CS 410/510

Languages & Low-Level Programming

Mark P Jones
Portland State University

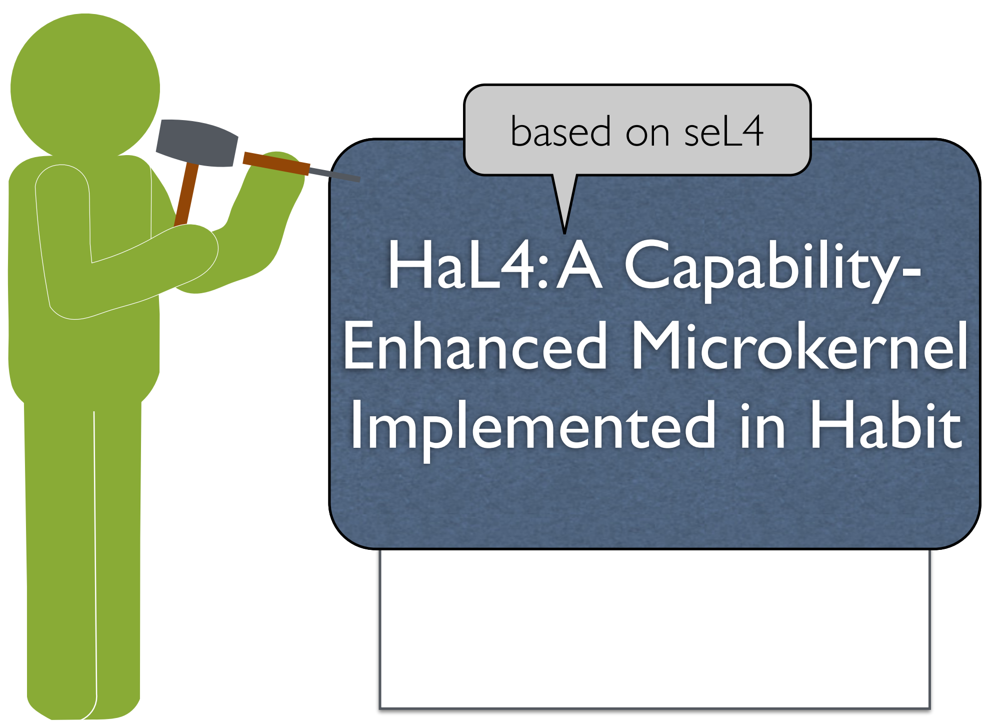
Fall 2018

Week 10: Abstractions and Performance

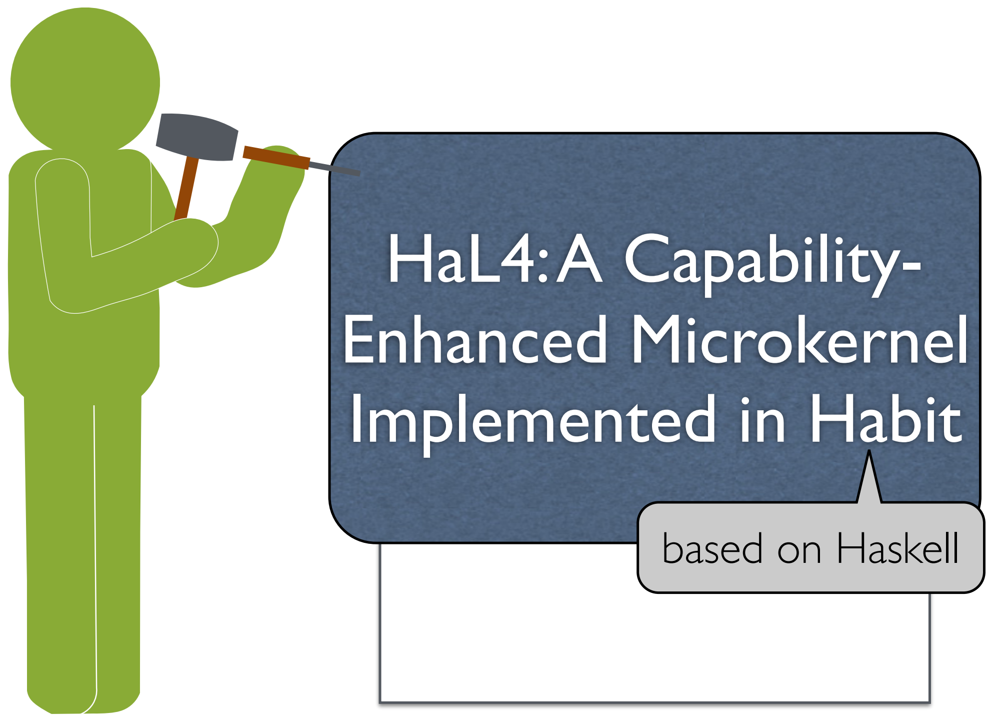
The CEMLaBS Project

- "Using a Capability-Enhanced Microkernel as a Testbed for Language-Based Security"
- Started October 2014, Funded by The National Science Foundation
- Three main questions:
    - **Feasibility**: Is it possible to build an inherently "unsafe" system like seL4 in a "safe" language like Habit?
    - **Benefit**: What benefits might this have, for example, in reducing verification costs?
    - **Performance**: Is it possible to meet reasonable performance goals for this kind of system?

Chipping away ...



Chipping away ...



Opportunities for high-level abstractions?

- Are there good uses for higher-level abstractions in bare metal programming?
  - Algebraic datatypes?
  - First class and higher-order functions?
  - Classes and objects?
  - ...
- And with concerns about performance, can we afford to use them?

Algebraic Datatypes

Sums types and product types

- A **sum type** allows us to capture alternatives:

```
data Bool = False | True    -- Haskell
enum Bool { False, True }   // Rust
```

- A **product type** allows us to package multiple values up as a single, composite value:

```
data Point = MkPoint Int Int        -- Haskell
enum Point { MkPoint(i32, i32) }  // Rust
```

(tuples, arrays, records, structures, etc. are also examples of of product type)

Algebraic datatypes

- **Algebraic datatypes** provide a unified framework for sum and product types as well as arbitrary sums of products:

```
-- Haskell
data Maybe a     = Nothing | Just a
data Either a b = Left a | Right b

// Rust
enum Option<T>     { None, Some(T) }
enum Result<T, E> { Ok(T), Err(E) }
```

- These examples are taken from the standard libraries of the respective languages
- They are also examples of **parameterized types**, allowing reuse over many type parameter combinations

Constructing values of algebraic datatypes

- To make a value of an algebraic datatype, just write the **constructor** followed by an appropriate list of arguments:

In Haskell:

- `Nothing` and `Just 12` are values of type `Maybe Int`
- `Left True` and `Right "hello"` are values of type `Either Bool String`

In Rust:

- `None` and `Some(12)` are values of type `Option<i32>`
- `Ok(true)` and `Err("hello")` are values of type `Result<bool, String>`

Using values of algebraic datatypes

- We use **pattern matching** constructs to inspect and extract data from values of algebraic datatypes:

In Haskell, assuming `val` has type `Maybe String`:

```
case val of
     Nothing   -> "I don't know your name"
     Just name -> "hello " ++ name
```

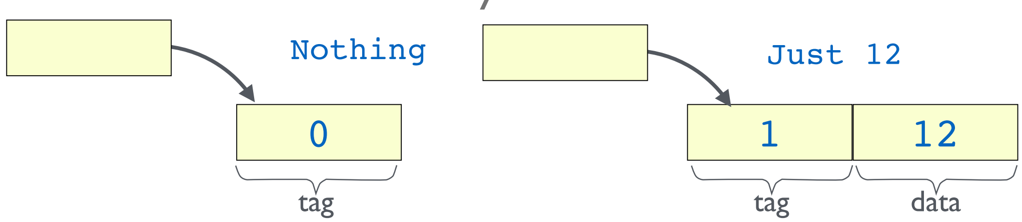
In Rust, assuming `val` has type `Option<String>`:

```
match val {
     None => "I don't know your name"
     Some(name) => "hello " + name
}
```

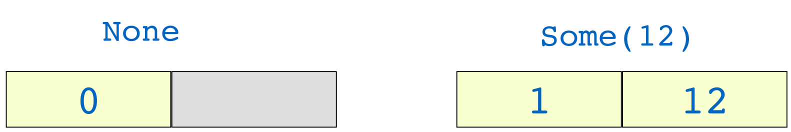

Representing values of algebraic datatypes

- Language definitions typically do not specify exactly how values of algebraic datatypes are represented
- Two common approaches:

Boxed representations: Every value is described by a pointer to a block of memory:



Union representations: Every value is described by a block of memory big enough to store any value of that type:



Algebraic datatypes + recursion

- Algebraic datatypes become even more powerful when combined with **recursion**:

```
-- Haskell
data List a = Nil | Cons a (List a)

// Rust
enum List<A> { Nil, Cons(Box<(A, List<A>)>)}
```

- (`Box<T>` is the Rust type for boxed values of type `T`)
- Example: `Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))` is a value of type `List Int` (might also be written `[1, 2, 3, 4]`)
- Unsurprisingly, we can define recursive functions to work with recursive types like these ...

Algebraic datatypes using classes

- We can simulate algebraic datatypes with OO classes:

```
abstract class List<A> {
   Cons isCons() { return null; }
}
class Nil<A> extends List<A> { }
class Cons<A> extends List<A> {
   A       head;
   List<A> tail;
   Cons(A head, List<A> tail) {
       this.head = head;
       this.tail = tail;
   }
   Cons isCons() { return this; }
}
```

- More verbose, but also more extensible
- Combines/tangles type and code definitions in classes

Habit's bitdata types

- The Habit programming language provides special syntax for defining bitdata types:

```
bitdata Perms = Perms [ r, w, x :: Bool]

bitdata Fpage = Fpage [ base :: Bit 22 | size :: Bit 6
                      | reserved :: Bit 1 | perms :: Perms ]
```

- A crucial feature of definitions like these is the ability to specify bit-level representations/layout
- In other respects, bitdata types are like algebraic datatypes:
  - Construct and update values without use of `<<`, `&`, `|`, etc.
  - Pattern match to deconstruct values

Example: IA32 Paging Structures

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			**CR3**
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	**PDE: 4MB page**
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**
Ignored																															0	**PDE: not present**
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**
Ignored																															0	**PTE: not present**

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

Example: IA32 Paging Structures

Here is how we describe page directory entries in Habit:

```
bitdata PDE /WordSize                               -- Page Directory Entries

 = UnmappedPDE  [ unused=0        :: Bit 31 | B0 ]  -- Unused entry (present bit reset)

 | PageTablePDE [ ptab            :: Phys PageTable -- physical address of page table
                | unused=0        :: Bit 4
                | B0                                -- signals PageTablePDE
                | attrs=readWrite :: PagingAttrs    -- paging attributes
                | B1 ]                              -- present bit set

 | SuperPagePDE [ super           :: Phys SuperPage -- physical address of superpage
                | unused=0        :: Bit 13
                | global=0        :: Bit 1          -- 1 => global translation (if cr4.pge=1)
                | B1                                -- signals SuperPagePDE
                | attrs           :: PagingAttrs    -- paging attributes
                | B1 ]                              -- present bit set

bitdata PagingAttrs /6
 = PagingAttrs [ dirty    = 0         :: Bit 1   -- Dirty; 1 => data written to page
               | accessed = 0         :: Bit 1   -- Accessed; 1 => page accessed
               | caching  = Caching[] :: Caching
               | us                   :: Bit 1   -- User/supervisor; 1 => user access allowed
               | rw                   :: Bit 1 ] -- Read/write; 1 => write access allowed
```


Example: IA32 Paging Structures

And here is how we might write functions that use these definitions to implement useful operations on paging structures:

```
mapPage pdir virt phys
  = case<- readRef (pdir @ virt.dir) of
        UnmappedPDE    -> ... add page table and map page ...
        SuperPagePDE[] -> ... superpage already mapped ...
        PageTablePDE[ptab] ->
          case<- readRef (fromPhys ptab @ virt.tab) of
             MappedPTE[] -> ... page already mapped ...
             UnmappedPTE -> ... map the page ...
```

There are no messy bit-level operations to worry about here: all of that is handled automatically by bitdata mechanisms ...

First-class Functions
and
Higher-order Functions

First-class functions

- A lot of modern programming languages provide mechanisms for writing down anonymous functions / lambda expressions:

<table>
<tr><td>Haskell</td><td><code>\x -> x + 1</code></td></tr>
<tr><td>LISP</td><td><code>(lambda (x)</code>
<code>(+ x 1))</code></td></tr>
<tr><td>Python</td><td><code>lambda x: x + 1</code></td></tr>
</table>

<table>
<tr><td>Javascript</td><td><code>function (x) x + 1</code></td></tr>
<tr><td>C++ 11</td><td><code>[] (int x) -> int</code>
<code>{ return x + 1; }</code></td></tr>
<tr><td>Rust</td><td><code>|x| (x + 1)</code></td></tr>
</table>

- These expressions construct functions as **first class** values:
  - they can be passed as arguments to other functions
  - returned as results
  - stored in data structures

Simple examples

- The identity function:

```
id = \x -> x
```

id has a **polymorphic** type: It can be treated as a function of type `t -> t` for any type `t`

- The "successor" function

```
succ = \x -> x + 1
```

succ has type `Int -> Int`

- The "add" function

```
add = \x -> (\y -> x + y)
```

add has type `Int -> Int -> Int`

- The "compose" function

```
compose = \f -> \g -> \x -> f (g x)
```

compose has type `(b -> c) -> (a -> b) -> (a -> c)`

Higher-order functions

- **Higher-order functions** are functions that take other functions as inputs or return functions as outputs
- `compose` and `map` are classic examples of higher-order functions

```
map = \f xs ->
         case xs of
          Nil        -> Nil
          Cons y ys  -> Cons (f y) (map f ys)
```

- For example:

```
map (\x -> x + 1) [1,2,3,4]  == [2,3,4,5]
map (\x -> 2 * x) [1,2,3,4]  == [2,4,6,8]
```

- Good for capturing recurring patterns as reusable functions

First-class functions using classes

- We can use OO classes to represent first-class functions:

```
abstract class Func<A, B> {
   abstract B applyTo(A arg);
}

class Id<A> extends Func<A, A> {
    A applyTo(A arg) { return arg; }
}

class Succ extends Func<int, int> {
   int applyTo(int arg) { return arg + 1; }
}
```

- Objects that represent first-class functions are called **closures**
- Some language descriptions even use the term "closure" instead of "first-class function"

First-class functions using classes, continued

- We can build closures for functions with multiple arguments:

```
class Add1 extends Func<int, int> {
   private int n;
   new Add1(int n) { this.n = n; }
   int applyTo(int arg) { return arg + n; }
}

class Add extends Func<int, int> {
   Func<int, int> applyTo(int arg) { return new Add1(n); }
}
```

- Sample use:

```
new Add().applyTo(1).applyTo(2)   ===>   returns 3
```

- A single class can have many methods, which might require multiple functions
- But the verbose notation can discourage users ...

Functions vs procedures

- In many languages, the terms "function" and "procedure" are used almost interchangeably
- In Habit, they are different!
- A **function** is a value of type `a -> b` for some *input* type `a` and *output* type `b`

  For any given input value, a function always produces the same output value

- A **procedure** is a value of type `Proc a` for some *result* type `a`

  Every time it is executed, a procedure can have a side effect and produce a result of type `a` (both which could be different every time ...)

Combining functions and procedures

- We can use these together to describe procedures with arguments
- Compare:

A_1 -> A_2 -> ... -> R

a pure function, no side effects

A_1 -> A_2 -> ... -> Proc R

a parameterized procedure, may have side effects

- A typical C prototype for a function like this:

R f(A_1 arg_1, A_2 arg_2, ...)

no guarantees, could do almost anything!

Why is this useful?

1. We can distinguish between procedures that can have side effects and pure functions that do not

   Useful documentation; simplifies reasoning; enables optimizations

2. We can generalize to support multiple procedure types:

   `Proc a` for regular procedures

   `Init a` for procedures that can only run during kernel initialization

   Now we can enforce restrictions on the use of functions that should only be called during initialization (e.g., `allocPage()` in the capabilities lab) via compile-time type checking

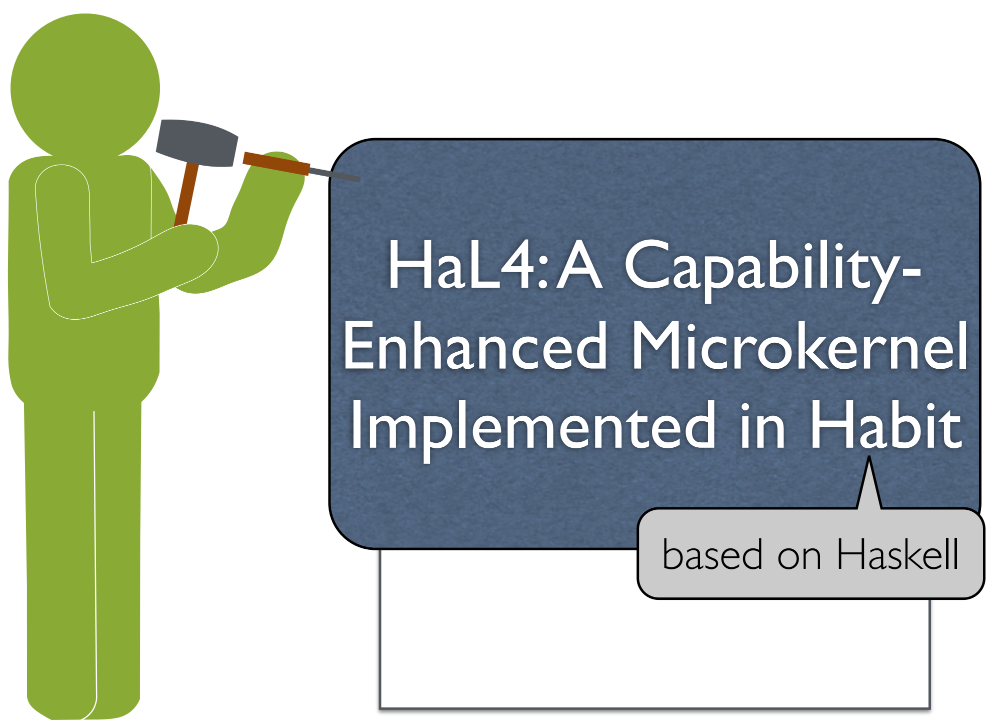
> Talk to me for further details; this is related to"Monads" in functional programming

Opportunities for high-level abstractions?

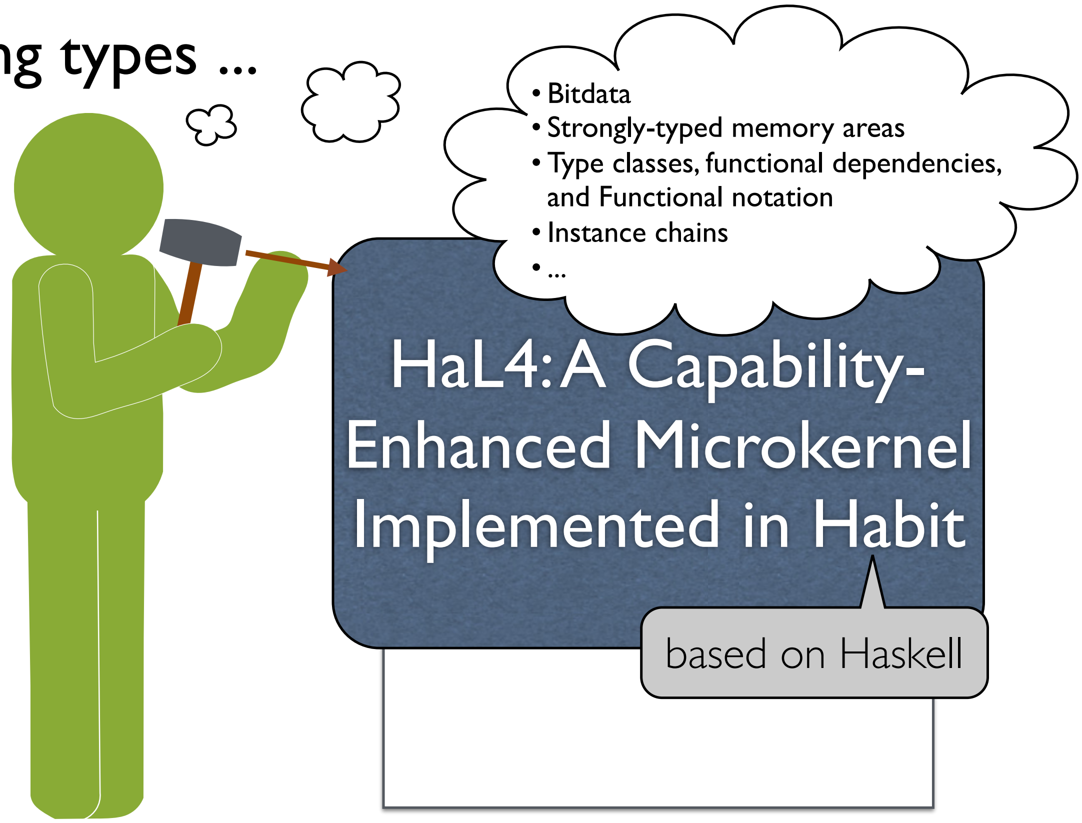
- Are there good uses for higher-level abstractions in bare metal programming?
  - Algebraic datatypes?
  - First class and higher-order functions?
  - Classes and objects?
  - ...
- And with concerns about performance, can we afford to use them?

A small case study:
The Multiboot Information Structure

Chipping away ...



Using types ...

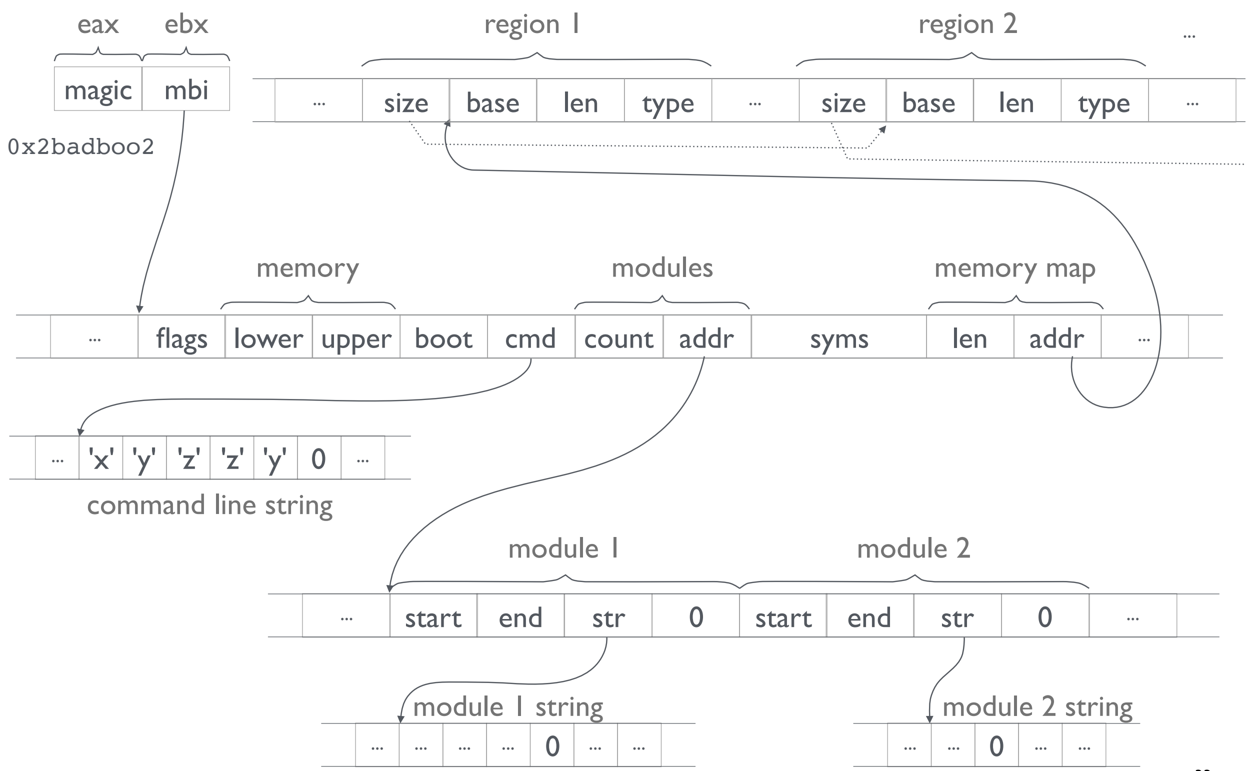


Booting a PC via GRUB

When you turn on a PC:

- The CPU initializes itself and performs a self test, before jumping to a known address in the BIOS ROM
- The BIOS searches for a "bootable device" and loads a 446 byte program into memory from its first sector (the MBR)
- The MBR code uses BIOS functions to load a full featured boot loader (GRUB) in to memory
- GRUB searches the disk for a configuration file and interprets the commands there to load a full featured OS in to memory
- The OS configures itself using information passed in from GRUB via a "Multiboot Information Structure"

The Multiboot Information Structure



The Multiboot Information Structure, in C

```
extern struct MultibootInfo* mbi;
extern unsigned              mbi_magic;
#define MBI_MAGIC 0x2BADB002

struct MultibootInfo {
  unsigned                 flags;
# define MBI_MEM_VALID     (1 << 0)
# define MBI_CMD_VALID     (1 << 2)
# define MBI_MODS_VALID    (1 << 3)
# define MBI_MMAP_VALID    (1 << 6)

  unsigned                 memLower;
  unsigned                 memUpper;
  unsigned                 bootDevice;
  char*                    cmdline;
  unsigned                 modsCount;
  struct MultibootModule*  modsAddr;
  unsigned                 syms[4];
  unsigned                 mmapLength;
  unsigned                 mmapAddr;
};
```

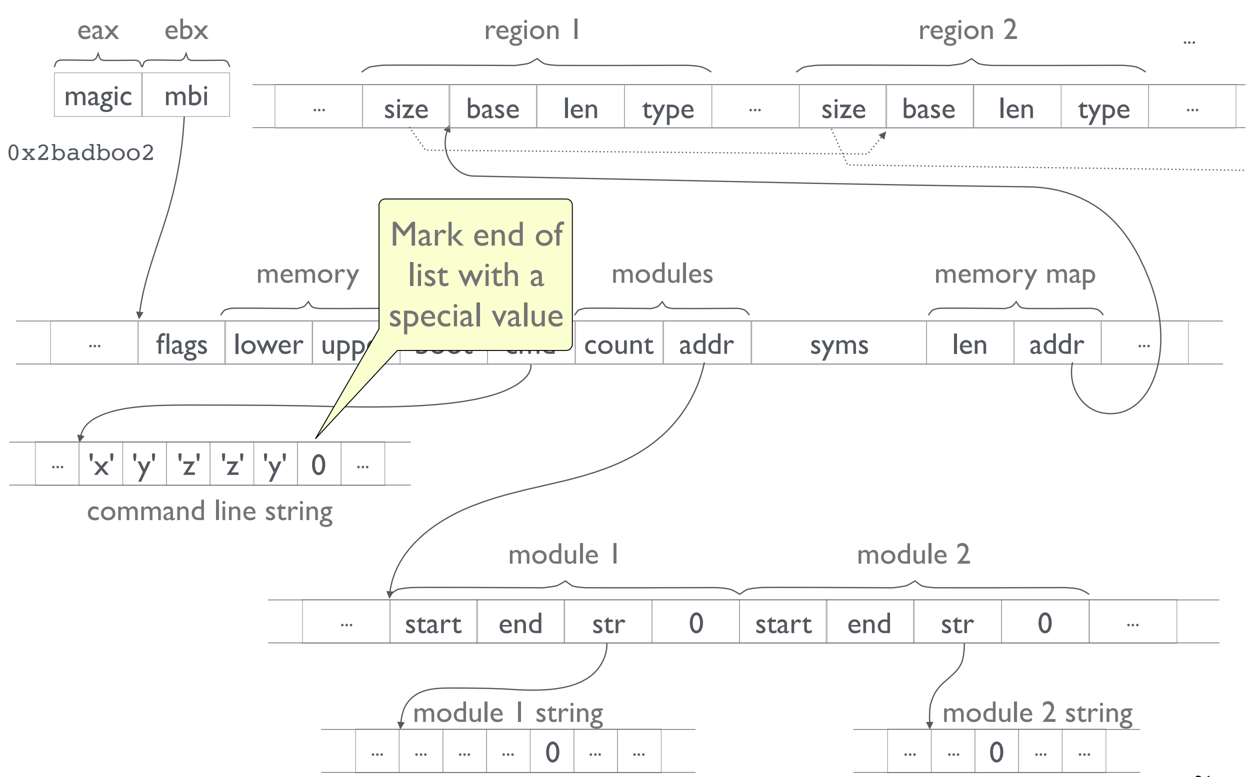
```
struct MultibootModule {
  unsigned modStart;
  unsigned modEnd;
  char*    modString;
  unsigned reserved;
};

struct MultibootMMap {
  unsigned size;
  unsigned baseLo;
  unsigned baseHi;
  unsigned lenLo;
  unsigned lenHi;
  unsigned type;
};
```

Intentionally or otherwise, the multiboot designers used multiple techniques to represent variable-length components

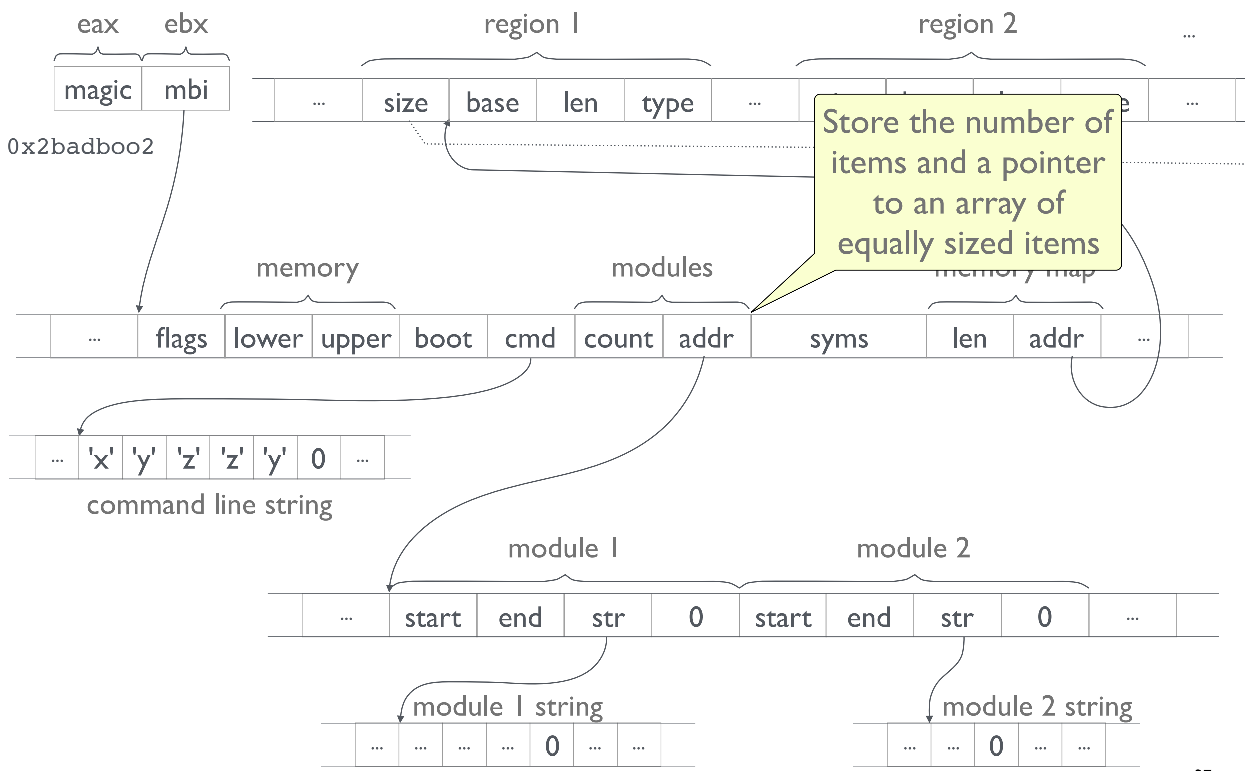
Representing variable length components

- Intentionally or otherwise, the multiboot designers used multiple techniques to represent variable-length components:
  - Mark end of list with a special value, no need to store the length explicitly
  - Store the number of items and a pointer to the first (0^{th}) entry in an array of equally sized items
  - Store the size (in bytes) of the array with a pointer to (some known position in) the first item; access later items by an offset (or pointer) to allow for varying item sizes
- Many other variations are possible (e.g., store address or offset of last byte; pack pointer + size in single word; ...)

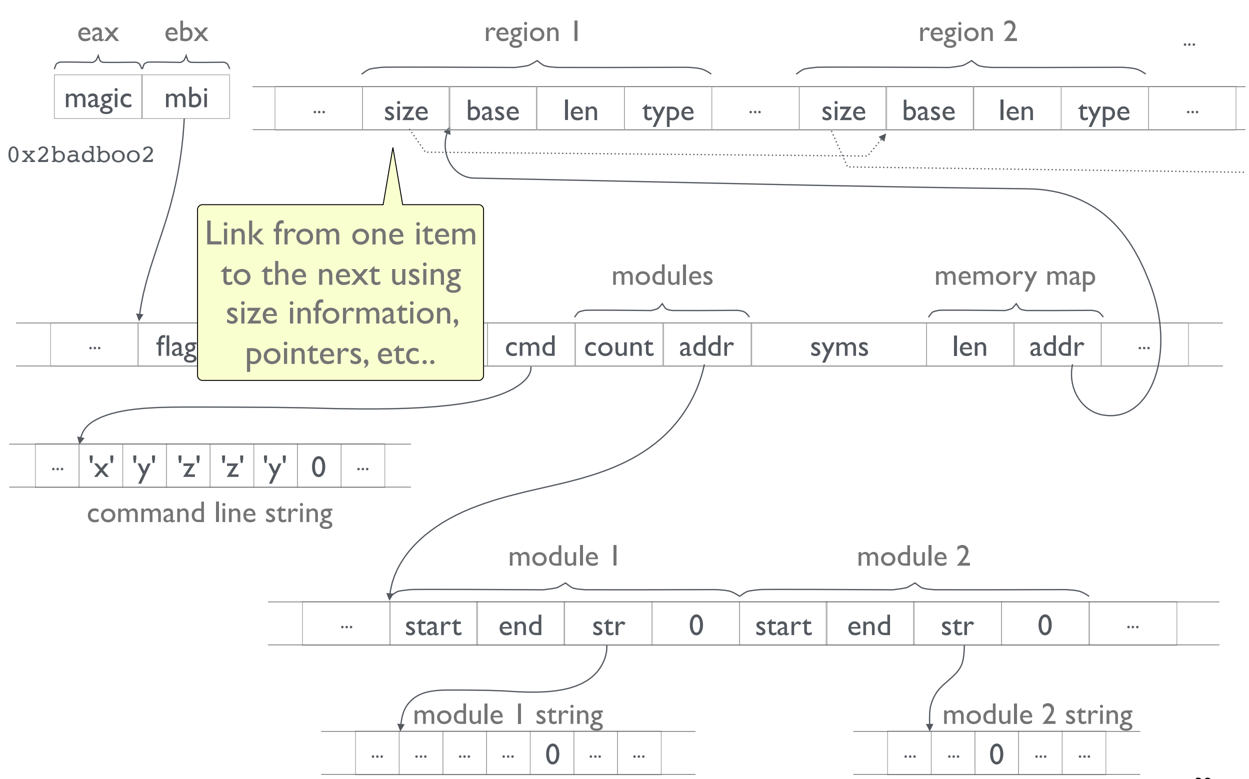
The Multiboot Information Structure



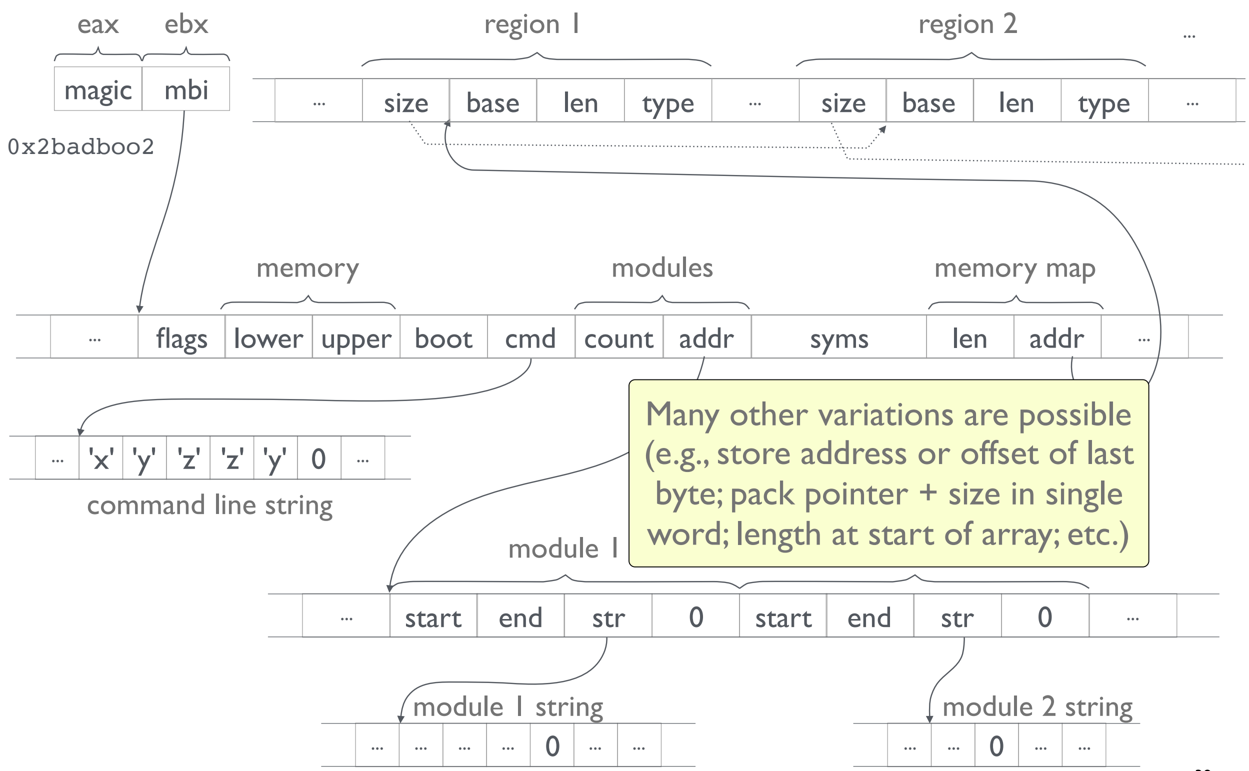
The Multiboot Information Structure



The Multiboot Information Structure



The Multiboot Information Structure



Programming challenges

- What could go wrong if we're writing C programs to work with a Multiboot Information Structure?
  - How do we enforce checking for the magic number?
  - How do we identify/access individual flag bits?
  - How do we find the start of a variable length component?
  - How do we move to the next component?
  - How do we determine when we have reached the end?
  - How do we prevent access to adjacent regions of memory that are not part of the Multiboot Information Structure?
- Current practices to avoid/minimize errors: Disciplined programming; Code reviews; Extensive testing; Limit revisions.
- Do modern language designs have anything to offer here?

Abstract types

- Instead of exposing the underlying pointer type, with full (and unsafe) pointer arithmetic, we could use an *abstract type*
  - Key idea: separate specifications from implementations
- **Specification**: We can work with null-terminated strings by introducing a type `AsciiZ` with a single operation:

```
next :: AsciiZ -> Proc (Maybe (Char, AsciiZ))
```

- **Implementation**: An `AsciiZ` value is a (non-null) pointer to a null-terminated string of characters
  - `next s` returns `Just (c, s1)` if `s` points to character `c` and the remainder of the string is `s1`
  - Otherwise `next s` returns `Nothing`

Notes

- The `next` operation encapsulates checking for null, reading a character, and incrementing the pointer *in a single operation*
- In general, an abstract type's design should:
  - ensure safety (leverage types)
  - avoid redundant computation (e.g., repeated tests)
  - allow for an efficient implementation ...
- Don't underestimate the challenges of figuring out a good design!

Cursors

- This approach generalizes quite easily to handle other components of the MultiBoot Information Structure as well as other table and tree structures in low-level code

```
next :: Cursor -> Proc (Maybe (Val, Cursor))
```

- For example, we could traverse an array using a `Cursor` that encapsulates two components:
  - The number of remaining elements
  - A pointer to the current element

A sample consumer of `AsciiZ` strings

- Using some notation from Habit:

```
putStr   :: AsciiZ -> Proc ()
putStr s = case<- next s of
             Nothing      -> return ()
             Just (c, s1) -> do putChar c
                                putStr s1
```

- A simple implementation of `next` would construct a value of the form `Just (c,s1)` for *every* character in the string
  - ⇒ Significant heap allocation, performance will suffer
  - ⇒ Garbage collection; predictability will be compromised
  - ⇒ Heavyweight approach: a single pointer is all you need ...
- It might be hard to get good performance out of this ...

A sample consumer of AsciiZ strings

- Using some notation from Habit:

```
putStr   :: AsciiZ -> Proc ()
putStr s = case<- next s of
                 Nothing      -> return ()
                 Just (c, s1) -> do putChar c
                                    putStr s1
```

- putStr immediately consumes values produced by next

A sample consumer of AsciiZ strings

- Using some notation from Habit:

```
putStr   :: AsciiZ -> Proc ()
putStr s = case<- next s of
                 Nothing      -> return ()
                 Just (c, s1) -> do putChar c
                                    putStr s1
```

- putStr immediately consumes values produced by next

  a whole program optimizer should be able to fuse the code for the two functions to eliminate the overhead ...

The compiled version of `putStr`

```
putStr <- k54{}

k54{} t564 = k53{t564}

k53{t563} [] = b97[t563]

b97[t560] =
  t561 <- readChar((t560))
  t562 <- nullChar((t561))
  if t562
    then b96[]
    else b102[t560, t561]
```

```
b102[t555, t556] =
  t557 <- incAsciiz((t555))
  [] <- putChar((t556))
  t558 <- readChar((t557))
  t559 <- nullChar((t558))
  if t559
    then b96[]
    else b102[t557, t558]

b96[] = return Unit

Unit <- Unit()
```

Key details:

- No allocation in the main `putStr` loop (i.e., in block `b102`)!
- Simple pointers

Another example: CursorSum in Habit

Add a collection of items accessed via a cursor:

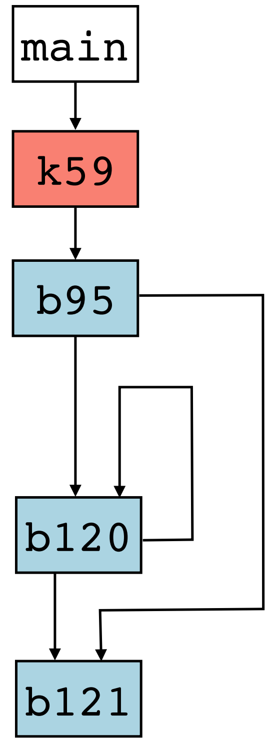
```
main :: Proc Word
main  = do c <- getCursor
           foldCursor accum c 0

accum    :: ItemRef -> Word -> Proc Word
accum i a = fmap (add a) (itemData i)

foldCursor :: (ItemRef -> a -> Proc a) -> Cursor -> a -> Proc a
foldCursor f c a
  = case next c of
      Nothing      -> return a
      Just (i, nc) -> f i a >>= foldCursor f nc
```

Things to note: higher-order functions, pattern matching, monads, polymorphic types, etc...

Things to ignore: everything else!

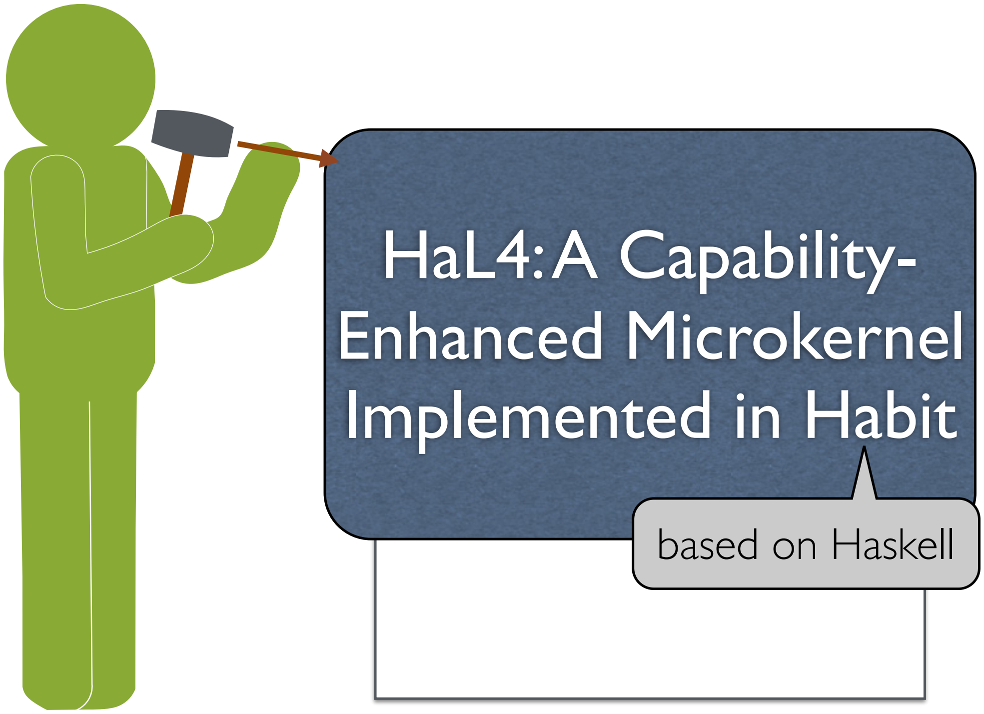
Another example: CursorSum in Habit

```
main <- k59{}

k59{} [] = b95[]

b95[] =
  t618 <- getCursor(())
  t619 <- Cursor 0 t618
  t620 <- Cursor 1 t618
  t621 <- primGte((t620, 0))
  if t621
    then b120[t619, t620, 0]
    else b121[]

b120[t610, t611, t612] =
  t613 <- add((t611, -1))
  t614 <- incItemRef((t610))
  t615 <- itemData((t610))
  t616 <- add((t612, t615))
  t617 <- primGte((t613, 0))
  if t617
    then b120[t614, t613, t616]
    else b121[]

b121[] = return 0
```

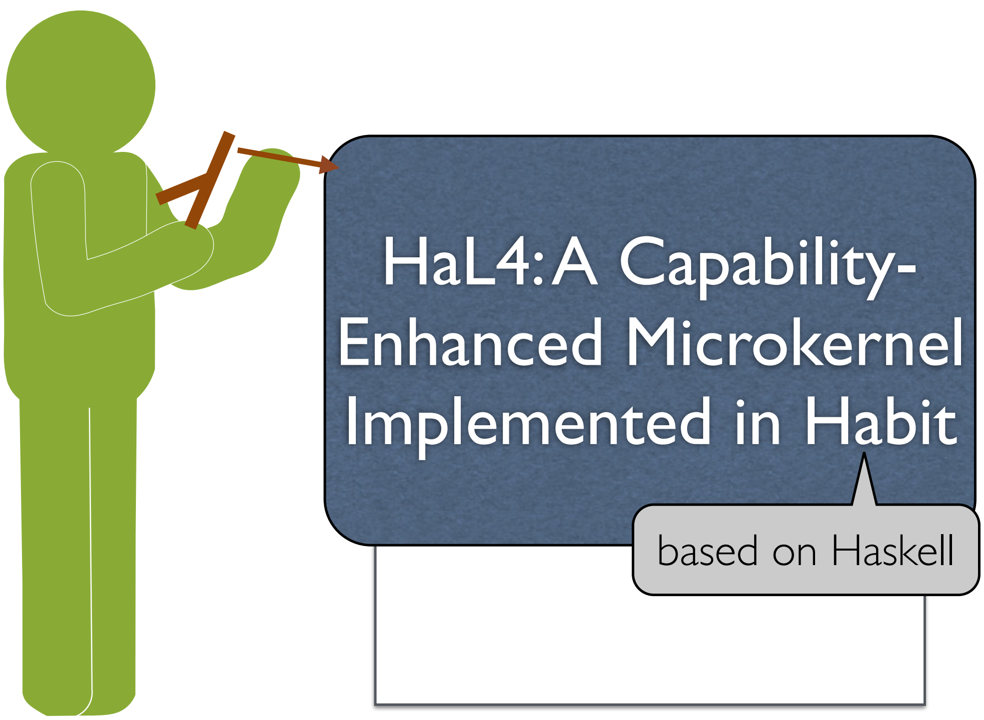


Another Case Study: System Call Validators

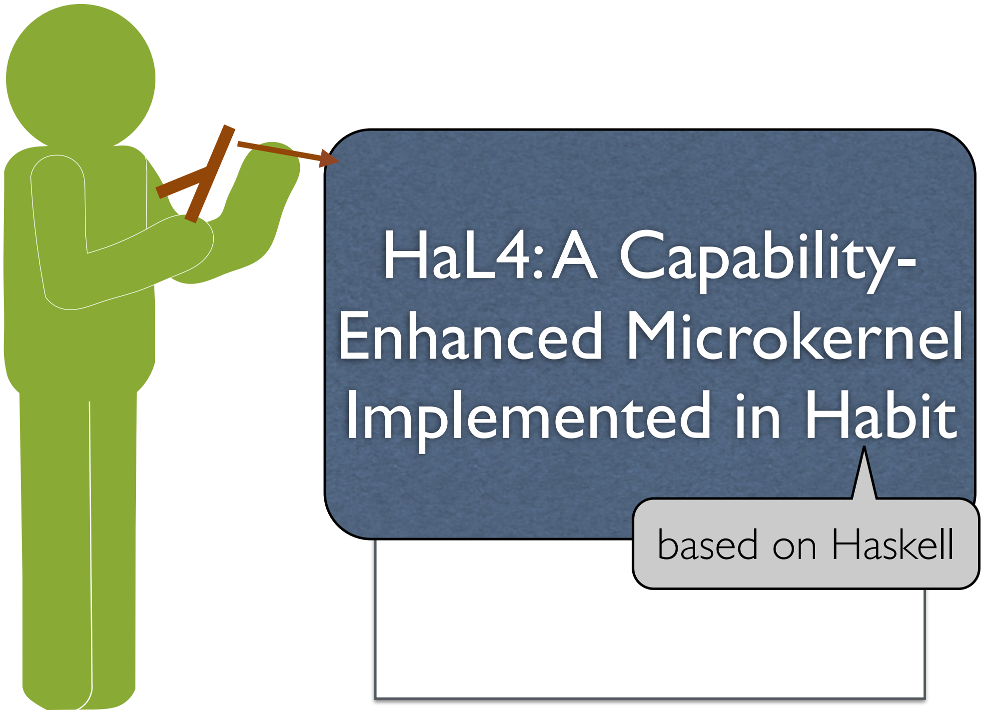
Using types ...



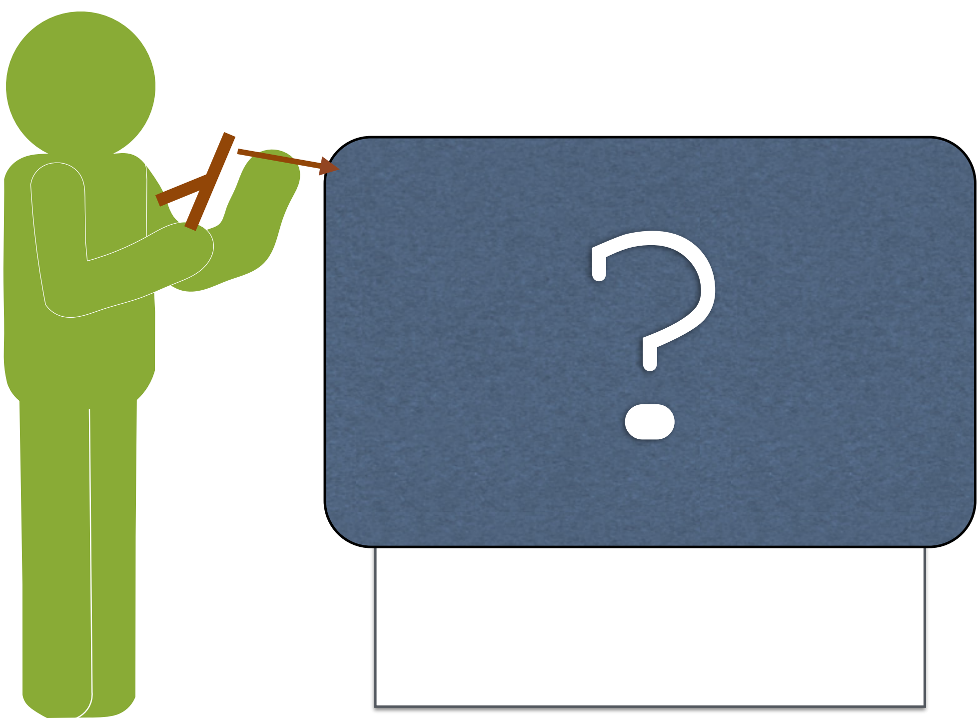
Using lambda ...



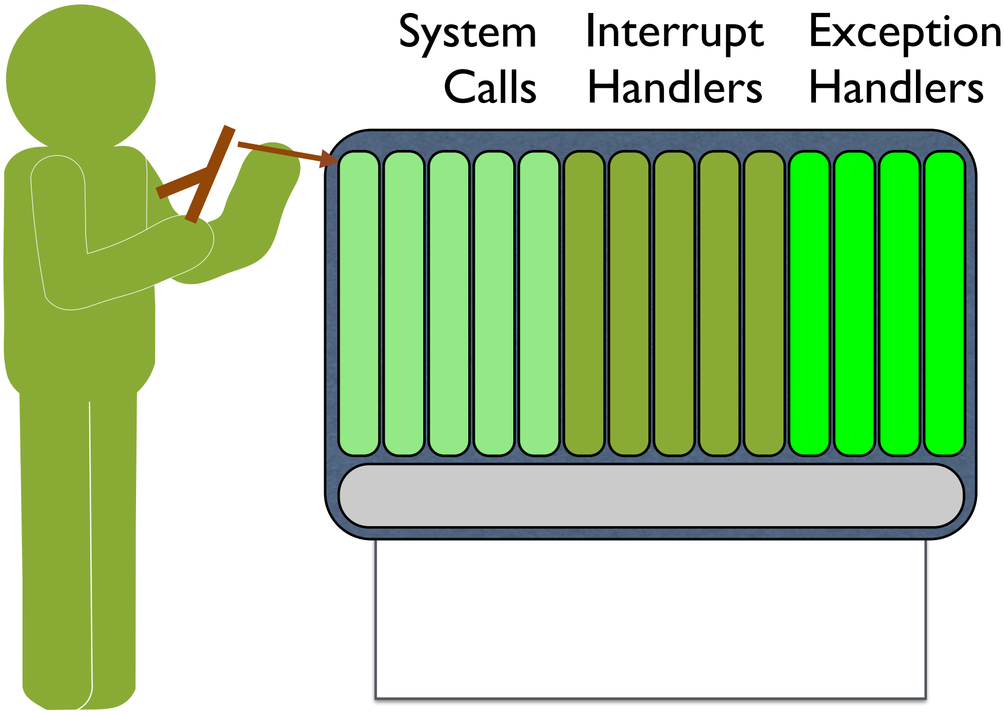
Using lambda ...



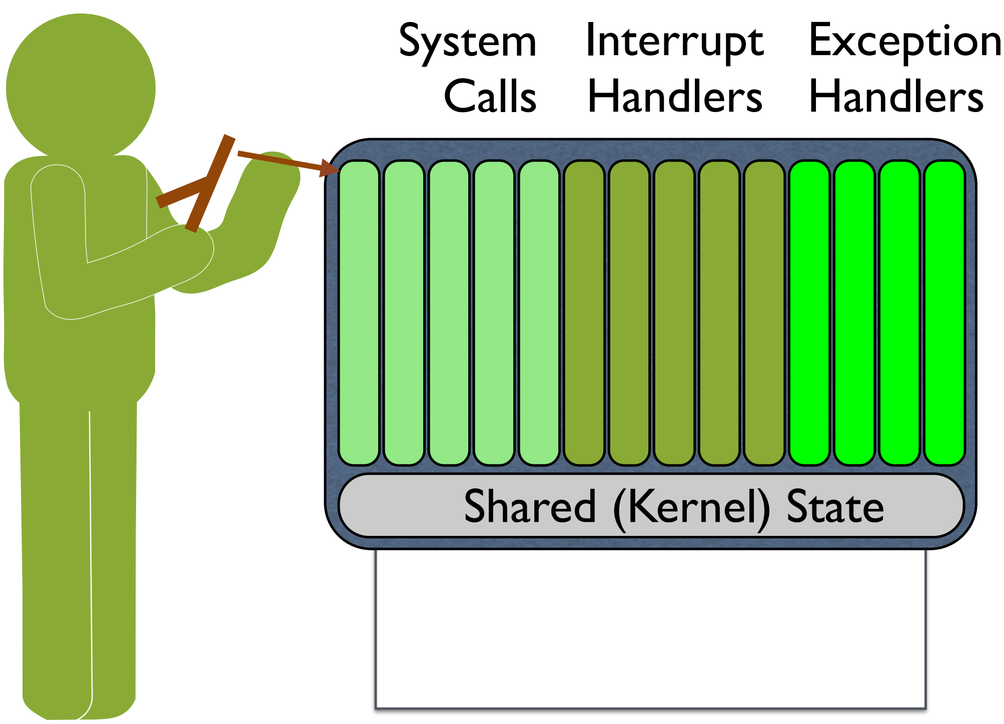
Using lambda ...

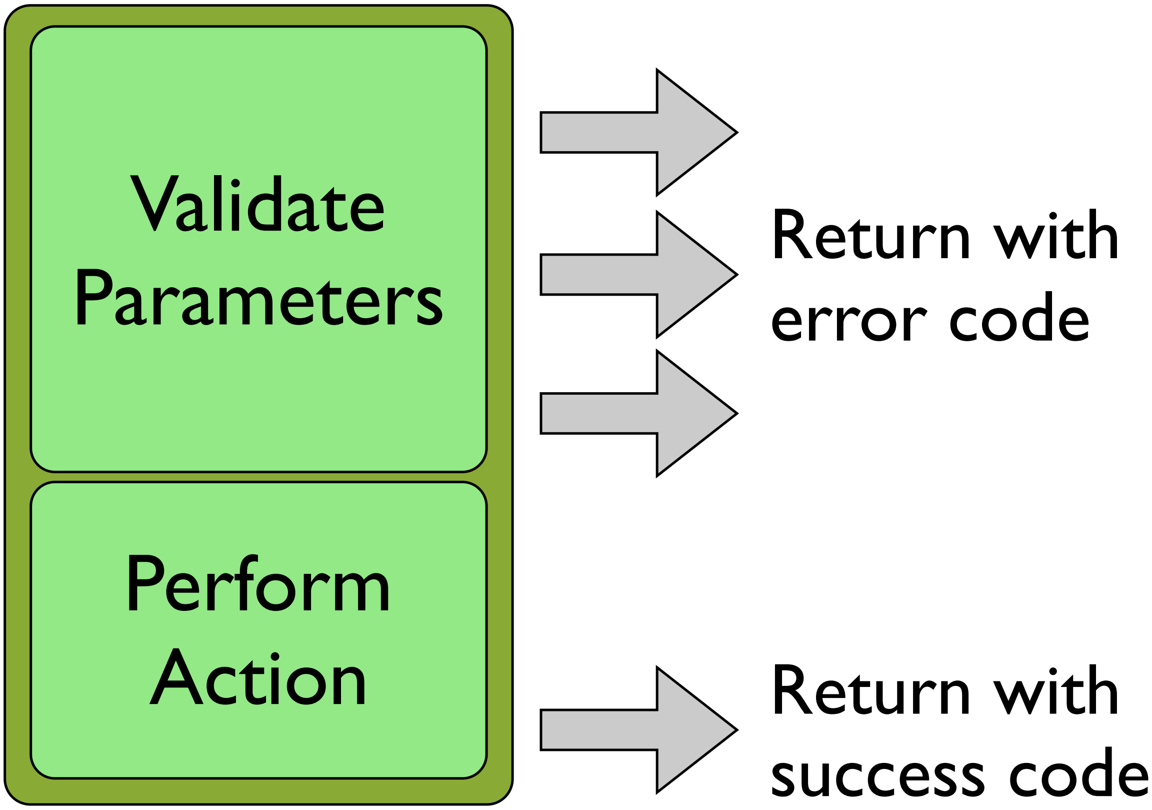


Using lambda ...



Using lambda ...






```
syscallMapPageDir :: (KE k, KW k) => k a
syscallMapPageDir
 = do curr <- getCurrent
      asidIdx <- getReg asidCapReg curr
      case<- lookupCapAll curr.cspace asidIdx of
        Ref asidCap ->
         case<- get asidCap.objptr of
           ASIDTableObj[] ->
             range  <- getCapdata asidCap
             offset <- getReg offsetReg
             case offset `inRange` range of
               Just asid ->
                 let slot = asidTable @@ asid
                 count <- get slot.count
                 if count==0 then
                   pdirIdx <- getReg pdirCapReg curr
                   case<- lookupCapAll curr.cspace pdirIdx curr of
                     Ref pdirCap ->
                       case<- get pdirCap.objptr of
                        PageDirObj[pdir] ->
                          case<- getCapdata pdirCap of
                            UnmappedPD[] ->
                              set slot.pdir (Ref pdir)
                              set slot.count 1
                              setCapdata pdcap MappedPD[asid]
                              success curr
                            _ -> mappedErr curr
                        _ -> invalidCapabilityErr curr
                     Null -> invalidCapabilityErr curr
                 else mappedErr curr
             Nothing -> rangeErrorErr curr
           _ -> invalidCapabilityErr curr
        Null -> invalidCapabilityErr curr
```

Parameter Validation

Action

Error Reporting

Imperative Functional Programming

- Traditional sequential control flow

```
do f  <- openFile "file.txt"
   l1 <- readLine f
   l2 <- readLine f
   out (l1, l2)
   closeFile f
```

- How to deal with errors? multiple results?
  - Make functions return error codes (and hope that callers will check those codes)?
  - Add the ability to throw and catch exceptions?
  - Use continuations …

Programming with continuations

- Instead of

```
openFile :: String -> Proc FileHandle
```

- Try:

higher-order, or first-class functions

```
openFile :: String
              -> (ErrorCode  -> Proc a)
              -> (FileHandle -> Proc a)
              -> Proc a
```

- It's as if we've given `openFile` *two* return addresses: one to use when an error occurs, and one to use when the call is successful.

Programming with continuations

- Our original program using continuations:

```
openFile "file.txt"
   (\error -> …)
   (\f -> do l₁ <- readLine f
             l₂ <- readLine f
             out (l₁, l₂)
             closeFile f)
```

- Could we do the same for `readLine`?

Programming with continuations

- Our original program using continuations:

```
openFile "file.txt"
  (\error -> …)
  (\f -> readLine f
      (\error -> …)
      (\l₁ -> readLine f
           (\error -> …)
           (\l₂ <- do out (l₁, l₂)
                      closeFile f)))
```

- Hmm, not so pretty …

Programming with continuations

- Name the error handlers:

```
openFile "file.txt"
  err₁
  (\f -> readLine f
      err₂
      (\l₁ -> readLine f
           err₃
           (\l₂ <- do out (l₁, l₂)
                      closeFile f)))
```


Programming with continuations

- Reformat:

```
openFile "file.txt" err₁ (\f  ->
readLine f          err₂ (\l₁ ->
readLine f          err₃ (\l₂ ->
do out (l₁, l₂)
   closeFile f)))
```

- Looking better …

Programming with continuations

- Add an infix operator: `f $ x = f x`

```
openFile "file.txt" err₁ $ \f  ->
readLine f          err₂ $ \l₁ ->
readLine f          err₃ $ \l₂ ->
do out (l₁, l₂)
   closeFile f
```

- Fewer parentheses …
- Easier to add or remove individual lines …
- … still a little cluttered by error handling behavior

Programming with continuations

- Continuation-based control flow, integrated error handlers:

```
openFile "file.txt" $ \f  ->
readLine f          $ \l₁ ->
readLine f          $ \l₂ ->
do out (l₁, l₂)
   closeFile f
```

- Not always applicable …
- … but a good choice for HaL4 where the response to a particular type of invalid parameter is always the same (typically, returning an error code to the caller)
- … and this also encourages consistent API behavior

“Validators”

The implementation of prototype HaL4 includes a small library of validator functions:

```
getCurrent           :: KR k => (TCBRef -> k a) -> k a
getRegCap            :: KE k => #r -> TCBRef
                                   -> (CapRef -> k a) -> k a
emptyCapability      :: KE k => TCBRef -> CapRef -> k a -> k a
cdtLeaf              :: KE k => TCBRef -> CapRef -> k a -> k a
notMaxDepth          :: KE k => TCBRef -> CapRef -> k a -> k a
untypedCapability    :: KE k => TCBRef -> CapRef
                               -> (UntypedRef -> k a) -> k a
pageDirCapability    :: KE k => TCBRef -> CapRef
                    -> (PageDirRef -> PDMapData -> k a) -> k a
pageTableCapability :: KE k => TCBRef -> CapRef
                    -> (PageTableRef -> MapData -> k a) -> k a
```


“Validators”

- In effect, we have built an embedded domain specific language, just for validating parameters in HaL4
- Benefits include:
    - Ease of reuse
    - Consistency
    - Clarity
    - Ability to pass multiple results on to continuation

```
syscallMapPageDir :: (KE k, KW k) => k a
syscallMapPageDir
 = getCurrent                           $ \curr      ->
   getMapPageDirASIDTab curr            $ \asidcap   ->
   asidTableCapability curr asidcap     $ \range     ->

   getMapPageDirOffset curr             $ \offset    ->
   asidInRange curr offset range        $ \asid      ->
   asidNotUsed curr asid                $ \slot      ->

   getMapPageDirPDir curr               $ \pdcap     ->
   pageDirCapability curr pdcap         $ \pdir pdmd ->
   unmappedPD curr pdmd                 $

   do set slot.pdir (Ref pdir)
      set slot.count 1
      setCapdata pdcap MappedPD[asid]
      success curr
```

Validators

Action


```
syscallMapPageDir :: (KE k, KW k) => k a
syscallMapPageDir
 = getCurrent                           $ \curr      ->
   getMapPageDirASIDTab curr            $ \asidcap   ->
   asidTableCapability curr asidcap     $ \range     ->

   getMapPageDirOffset curr             $ \offset    ->
   asidInRange curr offset range        $ \asid      ->
   asidNotUsed curr asid                $ \slot      ->

   getMapPageDirPDir curr               $ \pdcap     ->
   pageDirCapability curr pdcap         $ \pdir pdmd ->
   unmappedPD curr pdmd                 $

   do set slot.pdir (Ref pdir)
      set slot.count 1
      setCapdata pdcap MappedPD[asid]
      success curr
```

"clear" and "concise"

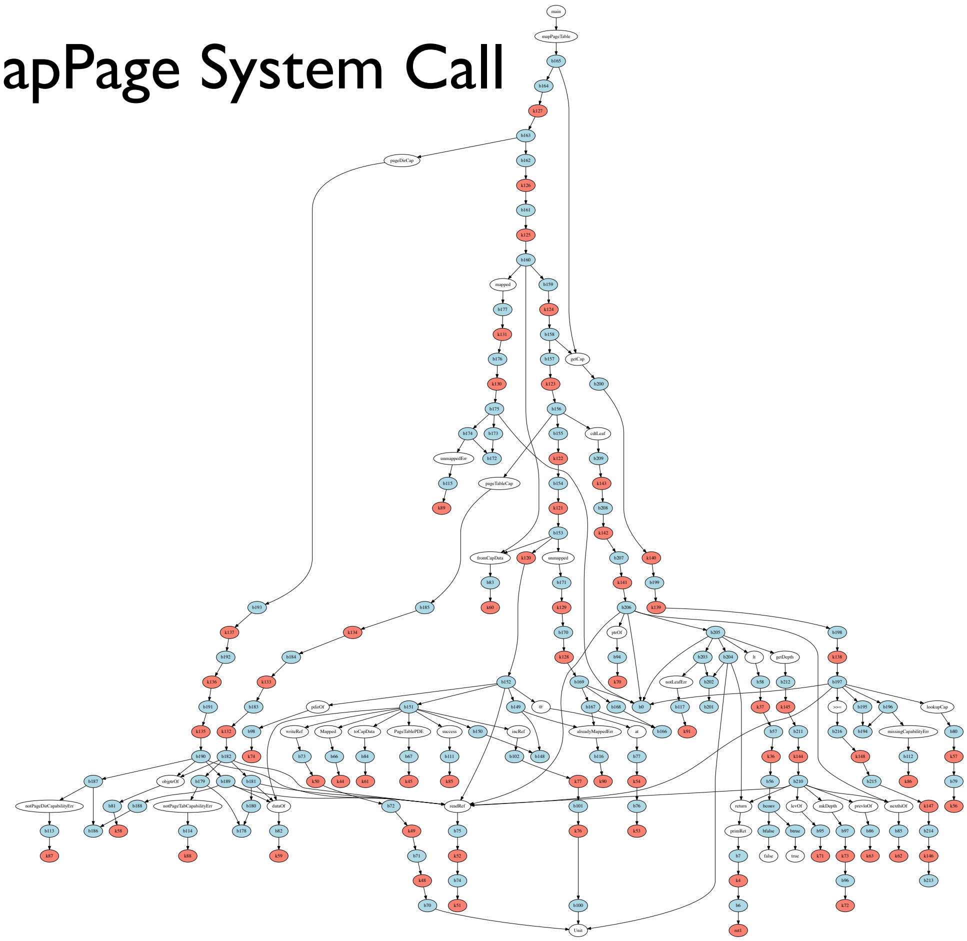
```
syscallMapPageDir :: (KE k, KW k) => k a
syscallMapPageDir
 = getCurrent                                   $ \curr      ->
   getMapPageDirASIDTab curr                    $ \asidcap   ->
   asidTableCapability curr asidcap             $ \range     ->

   getMapPageDirOffset curr                     $ \offset    ->
   asidInRange curr offset range                $ \asid      ->
   asidNotUsed curr asid                        $ \slot      ->

   getMapPageDirPDir curr                       $ \pdcap     ->
   pageDirCapability curr pdcap                 $ \pdir pdmd ->
   unmappedPD curr pdmd                         $

   do set slot.pdir (Ref pdir)
      set slot.count 1
      setCapdata pdcap MappedPD[asid]
      success curr
```

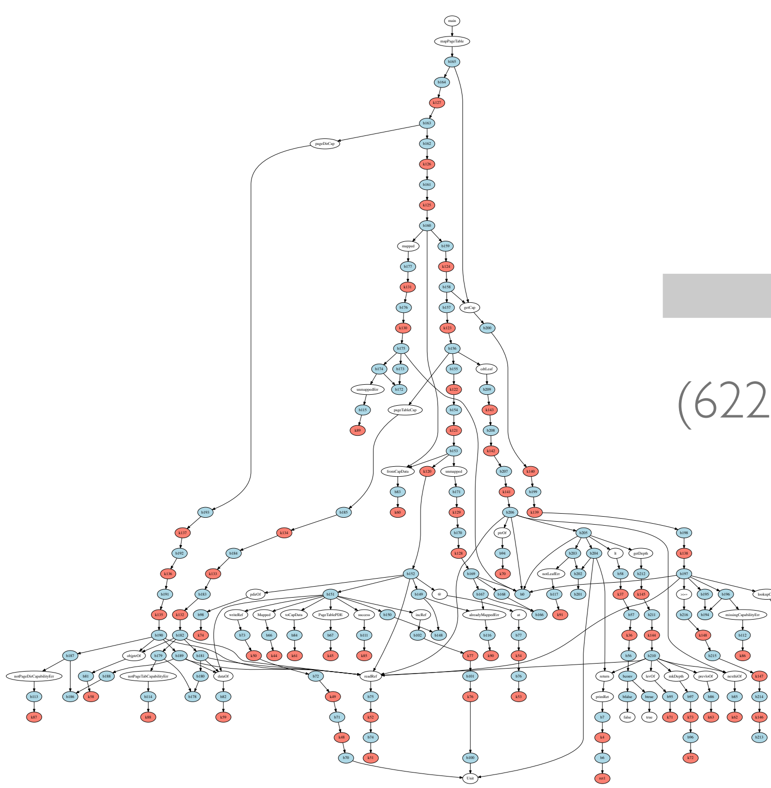
reusable


```
syscallMapPageDir :: (KE k, KW k) => k a
syscallMapPageDir
 = getCurrent                                   $ \curr      ->
   getMapPageDirASIDTab curr                    $ \asidcap   ->
   asidTableCapability curr asidcap             $ \range     ->

   getMapPageDirOffset curr                     $ \offset    ->
   asidInRange curr offset range                $ \asid      ->
   asidNotUsed curr asid                        $ \slot      ->

   getMapPageDirPDir curr                       $ \pdcap     ->
   pageDirCapability curr pdcap                 $ \pdir pdmd ->
   unmappedPD curr pdmd                         $

   do set slot.pdir (Ref pdir)
      set slot.count 1
      setCapdata pdcap MappedPD[asid]
      success curr
```

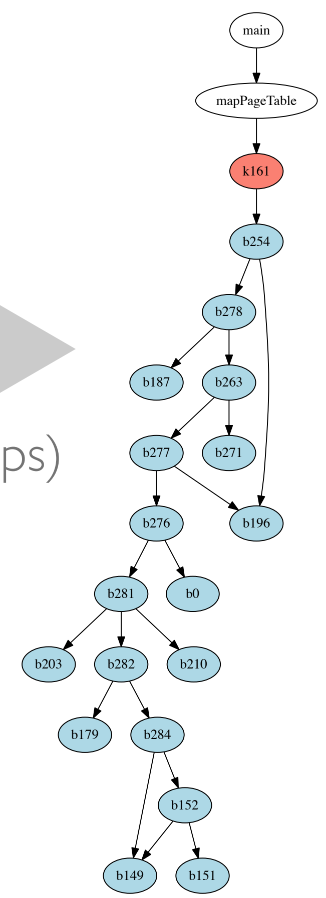
performance concerns?

The MapPage System Call



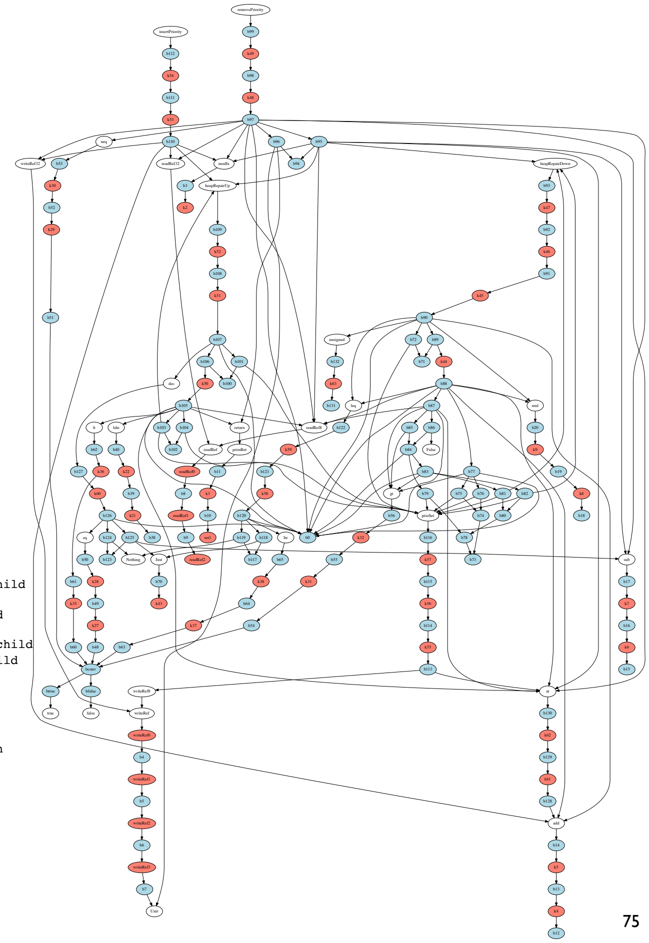


(622 steps)



Prioset

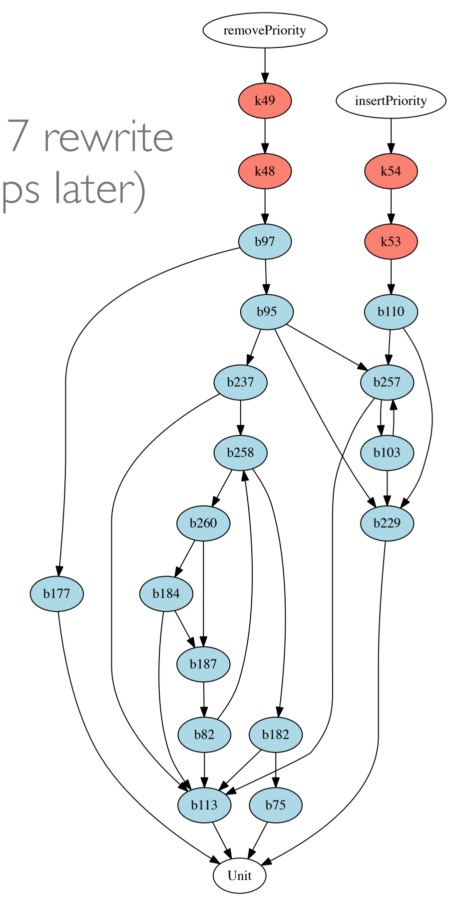
```
prioSet = \i prio -> do writeRef (at prioset i) prio
                        writeRef (at prioidx prio) i

insertPriority = \prio -> do s <- readRef priosetSize
                             writeRef priosetSize (add s 1)
                             heapRepairUp (modIx s) prio

heapRepairUp = \i prio ->
  case dec i of
    Nothing -> prioSet 0 prio
    Just j  -> do parent <- ret (shiftR j 1)
                  pprio  <- readRef (at prioset parent)
                  if lt pprio prio then
                    prioSet i pprio
                    heapRepairUp parent prio
                  else
                    prioSet i prio

removePriority = \prio ->
  do s <- readRef priosetSize
     writeRef priosetSize (sub s 1)
     rprio <- readRef (at prioset (modIx (sub s 1)))
     if neq prio rprio then
       i <- readRef (at prioidx prio)
       heapRepairDown i rprio (modIx (sub s 2))
       nprio <- readRef (at prioset i)
       heapRepairUp i nprio

heapRepairDown = \i prio last ->
 do let u = unsigned i  // <- ret (unsigned i)
    case leq (add (mul 2 u) 1) last of            // Look for a left child
     Nothing -> prioSet i prio                    // i has no children
     Just l  ->                                   // i has a left child
       do lprio <- readRef (at prioset l)
          case leq (add (mul 2 u) 2) last of      // Look for a right child
            Nothing ->                            // i has no right child
              if gt lprio prio then
                prioSet i lprio
                prioSet l prio
              else
                prioSet i prio
            Just r  ->                            // i has two children
              do rprio <- readRef (at prioset r)
                 if gt prio lprio && gt prio rprio then
                   prioSet i prio
                 else if gt lprio rprio then
                        prioSet i lprio // left is higher
                        heapRepairDown l prio last
                      else
                        prioSet i rprio // right is higher
                        heapRepairDown r prio last
```



Prioset

```
prioSet = \i prio -> do writeRef (at prioset i) prio
                        writeRef (at prioidx prio) i

insertPriority = \prio -> do s <- readRef priosetSize
                             writeRef priosetSize (add s 1)
                             heapRepairUp (modIx s) prio

heapRepairUp = \i prio ->
  case dec i of
    Nothing -> prioSet 0 prio
    Just j  -> do parent <- ret (shiftR j 1)
                  pprio  <- readRef (at prioset parent)
                  if lt pprio prio then
                    prioSet i pprio
                    heapRepairUp parent prio
                  else
                    prioSet i prio

removePriority = \prio ->
  do s <- readRef priosetSize
     writeRef priosetSize (sub s 1)
     rprio <- readRef (at prioset (modIx (sub s 1)))
     if neq prio rprio then
       i <- readRef (at prioidx prio)
       heapRepairDown i rprio (modIx (sub s 2))
       nprio <- readRef (at prioset i)
       heapRepairUp i nprio

heapRepairDown = \i prio last ->
 do let u = unsigned i  // <- ret (unsigned i)
    case leq (add (mul 2 u) 1) last of            // Look for a left child
     Nothing -> prioSet i prio                    // i has no children
     Just l  ->                                   // i has a left child
       do lprio <- readRef (at prioset l)
          case leq (add (mul 2 u) 2) last of      // Look for a right child
            Nothing ->                            // i has no right child
              if gt lprio prio then
                prioSet i lprio
                prioSet l prio
              else
                prioSet i prio
            Just r  ->                            // i has two children
              do rprio <- readRef (at prioset r)
                 if gt prio lprio && gt prio rprio then
                   prioSet i prio
                 else if gt lprio rprio then
                        prioSet i lprio // left is higher
                        heapRepairDown l prio last
                      else
                        prioSet i rprio // right is higher
                        heapRepairDown r prio last
```


(1217 rewrite steps later)



Wrapping Up ...

Current status

- For the three main questions for CEMLaBS:
  - **Feasibility**: Still chipping away ... but getting closer!
  - **Benefit**: Good evidence that we will benefit from the use of functional language features
    - +Types
    - +Higher-order functions
  - **Performance**: acceptable performance may be within reach
    - +We can generate good quality code, even when lambdas are used in fundamental ways
    - +Some code duplication (but, so far, this is entirely tolerable for our specific use case ...)

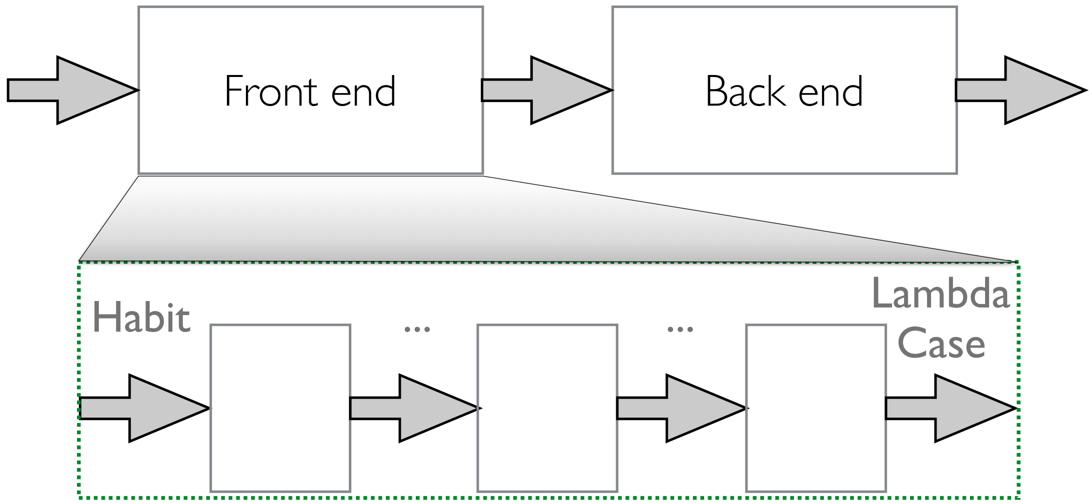
Acknowledgement (likely incomplete!)

Numerous people at PSU (and beyond) have contributed to the design and implementation of Habit, including:

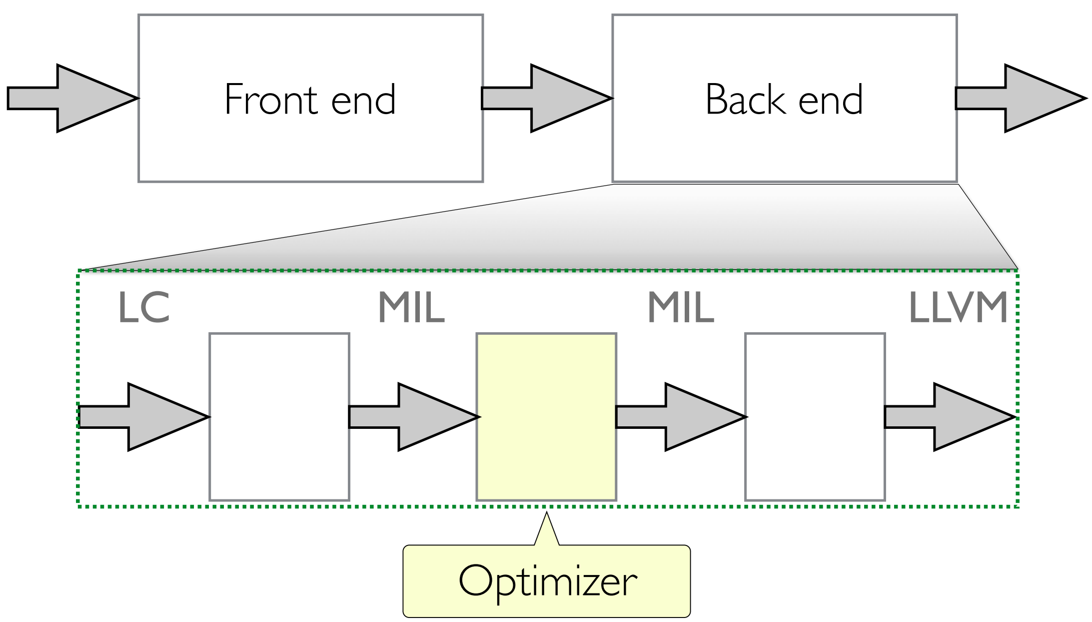
- Michael Adams
- Aaron Altman
- Justin Bailey
- Tim Chevalier
- Lewis Coates
- Ted Cooper
- Dan Cristofani
- Iavor Diatchki
- Thomas DuBuisson
- Kenneth Graunke
- Thomas Hallgren
- Tom Harke
- Caylee Hogg
- Jim Hook
- Brian Huffman
- Mark Jones
- Dick Kieburtz
- Rebekah Leslie-Hurd
- John Matthews
- Andrew McCreight
- Garrett Morris
- Ryan Niebur
- Andrew Sackville-West
- Andrew Tolmach
- Peter White
- ...

Some Words about the Habit Implementation

The Habit Compiler



The Habit Compiler



Why MIL?

- If we want a good optimizer, we need to work in a language that exposes key implementation details/sources of overhead
- Constructing a closure: `k{x1, …, xn}`
  - code pointer: `k`
  - stored fields: `x1, …, xn`
- Entering a closure: If `f` is a closure, then we write `f @ x` for the result of entering `f` with argument `x`
- Defining a closure: `k{x1, …, xn} a = t`
  - The code in `t` describes the result that is produced when you enter the closure with argument `a`

From Functional Source Code ...

```
id      = \x -> x

compose = \f g x -> f (g x)

map     = \f xs ->
            case xs of
              Nil         -> Nil
              Cons y ys -> Cons (f y) (map f ys)
```

... to MIL Programs

```
id          ← k0{}
k0{} x  =  b0[x]
b0[x]   =  return x

map             ← k4{}
k4{} f          =  k5{f}
k5{f} xs        =  b2[f,xs]
b2[f,xs]        =  case xs of
                      Nil() ⟶ b3[]
                      Cons(y,ys) ⟶ b4[f,y,ys]
b3[]            =  Nil()
b4[f,y,ys]      =  z ← f @ y
                   m ← map @ f
                   zs ← m @ ys
                   Cons(z,zs)
```

```
compose     ← k1{}
k1{} f      =  k2{f}
k2{f} g     =  k3{f,g}
k3{f,g} x   =  b1[f,g,x]
b1[f,g,x]   =  y ← g @ x
               f @ y
```

Intuition: arguments are like registers that have been loaded with values on entry to a basic block of code

... to Optimized MIL Programs

```
map             ← k4{}
k4{} f          =  k5{f}
k5{f} xs        =  b2[f,xs]
b2[f,xs]        =  case xs of
                      Nil() ⟶ b3[]
                      Cons(y,ys) ⟶ b4[f,y,ys]
b3[]            =  Nil()
b4[f,y,ys]      =  z ← f @ y
                   m ← map @ f
                   zs ← m @ ys
                   Cons(z,zs)
```

unknown function call

known function call

... to Optimized MIL Programs

```
map            ← k4{}
k4{} f         = k5{f}
k5{f} xs       = b2[f,xs]
b2[f,xs]       = case xs of
                   Nil()⟶ b3[]
                   Cons(y,ys)⟶ b4[f,y,ys]
b3[]           = Nil()
b4[f,y,ys]     = z ← f @ y
                 m ← map @ f
                 zs ← m @ ys
                 Cons(z,zs)
```

known function call

... to Optimized MIL Programs

```
map            ← k4{}
k4{} f         = k5{f}
k5{f} xs       = b2[f,xs]
b2[f,xs]       = case xs of
                   Nil()⟶ b3[]
                   Cons(y,ys)⟶ b4[f,y,ys]
b3[]           = Nil()
b4[f,y,ys]     = z ← f @ y
                 m ← k5{f}
                 zs ← m @ ys
                 Cons(z,zs)
```

known function call

... to Optimized MIL Programs

```
map           ← k4{}
k4{} f        = k5{f}
k5{f} xs      = b2[f,xs]
b2[f,xs]      = case xs of
                  Nil()⟶ b3[]
                  Cons(y,ys)⟶ b4[f,y,ys]
b3[]          = Nil()
b4[f,y,ys]    = z ← f @ y
                m ← k5{f}
                zs ← m @ ys
                Cons(z,zs)
```

known function call

... to Optimized MIL Programs

```
map           ← k4{}
k4{} f        = k5{f}
k5{f} xs      = b2[f,xs]
b2[f,xs]      = case xs of
                  Nil()⟶ b3[]
                  Cons(y,ys)⟶ b4[f,y,ys]
b3[]          = Nil()
b4[f,y,ys]    = z ← f @ y
                m ← k5{f}
                zs ← b2[f,ys]
                Cons(z,zs)
```

pure, dead code

… to Optimized MIL Programs

```
map           ← k₄{}
k₄{} f        = k₅{f}
k₅{f} xs      = b₂[f,xs]
b₂[f,xs]      = case xs of
                  Nil()⟶ b₃[]
                  Cons(y,ys)⟶ b₄[f,y,ys]
b₃[]          = Nil()
b₄[f,y,ys]    = z ← f @ y
                zs ← b₂[f,ys]
                Cons(z,zs)
```


MIL Optimization

- Basic strategy:
  - many small rewrites
  - combined in large numbers
- Sources of rewrites:
  - algebraic laws
  - simple data flow
  - specialization and derived blocks