeling language
- A “(semi-) formal method”

Design Document:



Bare Metal: A Programatica Mo

Mark P. Jones
Department of Computer Science & El
OGI School of Science & En
Oregon Health & Science U
20000 NW Walker Road, Beaverto
`mpj@cse.ogi.edu`

February 2005

Abstract

This document presents a high-level, abstr
ware platforms that includes features such
tected mode execution. This model was desi
ongoing development of Osker, which is a se
surance of critical security properties. Ou
the Programatica approach to "programm
combining executable Haskell code and for
a single source document.

1 Introduction

This document presents a high-level model
includes features such as virtual memory
original motivation for this work was to
"Oregon Separation Kernel"), which is a t
veloping. As such, the model focuses on t
for the implementation of Osker and inten
any specific architecture. A key goal of th
ance of critical security properties, includ
domains and/or virtual machines that a

Our work on Osker is being conducted
which deals with methodologies, tools, a
tion and certification of high assurance
ica approach combines three significan
executable, but formal modeling langu
ming logic, called P-logic, to capture

```
(⊑)    ::  Perms → Perms → Bool
p ⊑ q  =   (readable p ≤ readable q) && (wr
```

(To see how this works, it might be useful to point
standard Haskell ordering on Booleans.)

It is easy to see that this is not a total ordering
and *writePerm* are incomparable. (As a result,
define the ordering as an instance of the Haskel
operator does satisfy the laws for a partial orde

```
assert PermsPartialOrder  =  Partial(
```

The "smallest" and "largest" values with respe
by *noPerms* and *fullPerms*, respectively:

```
noPerms, fullPerms  ::  Perms
noPerms             =   Perms{ rea
fullPerms           =   Perms{ rea

assert NoPermsBottom  =  ∀p . Tr
assert FullPermsTop   =  ∀p . Tr
```

It follows, by a simple application of antisy
are indeed the smallest and the largest p

```
assert NoPermsSmallest
  =  ∀p . True{ p ⊑ noPerms}

assert FullPermsLargest
  =  ∀p . True{ fullPerms ⊑ p
```

Permissions can be combined by opera
that are defined in terms of a genera
"and" and "or", respectively:

```
infixr  5  ⊓, ⊔

(⊓), (⊔)              ::
(⊓)                   =
(⊔)                   =
combinePerms op p1 p2 =
```

Of course, both (⊓) and (⊔) inherit the associativity and commutativity of the
underlying (&&) and (||) operations on the Haskell *Bool* type.

```
assert CombinePermsAssoc
  =  ∀op . Associative op ⟹ Associative{ combinePerms op}

assert CombinePermsCommutative
  =  ∀op . Commutative op ⟹ Commutative{ combinePerms op}

assert IntersectPermsAssoc        =  Associative (⊓)
assert UnionPermsAssoc            =  Associative (⊔)

assert IntersectPermsCommutative  =  Commutative (⊓)
assert UnionPermsCommutative      =  Commutative (⊔)
```

The (⊓) operation can be used to ensure that permissions are not increased when an object is passed from one thread to another. If a sending thread has permissions p on a resource that it offers with permissions q to another thread, then the actual permissions that are given to the receiver should be the (potentially smaller) permission value $p \sqcap q$. Of course, this is just q in the special case where $q \sqsubseteq p$, which in turn follows vacuously in the further special case where p is *fullPerms*:

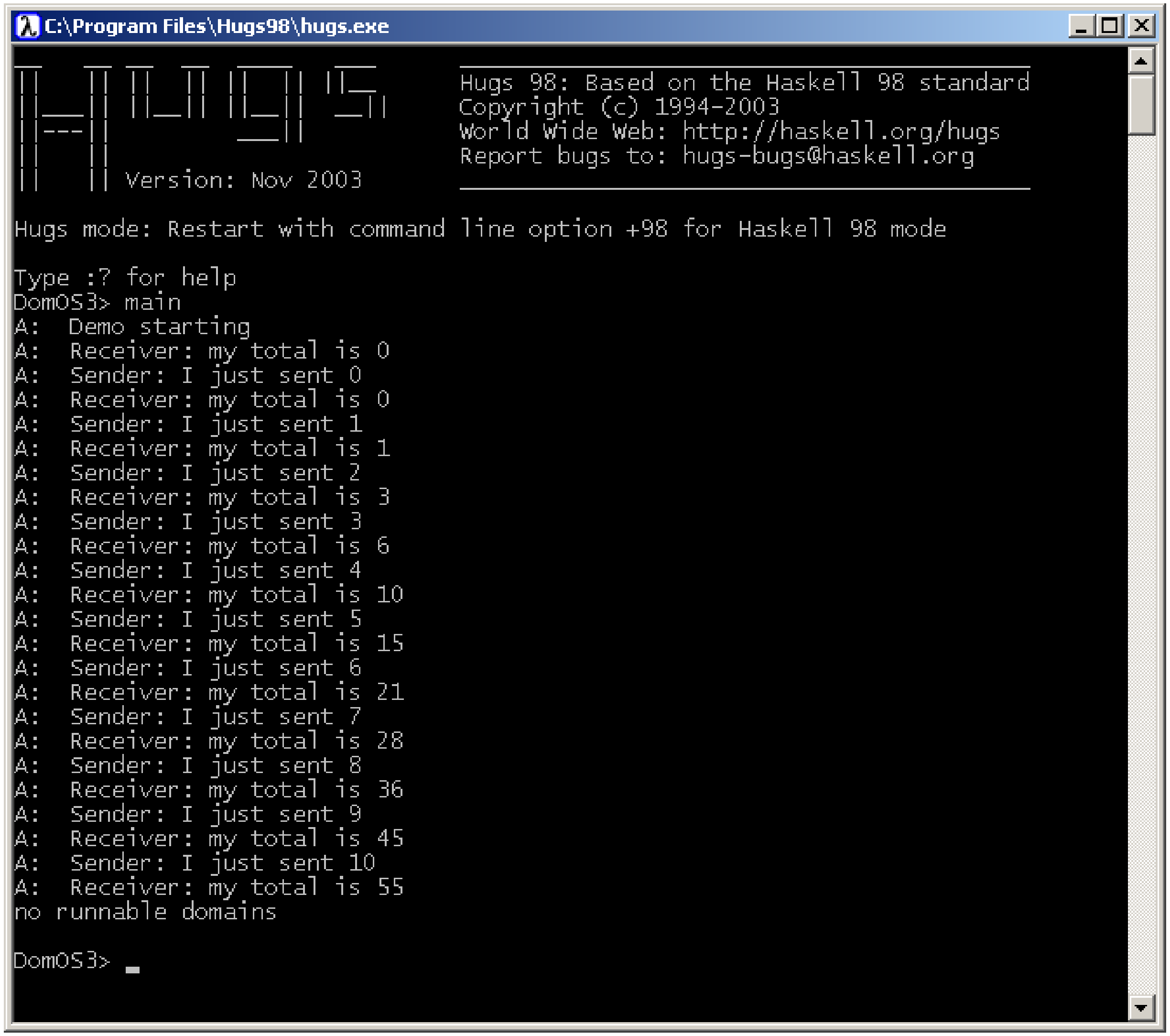
```
assert IntersectPermsLowerBound
  =  ∀p, q . True{ (p ⊓ q) ⊑ p}

assert IntersectPermsOrder
  =  ∀p, q . True{ q ⊑ p} ⟹ ({ p ⊓ q} === q)

assert IntersectPermsFullPerms
  =  ∀q .{ fullPerms ⊓ q} === q
```

Unsurprisingly, each of these properties has a corresponding dual for (⊔):

```
assert UnionPermsUpperBound
  =  ∀p, q . True{ p ⊑ (p ⊔ q)}

assert UnionPermsOrder
  =  ∀p, q . True{ q ⊑ p} ⟹ ({ p ⊔ q} === p)

assert UnionPermsNoPerms
  =  ∀q .{ noPerms ⊔ q} === q
```

5

Executable Model:



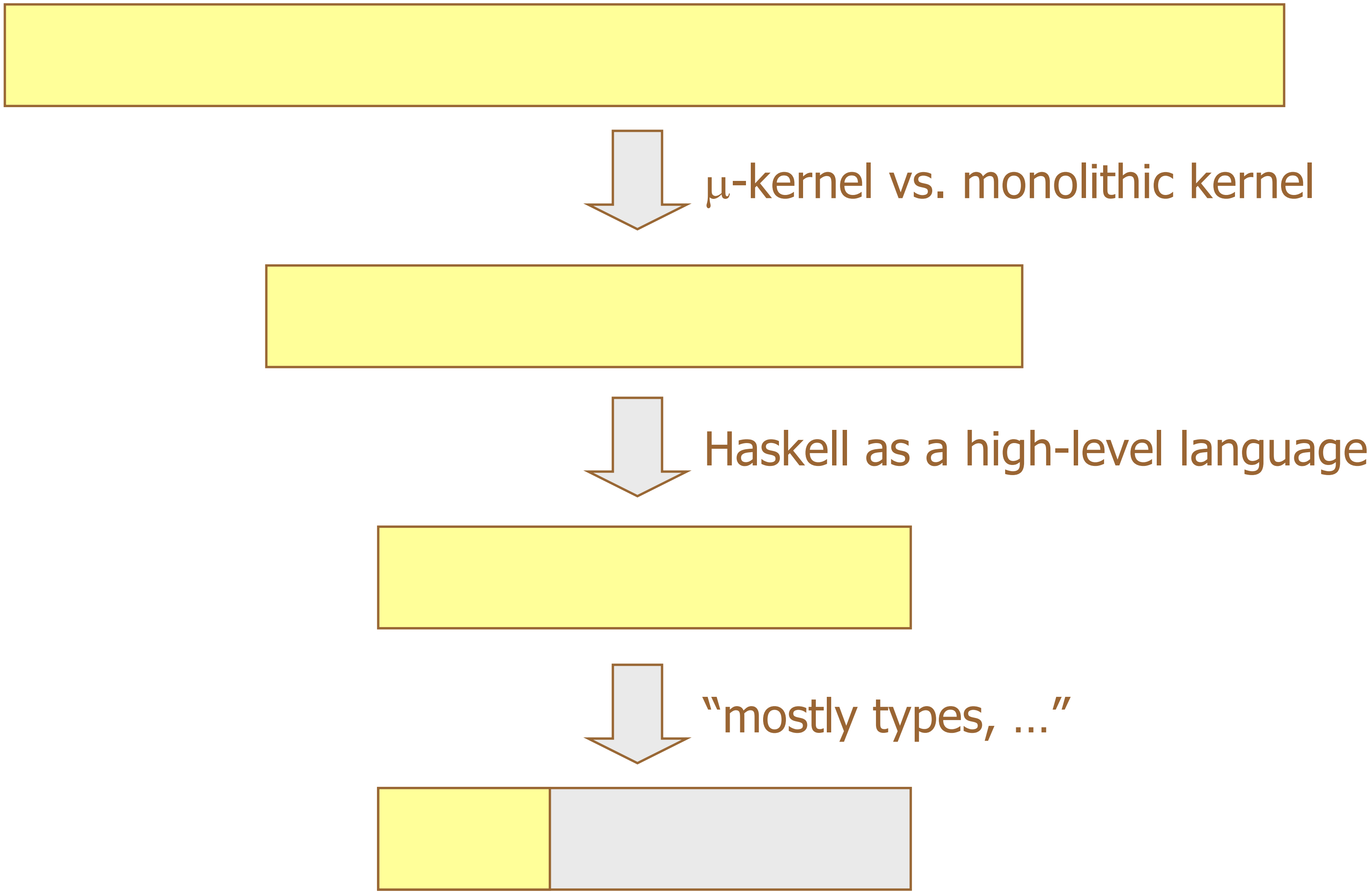
C:\Program Files\Hugs98\hugs.exe

```
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2003
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: Nov 2003       _________________________________________

Hugs mode: Restart with command line option +98 for Haskell 98 mode

Type :? for help
DomOS3> main
A:  Demo starting
A:  Receiver: my total is 0
A:  Sender: I just sent 0
A:  Receiver: my total is 0
A:  Sender: I just sent 1
A:  Receiver: my total is 1
A:  Sender: I just sent 2
A:  Receiver: my total is 3
A:  Sender: I just sent 3
A:  Receiver: my total is 6
A:  Sender: I just sent 4
A:  Receiver: my total is 10
A:  Sender: I just sent 5
A:  Receiver: my total is 15
A:  Sender: I just sent 6
A:  Receiver: my total is 21
A:  Sender: I just sent 7
A:  Receiver: my total is 28
A:  Sender: I just sent 8
A:  Receiver: my total is 36
A:  Sender: I just sent 9
A:  Receiver: my total is 45
A:  Sender: I just sent 10
A:  Receiver: my total is 55
no runnable domains

DomOS3> _
```

Why Haskell?

- **Purity**: the result of a function depends only on the argument value (i.e., no hidden dependencies)
- **Polymorphic types**: powerful and expressive; parametricity provides "theorems for free"
- **Formal semantics**: a foundation for meaningful assurance guarantees
- **Powerful abstract datatypes:** e.g., modular, scalable encapsulation and reasoning about effects using monads

Scalability:



μ-kernel vs. monolithic kernel

Haskell as a high-level language

“mostly types, …”

Note: Diagram not to scale … ☺

Future Prospects:

- Performance is not a primary goal … but it is an issue:
  - Paths through a μ-kernel must be short and fast
  - Runtime system assurance: e.g., garbage collection

- On the table:
  - Mechanisms for efficient construction and manipulation of data structures at the bit-level
  - Small size provides opportunities for aggressive optimization and whole program analysis
  - Default to strict instead of lazy evaluation

L4

What is L4?

- L4 is a “second generation” μ-kernel design

- Original Design: Jochen Liedtke
    - Original goal: To show that μ-kernel based systems are usable in practice with good performance

- Keep it simple:
    - Original API had just 7 system calls dealing with key abstractions:
        - Address spaces: Memory protection
        - Threads: Concurrency
        - IPC: Inter Process Communication

Why Pick L4?

- L4 is industrially and technically **relevant**
  - Multiple working implementations (Pistachio, Fiasco, etc…)
  - Multiple supported architectures (ia32, arm, powerpc, mips, sparc, …)
  - Already used in a variety of domains, including real-time, security, virtual machines & monitors, etc…

Why Pick L4?

- L4 is industrially and technically **relevant**

- L4 is small enough to be **tractable**
  - Original implementation ~ 12K executable
  - Recent/portable/flexible implementations ~ 10-20 KLOC C++

Why Pick L4?

- L4 is industrially and technically **relevant**
- L4 is small enough to be **tractable**
- L4 is real enough to be **interesting**
    - For example, we can run multiple, separated instances of Linux (specifically: L4Linux, Wombat) on top of an L4 μ-kernel

Why Pick L4?

- L4 is industrially and technically **relevant**
- L4 is small enough to be **tractable**
- L4 is real enough to be **interesting**
- L4 is a good **representative** of the target domain and a good tool for exposing core research challenges
  - Threads, address spaces, IPC, preemption, interrupts, etc… are core μ-kernel concepts, regardless of API details
  - It should be possible to retarget to a different API or μ-kernel design

House

An OS in Haskell!?

- OS implementations involve:
  - low-level data structure manipulation, “bit twiddling”
  - asynchronous interrupts, MMU, DMA, IO ports, …

- Haskell may not be your “typical systems programming language” …

- But details like these are within reach …

Page Table Maintenance:

```
type PAddr      -- physical addresses
type VAddr      -- virtual addresses
type PageMap    -- page map references

data PageInfo = PageInfo { pAddr     :: PAddr,
                           writeable:: Bool,
                           dirty     :: Bool,
                           accessed  :: Bool }

setPage :: PageMap → VAddr → Maybe PageInfo → H Bool
getPage :: PageMap → VAddr → H (Maybe PageInfo)

assert   {do setPage pm va pi; getPage pm va}
          ===
         {do setPage pm va pi; return pi}
```

House:

Run a.out executables

Page fault and syscall handlers

Haskell window system and applications

Cooperating concurrent processes

Device drivers (keyboard/mouse/text video/ graphics video/network)

Concurrent threads

Asynchronous exceptions

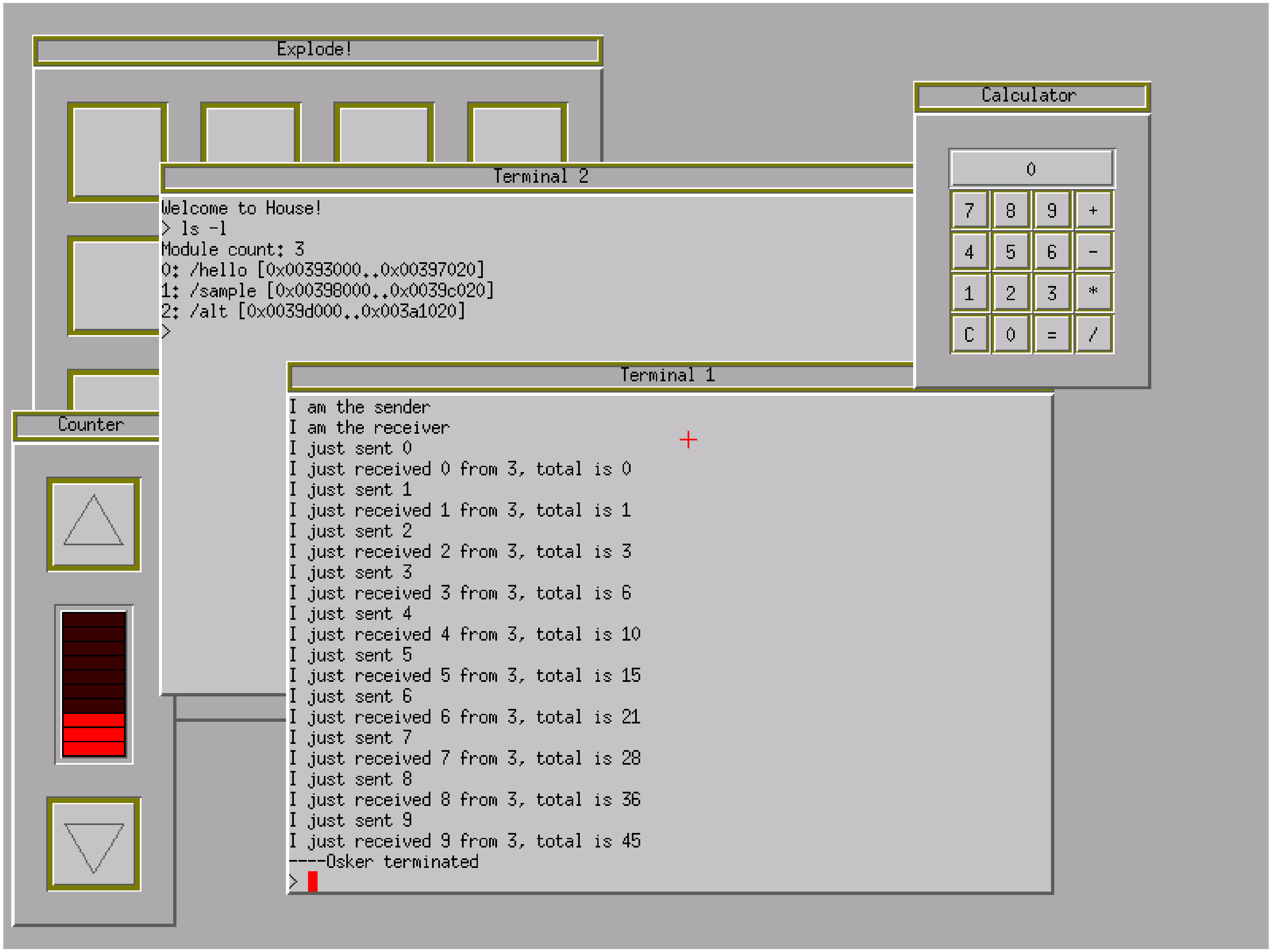
Garbage collection

Address space management

Hardware interrupts/faults

Initial memory configuration

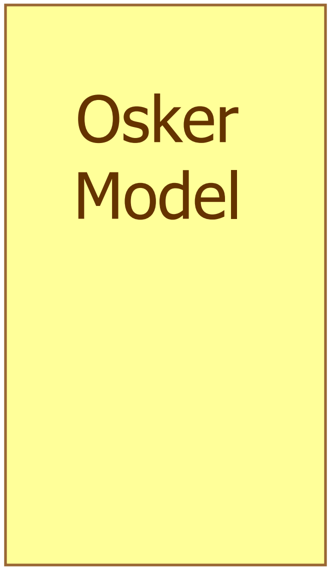
On Bare Metal:



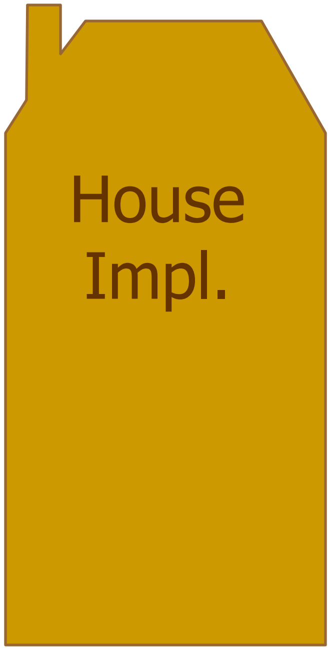
```
Welcome to House!
> ls -l
Module count: 3
0: /hello [0x00393000..0x00397020]
1: /sample [0x00398000..0x0039c020]
2: /alt [0x0039d000..0x003a1020]
>
```

```
I am the sender
I am the receiver
I just sent 0
I just received 0 from 3, total is 0
I just sent 1
I just received 1 from 3, total is 1
I just sent 2
I just received 2 from 3, total is 3
I just sent 3
I just received 3 from 3, total is 6
I just sent 4
I just received 4 from 3, total is 10
I just sent 5
I just received 5 from 3, total is 15
I just sent 6
I just received 6 from 3, total is 21
I just sent 7
I just received 7 from 3, total is 28
I just sent 8
I just received 8 from 3, total is 36
I just sent 9
I just received 9 from 3, total is 45
----Osker terminated
>
```

Relating Osker & House:

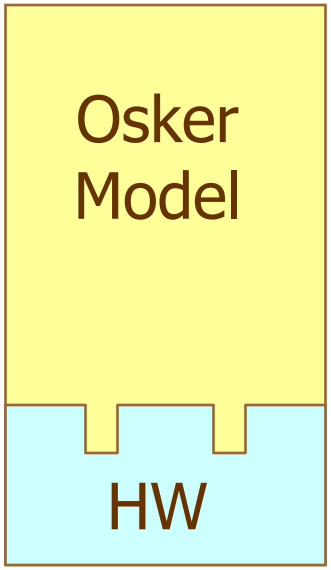


Different Shapes
⇔
Different Interfaces

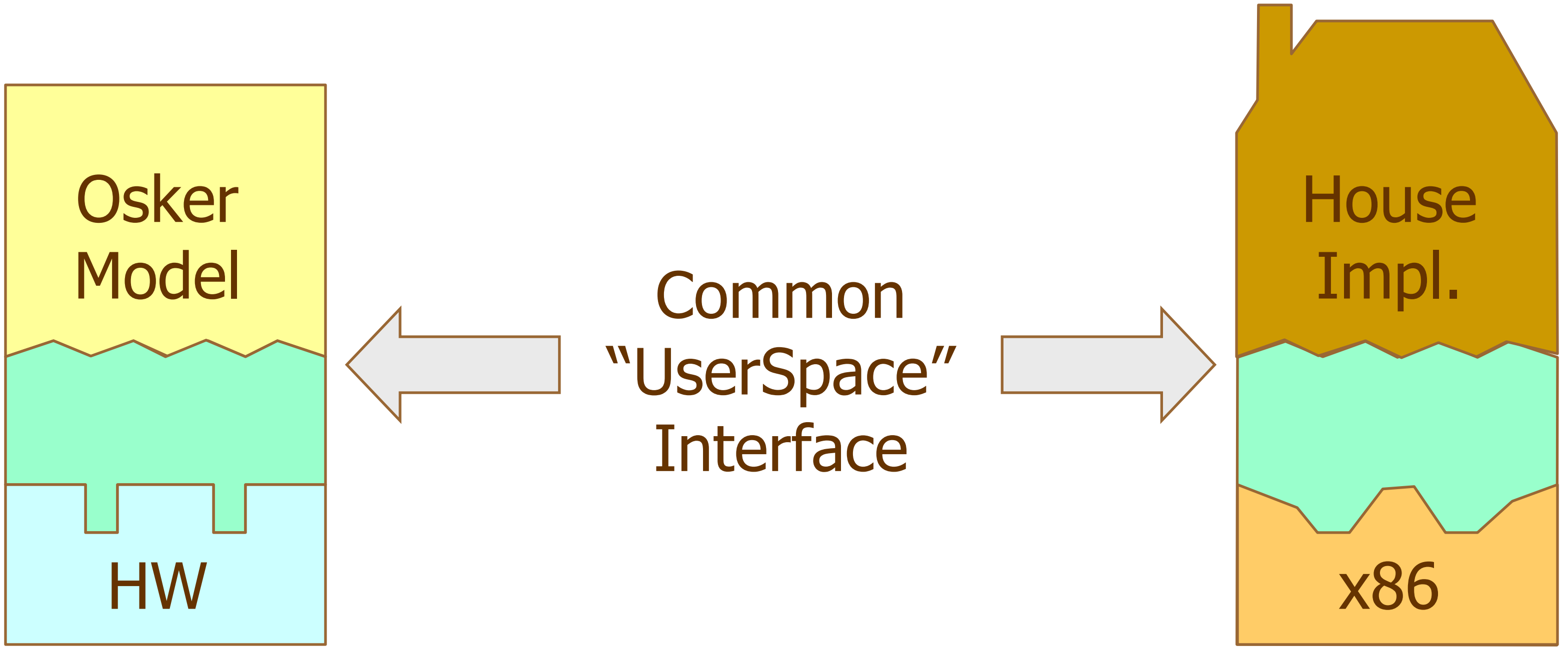


Interface
=
Signature + Properties

Modular Construction:



Modular Construction:



Osker
Model

HW

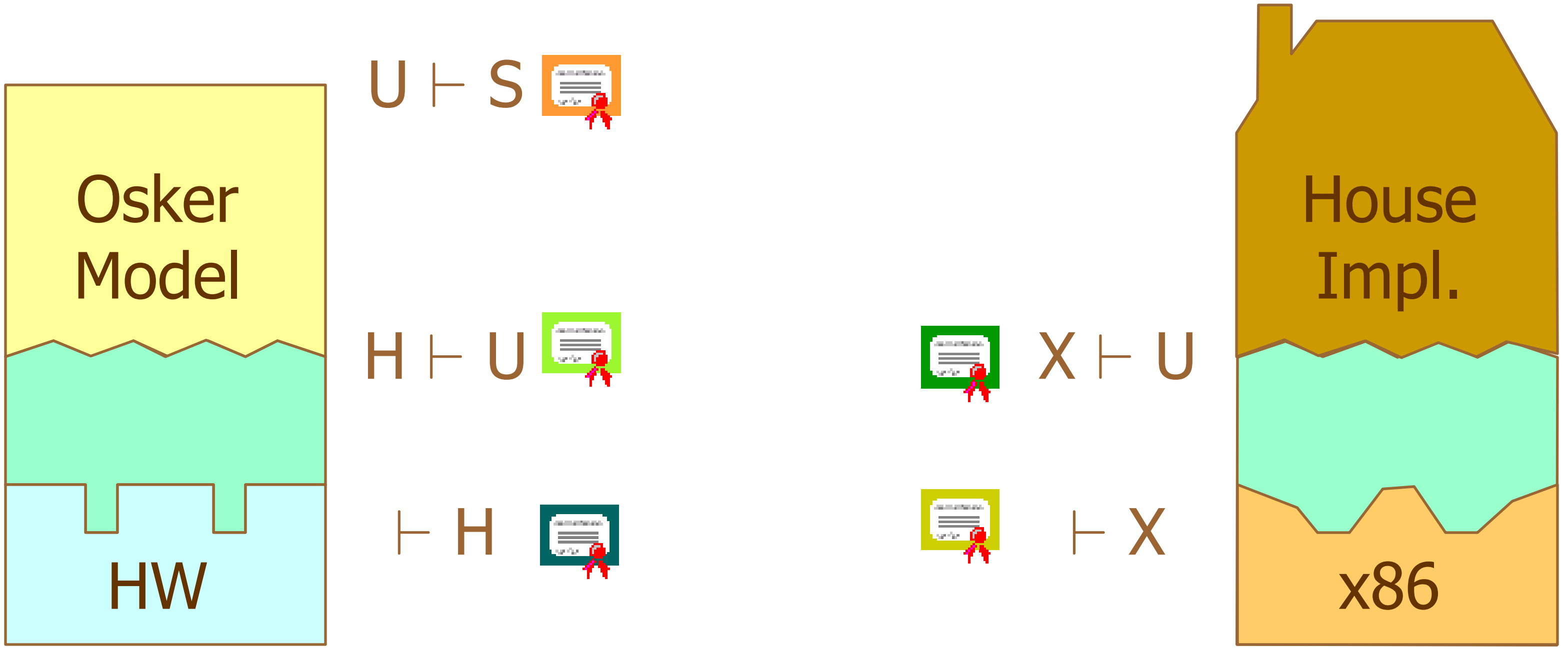
Common
“UserSpace”
Interface

House
Impl.

x86

```
execContext :: PageMap → Context → H (Interrupt, Context)

assert All m, pm, pa, c.
           m ::: NotMapped pm pa
       ==> m ::: Commutes {readPhys pa} {execContext pm c}
```

Modular Certification:



U ⊢ S

Osker

Model

H ⊢ U

X ⊢ U

House

Impl.

⊢ H

⊢ X

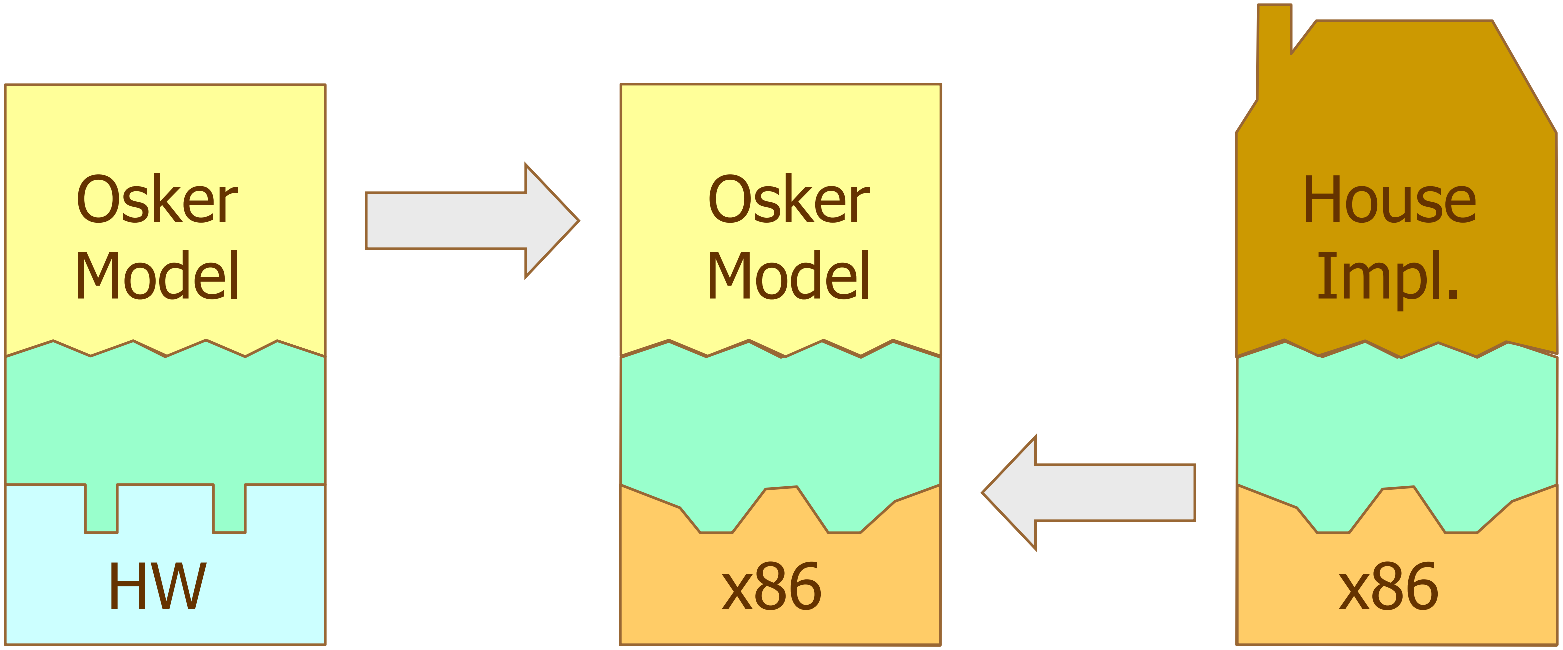
HW

x86

H = Properties of HW model
U = Properties of Userspace interface
S = Osker separation properties
X = Properties of x86 hardware

- Compositional certification
- Consistency checking on U
- Design input on X

Combining Osker & House:



A First Implementation
of Osker on Bare Metal

Standard C code, …

```
#define wait 1
#define sync 2
#define src  3
#define dest 4

extern void lock(), sender(), receiver();

void start() {
   fork(sync, lock);
   fork(src, sender);
   fork(dest,receiver);
   stop();
}

void lock() {
   for (;;) {
     recv(0, wait);
     send(getSender(), wait);
   }
}

#define LOCK(x) send(sync, wait); \
                x; \
```

#defines

Osker
system calls

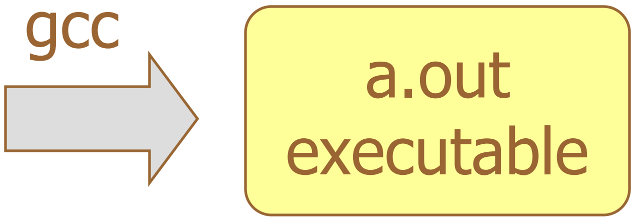
malloc(), protected execution
(divide by zero, segment violation,
time slice exhausted, etc…), …

```
void sender() {
   LOCK(printf("I am the sender\n"));
   int i;
   for (i = 0; i<10; i++) {
     setMsg(i);
     send(dest, wait);
     LOCK(printf("I just sent %d\n", i));
   }
   stop();
}

void receiver() {
   int total = 0;
   LOCK(printf("I am the receiver\n"));
   for (;;) {
      int x;
      int s;
      recv(src, wait);
      x = getMsg();
      s = getSender();
      total += x;
      LOCK(printf(“Received %d from %d,
            total is %d\n", x, s, total));
   }
   stop();
```

printf()

Standard tools, …

```
#define wait 1
#define sync 2
#define src  3
#define dest 4

extern void lock(), sender(), receiver();

void start() {
   fork(sync, lock);
   fork(src, sender);
   fork(dest,receiver);
   stop();
}

void lock() {
   for (;;) {
     recv(0, wait);
     send(getSender(), wait);
   }
}

#define LOCK(x) send(sync, wait); \
                x; \
                recv(sync, wait)

void sender() {
   LOCK(printf("I am the sender\n"));
   int i;
   for (i = 0; i<10; i++) {
     setMsg(i);
     send(dest, wait);
     LOCK(printf("I just sent %d\n",
        i));
   }
   stop();
}

void receiver() {
   int total = 0;
   LOCK(printf("I am the receiver\n"));
   for (;;) {
      int x;
      int s;
      recv(src, wait);
      x = getMsg();
      s = getSender();
      total += x;
      LOCK(printf(“Received %d from %d,
        total is %d\n", x, s, total));
   }
   stop();
}
```



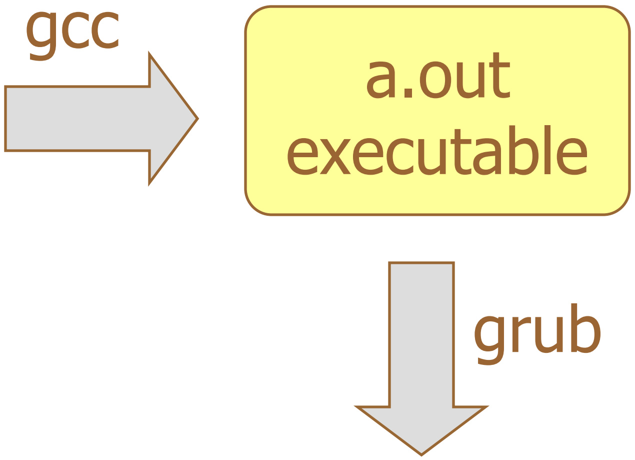
gcc

a.out
executable

Compile, boot, and run:

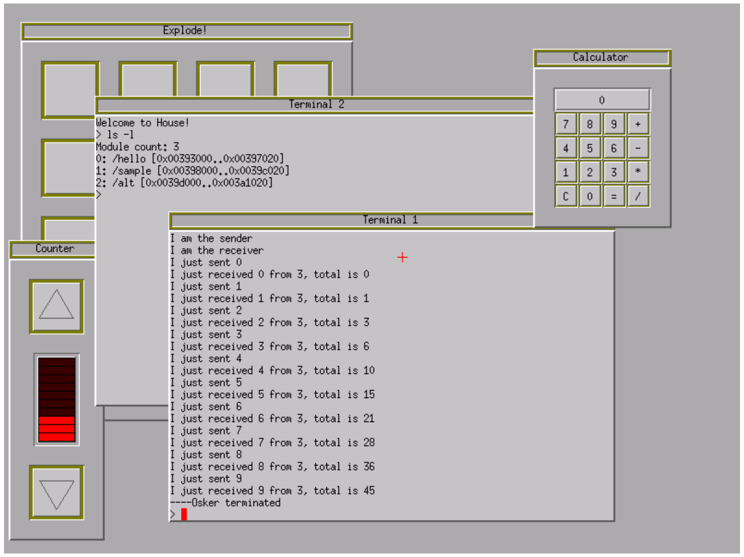
```
#define wait 1
#define sync 2
#define src  3
#define dest 4

extern void lock(), sender(), receiver();

void start() {
   fork(sync, lock);
   fork(src, sender);
   fork(dest,receiver);
   stop();
}

void lock() {
   for (;;) {
     recv(0, wait);
     send(getSender(), wait);
   }
}

#define LOCK(x) send(sync, wait); \
                x; \
                recv(sync, wait)
```

```
void sender() {
   LOCK(printf("I am the sender\n"));
   int i;
   for (i = 0; i<10; i++) {
     setMsg(i);
     send(dest, wait);
     LOCK(printf("I just sent %d\n",
        i));
   }
   stop();
}

void receiver() {
   int total = 0;
   LOCK(printf("I am the receiver\n"));
   for (;;) {
      int x;
      int s;
      recv(src, wait);
      x = getMsg();
      s = getSender();
      total += x;
      LOCK(printf(“Received %d from %d,
        total is %d\n", x, s, total));
   }
   stop();
}
```


gcc

a.out
executable

grub

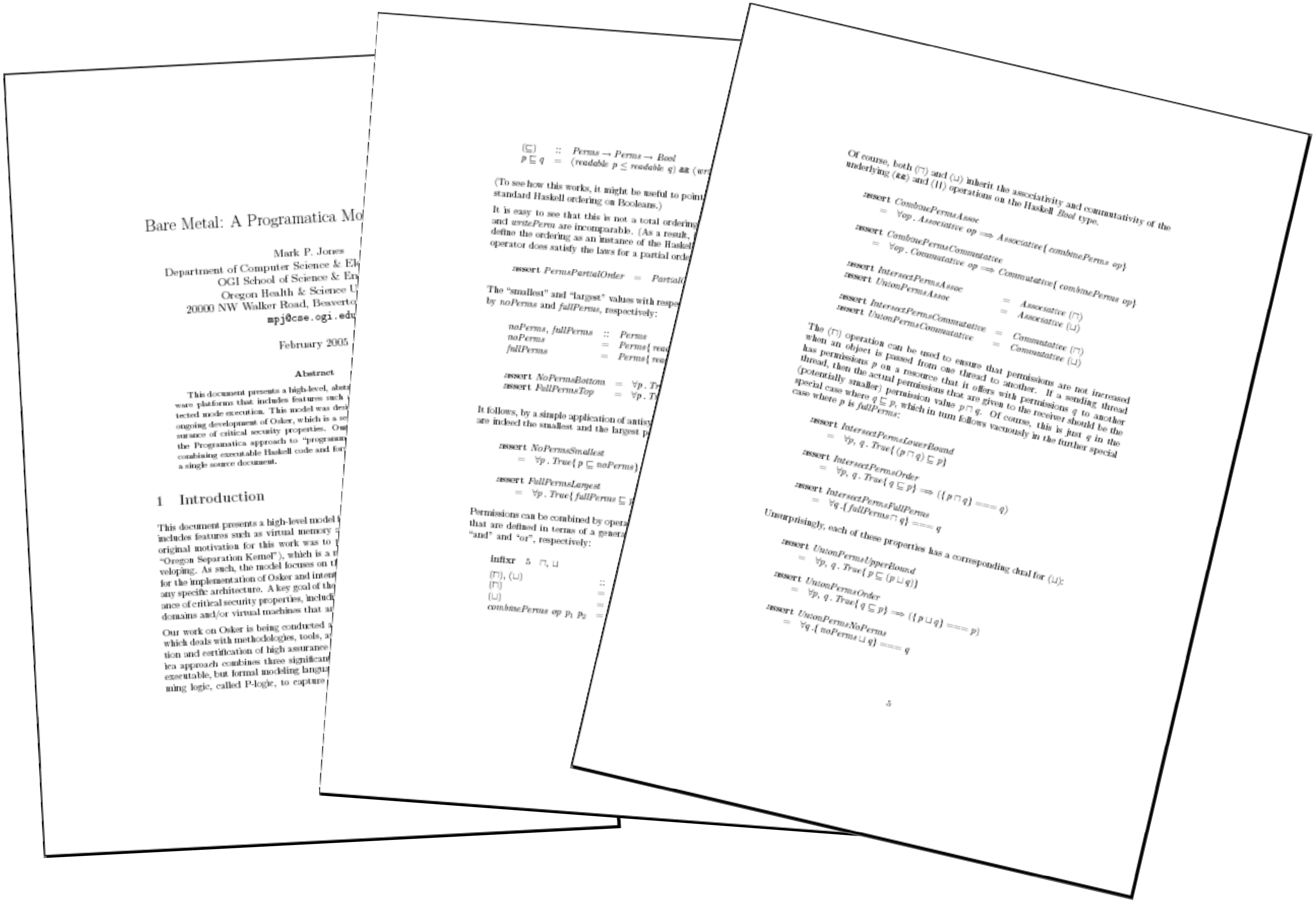


```
Welcome to House!
> ls -l
Module count: 3
0: /hello [0x00393000..0x00397020]
1: /sample [0x00398000..0x0039c020]
2: /alt [0x0039d000..0x003a1020]
>
```

```
I am the sender
I am the receiver
I just sent 0
I just received 0 from 3, total is 0
I just sent 1
I just received 1 from 3, total is 1
I just sent 2
I just received 2 from 3, total is 3
I just sent 3
I just received 3 from 3, total is 6
I just sent 4
I just received 4 from 3, total is 10
I just sent 5
I just received 5 from 3, total is 15
I just sent 6
I just received 6 from 3, total is 21
I just sent 7
I just received 7 from 3, total is 28
I just sent 8
I just received 8 from 3, total is 36
I just sent 9
I just received 9 from 3, total is 45
----Osker terminated
>
```

One Source, Many Uses:

Design Document:



Bare Metal: A Programatica Mo

Mark P. Jones
Department of Computer Science & El
OGI School of Science & En
Oregon Health & Science U
20000 NW Walker Road, Beaverto
mpj@cse.ogi.edu

February 2005

Abstract

This document presents a high-level, abstr
ware platforms that includes features such
tected mode execution. This model was desi
ongoing development of Osker, which is a se
surance of critical security properties. Ou
the Programatica approach to "programm
combining executable Haskell code and for
a single source document.

1 Introduction

This document presents a high-level model
includes features such as virtual memory
original motivation for this work was to
"Oregon Separation Kernel"), which is a
veloping. As such, the model focuses on t
for the implementation of Osker and inten
any specific architecture. A key goal of the
ance of critical security properties, includ
domains and/or virtual machines that a

Our work on Osker is being conducted
which deals with methodologies, tools, a
tion and certification of high assurance
ica approach combines three significan
executable, but formal modeling langu
ming logic, called P-logic, to capture

$(\sqsubseteq) \;::\; \mathit{Perms} \to \mathit{Perms} \to \mathit{Bool}$
$p \sqsubseteq q \;=\; (\mathit{readable}\ p \leq \mathit{readable}\ q)$ && (wr

(To see how this works, it might be useful to point
standard Haskell ordering on Booleans.)

It is easy to see that this is not a total ordering
and *writePerm* are incomparable. (As a result,
define the ordering as an instance of the Haskel
operator does satisfy the laws for a partial orde

assert *PermsPartialOrder* = *Partial*

The "smallest" and "largest" values with respe
by *noPerms* and *fullPerms*, respectively:

$\mathit{noPerms}, \mathit{fullPerms} \;::\; \mathit{Perms}$
$\mathit{noPerms} \;=\; \mathit{Perms}\{$ rea
$\mathit{fullPerms} \;=\; \mathit{Perms}\{$ rea

assert *NoPermsBottom* = $\forall p\,.$ Tr
assert *FullPermsTop* = $\forall p\,.$ T

It follows, by a simple application of antisy
are indeed the smallest and the largest p

assert *NoPermsSmallest*
$= \;\forall p\,.\,\mathit{True}\{\, p \sqsubseteq \mathit{noPerms}\}$

assert *FullPermsLargest*
$= \;\forall p\,.\,\mathit{True}\{\, \mathit{fullPerms} \sqsubseteq$ p

Permissions can be combined by opera
that are defined in terms of a genera
"and" and "or", respectively:

infixr 5 ⊓, ⊔

$(\sqcap), (\sqcup) \;::$
$(\sqcap) \;=$
$(\sqcup) \;=$
$\mathit{combinePerms}\ op\ p_1\ p_2 \;=$

Of course, both $(\sqcap)$ and $(\sqcup)$ inherit the associativity and commutativity of the underlying (&&) and (||) operations on the Haskell *Bool* type.

assert *CombinePermsAssoc*
$= \;\forall op\,.\,\mathit{Associative}\ op \Longrightarrow \mathit{Associative}\{\,\mathit{combinePerms}\ op\}$

assert *CombinePermsCommutative*
$= \;\forall op\,.\,\mathit{Commutative}\ op \Longrightarrow \mathit{Commutative}\{\,\mathit{combinePerms}\ op\}$

assert *IntersectPermsAssoc* $= \;\mathit{Associative}\ (\sqcap)$
assert *UnionPermsAssoc* $= \;\mathit{Associative}\ (\sqcup)$

assert *IntersectPermsCommutative* $= \;\mathit{Commutative}\ (\sqcap)$
assert *UnionPermsCommutative* $= \;\mathit{Commutative}\ (\sqcup)$

The $(\sqcap)$ operation can be used to ensure that permissions are not increased when an object is passed from one thread to another. If a sending thread has permissions p on a resource that it offers with permissions q to another thread, then the actual permissions that are given to the receiver should be the (potentially smaller) permission value $p \sqcap q$. Of course, this is just q in the special case where $q \sqsubseteq p$, which in turn follows vacuously in the further special case where p is *fullPerms*:

assert *IntersectPermsLowerBound*
$= \;\forall p, q\,.\,\mathit{True}\{\,(p \sqcap q) \sqsubseteq p\}$

assert *IntersectPermsOrder*
$= \;\forall p, q\,.\,\mathit{True}\{\, q \sqsubseteq p\} \Longrightarrow (\{\, p \sqcap q\} === q)$

assert *IntersectPermsFullPerms*
$= \;\forall q\,.\{\,\mathit{fullPerms} \sqcap q\} === q$

Unsurprisingly, each of these properties has a corresponding dual for $(\sqcup)$:

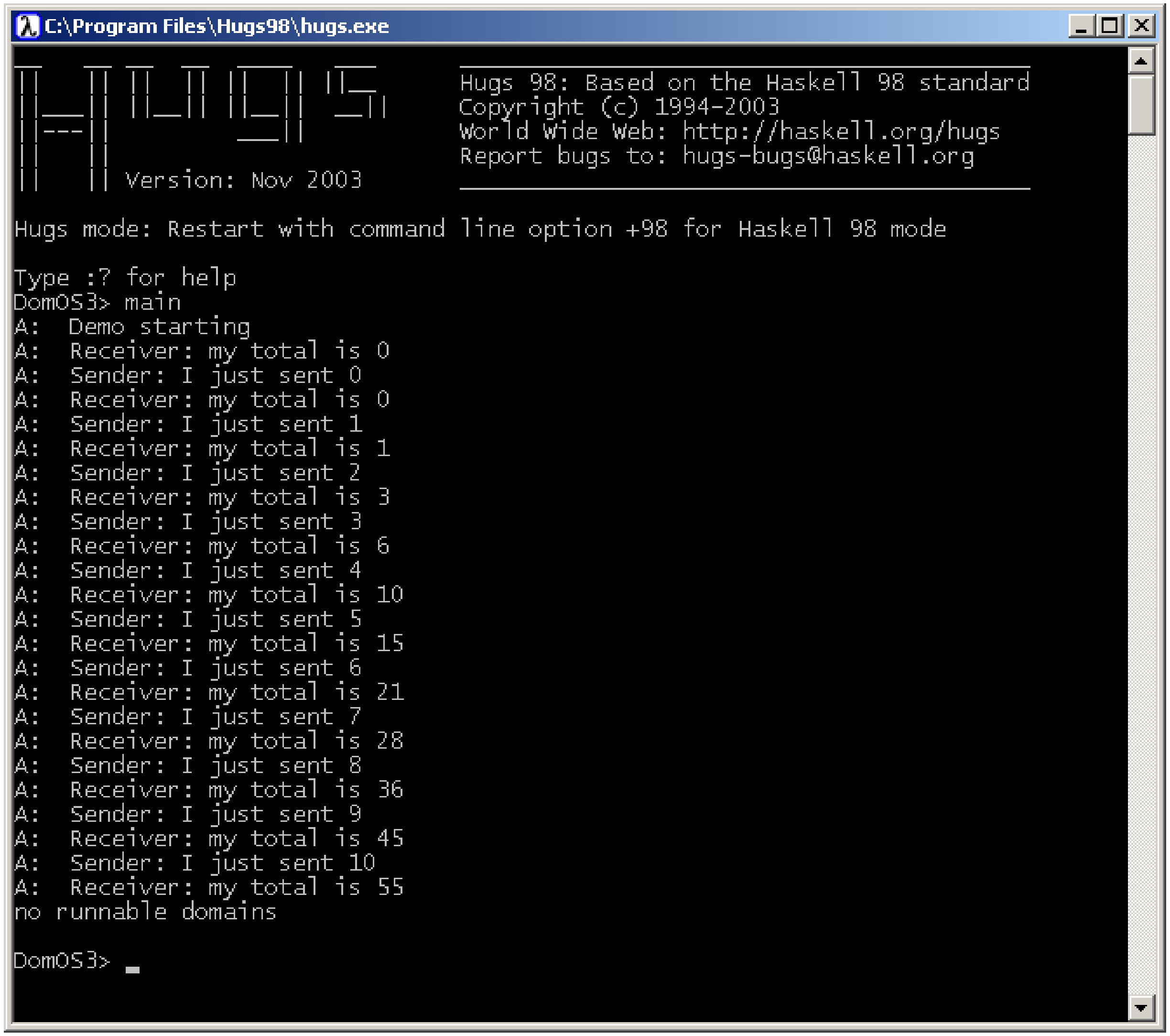
assert *UnionPermsUpperBound*
$= \;\forall p, q\,.\,\mathit{True}\{\, p \sqsubseteq (p \sqcup q)\}$

assert *UnionPermsOrder*
$= \;\forall p, q\,.\,\mathit{True}\{\, q \sqsubseteq p\} \Longrightarrow (\{\, p \sqcup q\} === p)$

assert *UnionPermsNoPerms*
$= \;\forall q\,.\{\,\mathit{noPerms} \sqcup q\} === q$

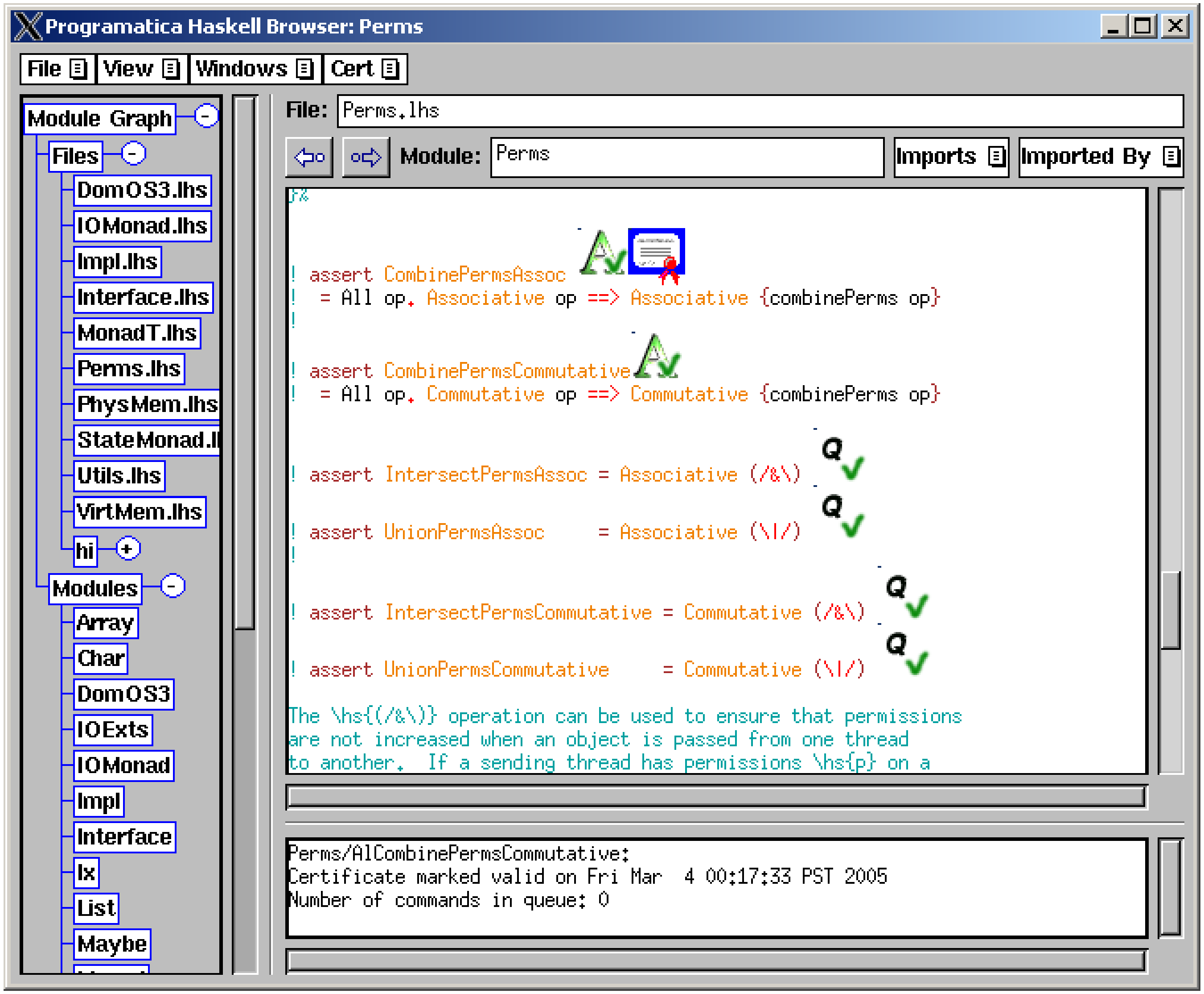
5

Executable Model:



```
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2003
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: Nov 2003       _________________________________________

Hugs mode: Restart with command line option +98 for Haskell 98 mode

Type :? for help
DomOS3> main
A:  Demo starting
A:  Receiver: my total is 0
A:  Sender: I just sent 0
A:  Receiver: my total is 0
A:  Sender: I just sent 1
A:  Receiver: my total is 1
A:  Sender: I just sent 2
A:  Receiver: my total is 3
A:  Sender: I just sent 3
A:  Receiver: my total is 6
A:  Sender: I just sent 4
A:  Receiver: my total is 10
A:  Sender: I just sent 5
A:  Receiver: my total is 15
A:  Sender: I just sent 6
A:  Receiver: my total is 21
A:  Sender: I just sent 7
A:  Receiver: my total is 28
A:  Sender: I just sent 8
A:  Receiver: my total is 36
A:  Sender: I just sent 9
A:  Receiver: my total is 45
A:  Sender: I just sent 10
A:  Receiver: my total is 55
no runnable domains

DomOS3> _
```

Certification Target:



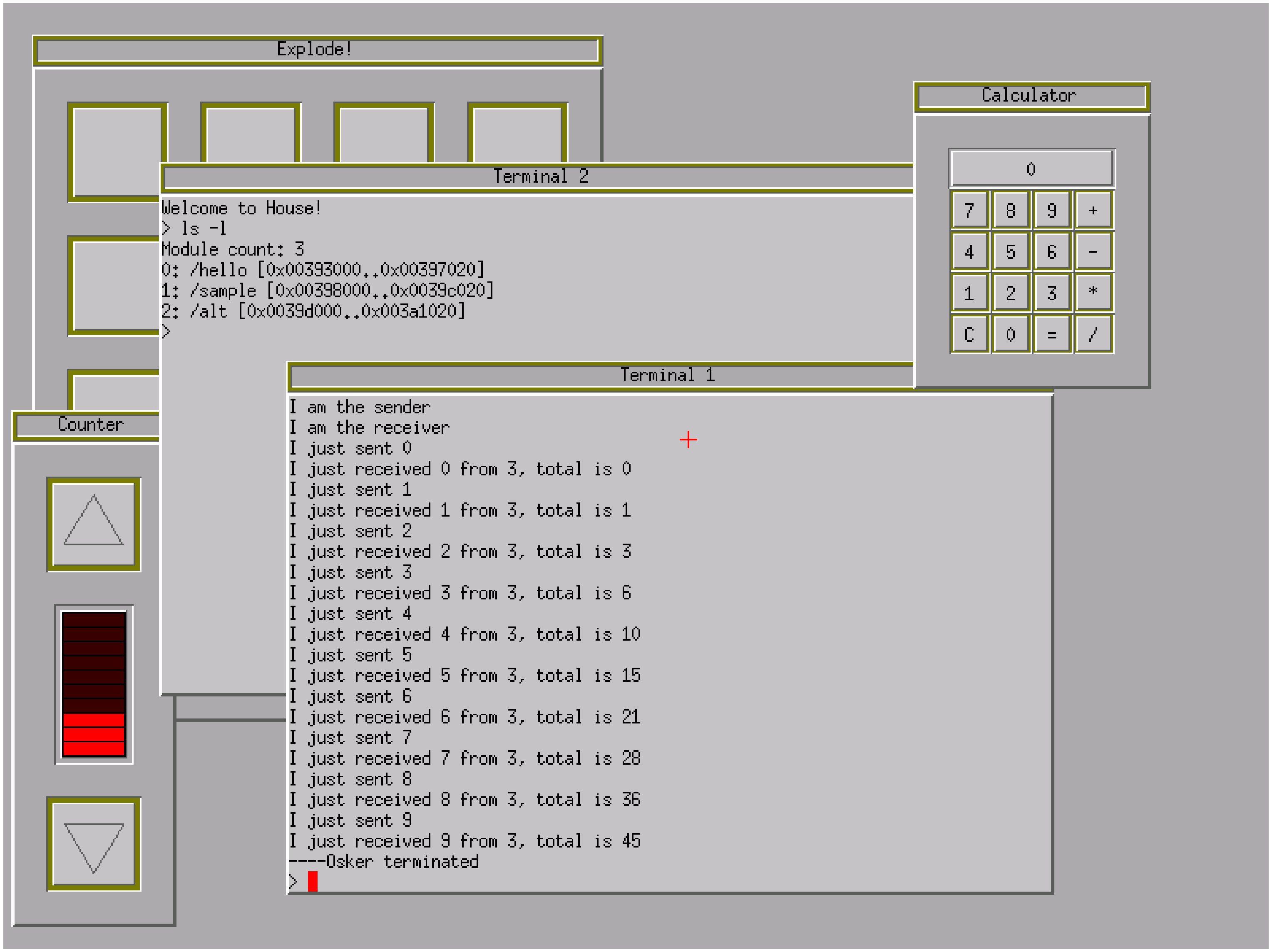
File: Perms.lhs

Module: Perms

```
}%

! assert CombinePermsAssoc
!  = All op. Associative op ==> Associative {combinePerms op}
!

! assert CombinePermsCommutative
!  = All op. Commutative op ==> Commutative {combinePerms op}


! assert IntersectPermsAssoc = Associative (/&\)

! assert UnionPermsAssoc     = Associative (\|/)
!

! assert IntersectPermsCommutative = Commutative (/&\)

! assert UnionPermsCommutative     = Commutative (\|/)

The \hs{(/&\)} operation can be used to ensure that permissions
are not increased when an object is passed from one thread
to another.  If a sending thread has permissions \hs{p} on a
```

```
Perms/AlCombinePermsCommutative:
Certificate marked valid on Fri Mar  4 00:17:33 PST 2005
Number of commands in queue: 0
```

Practical Implementation:



```
Welcome to House!
> ls -l
Module count: 3
0: /hello [0x00393000..0x00397020]
1: /sample [0x00398000..0x0039c020]
2: /alt [0x0039d000..0x003a1020]
>
```

```
I am the sender
I am the receiver
I just sent 0
I just received 0 from 3, total is 0
I just sent 1
I just received 1 from 3, total is 1
I just sent 2
I just received 2 from 3, total is 3
I just sent 3
I just received 3 from 3, total is 6
I just sent 4
I just received 4 from 3, total is 10
I just sent 5
I just received 5 from 3, total is 15
I just sent 6
I just received 6 from 3, total is 21
I just sent 7
I just received 7 from 3, total is 28
I just sent 8
I just received 8 from 3, total is 36
I just sent 9
I just received 9 from 3, total is 45
----Osker terminated
>
```

One Source, Many Uses:

Our Design Document

is also

Our Executable Model

and also

Our Certification Target

and also

Our Running Implementation

Why “House”?

- the “**H**askell **U**sers **O**perating **S**ystem **E**nvironment”

Why “House”?

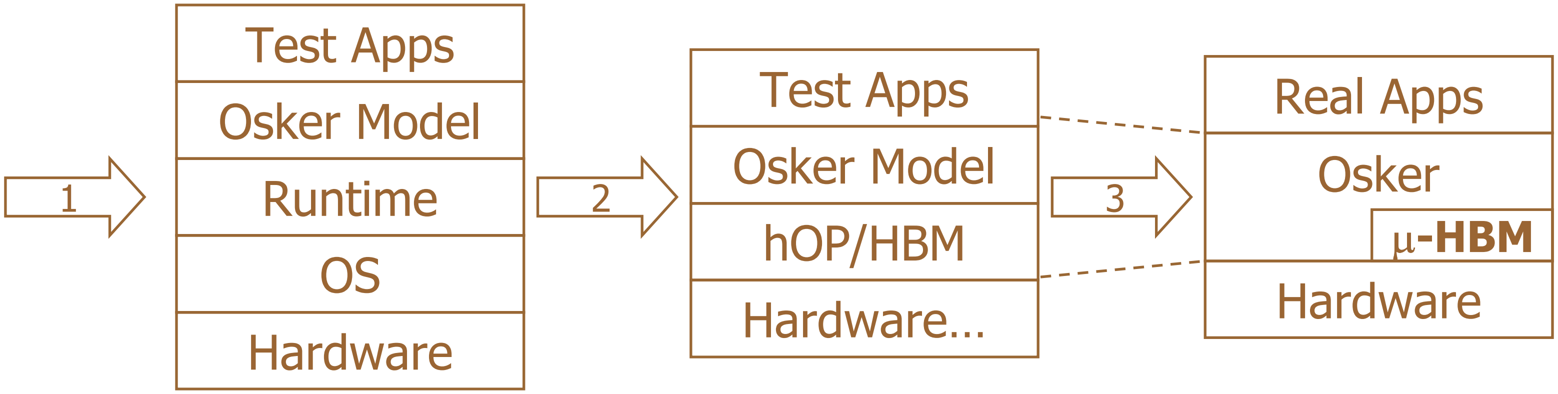
- You are more secure in a House …

- … than if you only have Windows …

Next Steps

Next Steps:

- OS Model: Continuing transition to a more accurate/more complete (and more complex) L4 API
- Hardware Model: Extensions to describe interrupts and hardware concurrency mechanisms
- Establish formal separation property
- Continued evolution of bare metal implementation



Increasing RTS Assurance:

- House illustrates that we can run Haskell programs on a very thin OS layer, obeying a small set of properties
- The runtime system (RTS) is large, complex, and written in C, which makes it hard to build confidence in the overall system
- We need high-confidence versions of two main services:
  - Pre-emptive concurrency (needed for interrupt handling)
  - Garbage collection (possibly real-time)

Possible approaches to high-confidence concurrency:

- Model RTS in Haskell
  - Prove key properties about the model;
  - Transfer results back to C code.
  - (The Galois “Haskell on Bare Metal” project is pursuing this.)

- Remove pre-emptive concurrency from the RTS:
  - Leverage Osker concurrency, handle interrupts explicitly
  - Use a language subset for which we can accurately bound execution times

Possible approaches to high-confidence garbage collection:

- Develop ad-hoc proof of correctness for conventional GC using recently developed separation logics.

- Rewrite the Osker model in a language variant with a region-based type system
  - Should require only simple RTS ⇒ relatively easy to validate