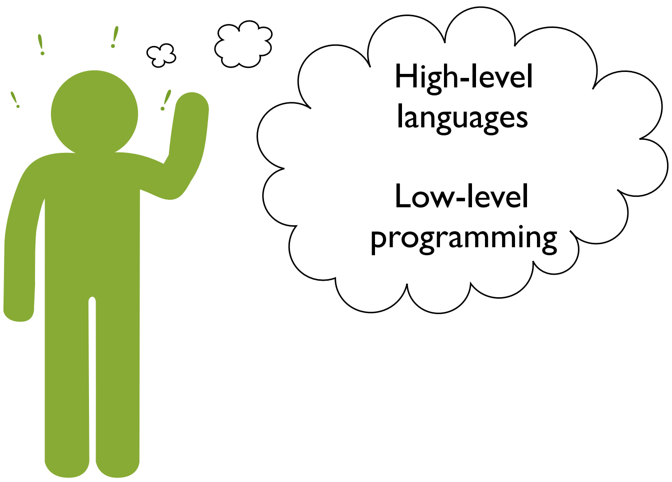
CS 410/510

Languages & Low-Level Programming

Mark P Jones
Portland State University

Spring 2016

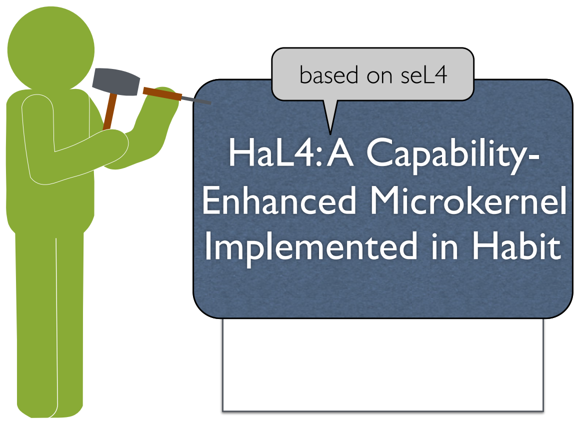
Week 10: Habit and HaL4



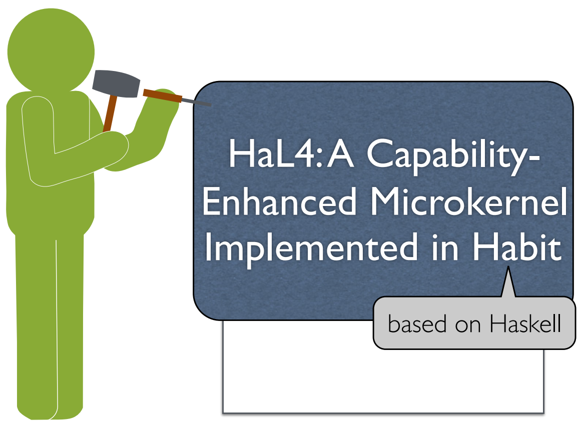
The CEMLaBS Project

- "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?

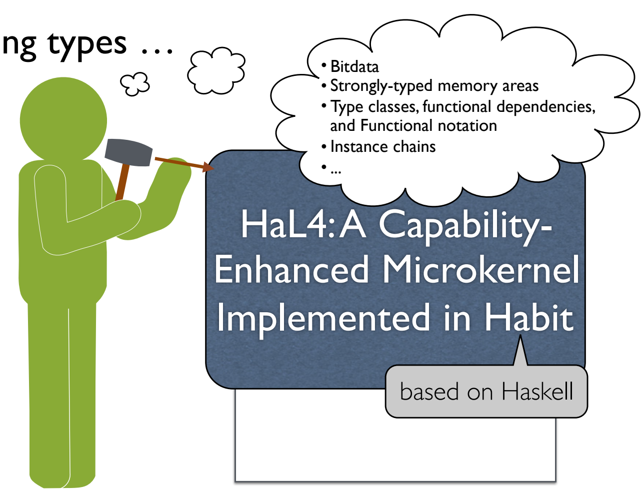
Chipping away ...



Chipping away ...



Using types …



Example: IA32 Paging Structures

Remember this?

31–22	21–17	16–13	12	11–9	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

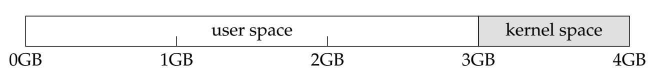
Example: IA32 Paging Structures

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 Address Space Layout

Remember this?

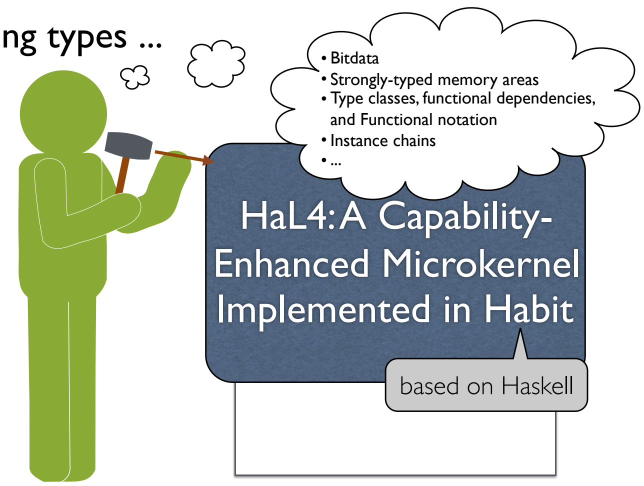



```
struct PageDir /4K [ pdes  :: Array UserSuperPages   (Stored PDE)
                   | kpdes :: Array KernelSuperPages (Stored KPDE) ]

type UserSuperPages   = 768  -- Number of superpages in the user space address range
type KernelSuperPages = 256  -- Number of superpages in the kernel space address range

bitdata KPDE /WordSize    -- Kernel Page Directory Entries, a variant of SuperPagePDE
 = KPDE [ B00                                -- leading zeros
        | ix       :: Ix KernelSuperPages    -- physical superpage number
        | unused=0 :: Bit 13
        | global=1 :: Bit 1                  -- 1 => global translation (if cr4.pge=1)
        | B1                                 -- indicates a form of SuperPagePDE
        | attrs = kernelOnly :: PagingAttrs  -- paging attributes
        | B1 ]

instance Initable PageDir where
  initialize = PageDir [ pdes  <- initArray (\ix -> initStored UnmappedPDE[])
                       | kpdes <- initArray (\ix -> initStored KPDE[ix]) ]
```

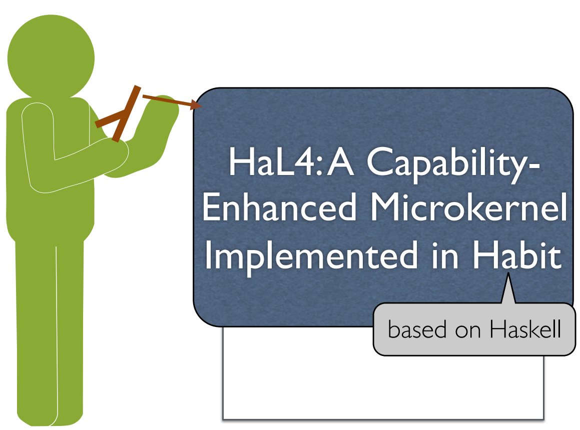
Summary

- The code is not simple …
- … but this reflects the underlying data structures
- Correct usage is enforced by the type system:
  - The separation between user space and kernel space
  - The requirement for physical rather than virtual addresses in page directory entries
  - Invariants on data structure size
  - Initialization

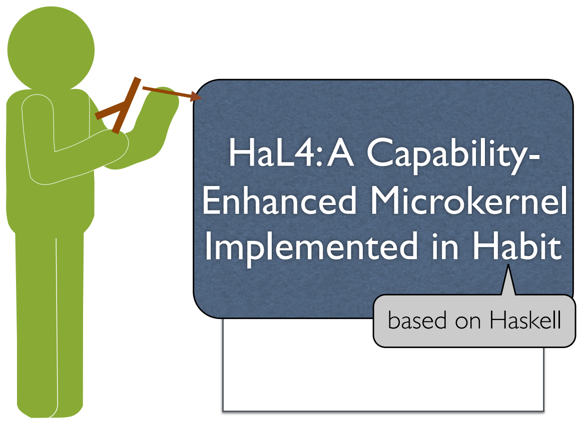
Using types ...



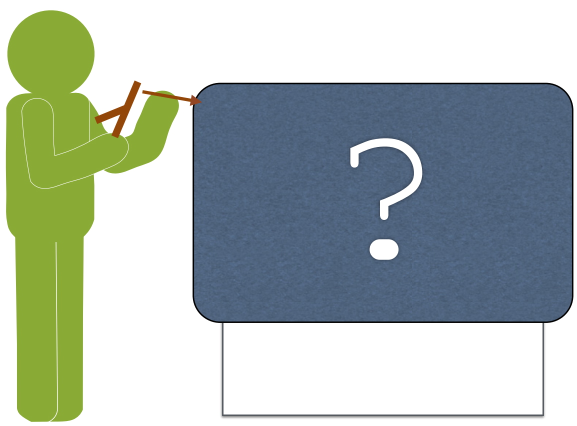
Using lambda ...



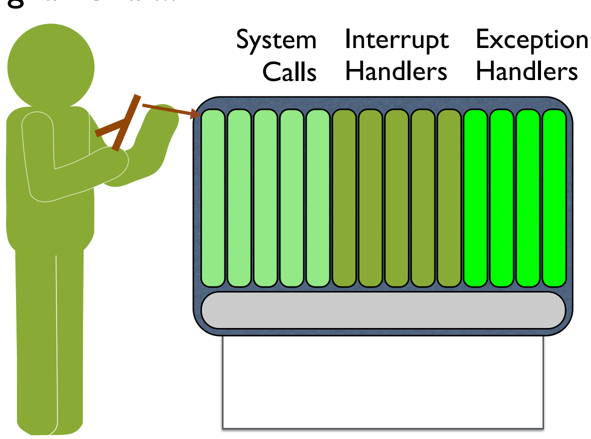
Using lambda ...



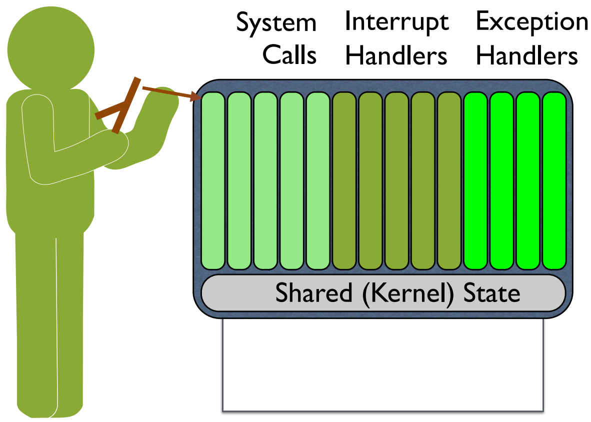
Using lambda ...

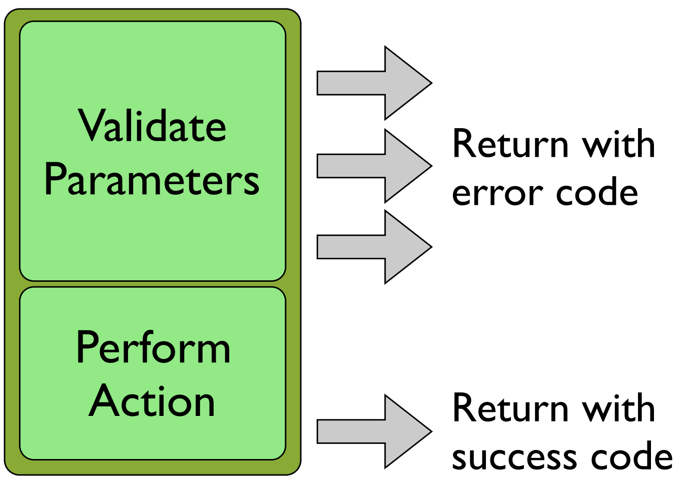


Using lambda ...



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

Programming with continuations

- Traditional control flow structure

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

- How to deal with errors?
  - 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 -> IO FileHandle
```

- Try:

higher-order, or first-class functions

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

lambda expressions: \var -> expr

```
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

- Integrate the error handlers in to the main operations:

```
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 ensures a consistent API

"Validators"

The implementation of 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               $
```

Validators

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

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
```

high-level


```
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?

Other examples: reading/writing registers

```
syscallRetype :: (KE k, RW k) => k a
syscallRetype
  = getCurrent                     $ \curr  ->
    getRootCap curr                $ \root  ->
    untypedCapability curr root    $ \ut    ->
    cdtLeaf curr root              $
    notMaxDepth curr root          $
    getCNodeCap curr               $ \cncap ->
    cnodeCapability curr cncap     $ \cnode ->
    getCapArray curr cnode         $ \caps  ->
    allEmptyCapabilities curr caps $
    validType curr ut caps         $ \objs  ->
    addChildren curr root caps objs
```


Other examples: reading/writing registers

```
syscallReadRegs  :: (KE k, RW k) => k a
syscallReadRegs
   = getCurrent                    $ \curr   ->   -- get current thread
     getDstCap curr                $ \cap    ->   -- get specified capability
     tcbCapability curr cap        $ \tcb ps ->   -- ... to a tcb object
     permitsRead curr ps.perms     $              -- ... that we can read
     transferRegs tcb (>->) curr

syscallWriteRegs :: (KE k, RW k) => k a
syscallWriteRegs
   = getCurrent                    $ \curr   ->   -- get current thread
     getDstCap curr                $ \cap    ->   -- get specified capability
     tcbCapability curr cap        $ \tcb ps ->   -- ... to a tcb object
     permitsWrite curr ps.perms    $              -- ... that we can write
     transferRegs tcb (<-<) curr
```

```
r1 <-< r2  =  r2 >-> r1
```

```
r1 >-> r2
  =  do x <- readRef r1
        writeRef r2 x
```

```
r1 >-> r2
  = readRef r1 >>= writeRef r2
```


Other examples: transferring registers

```
transferRegs :: KE k => TCBRef
                     -> (RegRef -> RegRef -> k ())
                     -> TCBRef -> k a
transferRegs tcb xfer curr
   = do let tcbIframe = tcb.context.iframe
            currRegs  = curr.context.regs
        tcbIframe.eip `xfer` currRegs.esi
        tcbIframe.esp `xfer` currRegs.ecx
        case<- getIPCBuffer curr of
          Ref buf -> let tcbRegs = tcb.context.regs
                     tcbRegs.eax `xfer` (buf.mrs @@ 3)
                     tcbRegs.ebx `xfer` (buf.mrs @@ 4)
                     tcbRegs.ecx `xfer` (buf.mrs @@ 5)
                     tcbRegs.edx `xfer` (buf.mrs @@ 6)
                     tcbRegs.esi `xfer` (buf.mrs @@ 7)
                     tcbRegs.edi `xfer` (buf.mrs @@ 8)
                     tcbRegs.ebp `xfer` (buf.mrs @@ 9)
                     success curr
```

<-< or >-> would work here!

Other examples: reading/writing registers

```
syscallReadRegs  :: (KE k, RW k) => k a
syscallReadRegs
   = getCurrent                   $ \curr   ->   -- get current thread
     getDstCap curr               $ \cap    ->   -- get specified capability
     tcbCapability curr cap       $ \tcb ps ->   -- ... to a tcb object
     permitsRead curr ps.perms    $              -- ... that we can read
     transferRegs tcb (>->) curr

syscallWriteRegs :: (KE k, RW k) => k a
syscallWriteRegs
   = getCurrent                   $ \curr   ->   -- get current thread
     getDstCap curr               $ \cap    ->   -- get specified capability
     tcbCapability curr cap       $ \tcb ps ->   -- ... to a tcb object
     permitsWrite curr ps.perms   $              -- ... that we can write
     transferRegs tcb (<-<) curr
```


Other examples: reading/writing registers

```
syscallReadRegs  :: (KE k, RW k) => k a
syscallReadRegs
   = regAccess               $ \curr tcb ps -> -- find thread
     permitsRead curr ps     $                 -- check for read perm
     transferRegs tcb (>->) curr

syscallWriteRegs :: (KE k, RW k) => k a
syscallWriteRegs
   = regAccess               $ \curr tab ps -> -- find thread
     permitsWrite curr ps    $                 -- check for write perm
     transferRegs tcb (<-<) curr

regAccess :: (KE k) => (TCBRef -> TCBRef -> Perms -> k a) -> k a
regAccess k
   = getCurrent             $ \curr   ->   -- get current thread
     getDstCap curr         $ \cap    ->   -- get specified capability
     tcbCapability curr cap $ \tcb ps ->   -- ... to a tcb object
     k curr tcb ps.perms
```


Representing function values

- How should we represent values of type `Int -> Int`?
  - There are many different values, including: `(\z -> x+z)`, `(\x -> x+1)`, `(\x -> x*2)`, `(\x -> f(g(x)))`, ...
  - ... any of which could be passed as arguments to other functions ...
  - ... so we need a uniform, but flexible way to represent them
- A common answer is to represent functions like these by a pointer to a "closure", a heap allocated object that contains:
  - a code pointer (i.e., the code for the function)
  - the values of its free variables

Closures

- *Every* function of type `Int -> Int` will be represented using the same basic structure:

codeptr	...

- The code pointer and list of variables vary from one function value to the next:

`(\z -> x+z)`	$codeptr_1$	x	
`(\x -> x+1)`	$codeptr_2$		
`(\x -> x*2)`	$codeptr_3$		
`(\x -> f(g(x)))`	$codeptr_4$	f	g

- To make a closure, allocate a suitably sized block of memory and save the required code pointer and variable values

Calling unknown functions

- If `f` is a known function, then we call it by pushing its arguments on the stack and jumping directly to its code
- What if `f` is an unknown function, represented by a variable that points to a closure structure instead?
- The System V ABI doesn't cover this case, but we can make up our own convention:
  - push the arguments
  - push a pointer to the closure (for access to free variables)
  - call the code that is pointed to at the start of the closure
- This process is known as "entering a closure"

Thunks

- A closure that can be entered without any arguments is sometimes referred to as a "thunk"
- Thunks represent suspended/delayed computations that can be executed (or "entered" or "invoked") repeatedly at a later stage.
- Examples:
  - procedure values / monadic computations

```
r1 >-> r2  =   do x <- readRef r1
                  writeRef r2 x
```

  - interrupt/exception/system call handlers
  - …

Pros/cons of using higher order functions

- Pros:
  - high-level
  - clear and concise
  - facilitates reuse (less code, greater consistency)
- Cons:
  - what impact do all these closures have on performance?
    - it's hard to explore this question without a way to talk about these things more precisely …
    - in other words, we need a *language* that we can use to express these ideas…

A notation for closures and thunks

- Closures: `k{x₁, …, xₙ}`
  - code pointer: `k`
  - stored fields: $x_1, \ldots, x_n$
  - If `f` is a closure, then we write `f @ x` for the result of entering `f` with argument `x`
- Thunks: `m[x₁, …, xₙ]`
  - code pointer: `m`
  - stored fields: $x_1, \ldots, x_n$
  - if `t` is a thunk, then we write `invoke t` for the result that is obtained by invoking `t`

Do notation and three address code

- Some uses of do notation look a lot like three address code:

```
do x <- f y z          x := y + z
   t <- g x y          t := x * y
   u <- h t            u := -t
   p x t               goto p
```

- Some optimization techniques are like algebraic properties:

```
(do x <- return y; c)   =   [y/x] c
```

- Is this a "monad law" or is it "copy propagation"?

Comprehending Monads

Types

$$
\begin{aligned}
K^* &= K \\
(U \to V)^* &= (U^* \to M\ V^*) \\
(U, V)^* &= (U^*, V^*)
\end{aligned}
$$

Terms

$$
\begin{aligned}
x^* &= [\, x \,]^M \\
(\lambda x \to v)^* &= [\, (\lambda x \to v^*) \,]^M \\
(t\ u)^* &= [\, y \mid f \leftarrow t^*,\ x \leftarrow u^*,\ y \leftarrow (f\ x) \,]^M \\
(u, v)^* &= [\, (x, y) \mid x \leftarrow u^*,\ y \leftarrow v^* \,]^M \\
(\mathit{fst}\ t)^* &= [\, (\mathit{fst}\ z) \mid z \leftarrow t^* \,]^M
\end{aligned}
$$

Assumptions

$$(x_1 :: T_1, \ldots, x_n :: T_n)^* = x_1 :: T_1^*, \ldots, x_n :: T_n^*$$

Typings

$$(A \vdash t :: T)^* = A^* \vdash t^* :: M\ T^*$$

Figure 7: Call-by-value translation.

Moggi,
Wadler,
Kennedy,
Benton,
...
Bailey

From source terms ...

```
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

compose    ← k1{}
k1{} f     = k2{f}
k2{f} g    = k3{f,g}
k3{f,g} x  = b1(f,g,x)
b1(f,g,x)  = u ← g @ x
             f @ u

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)
```

MIL, a monadic intermediate language

```
a    ::= v                              -- variable
      |  n                              -- integer literal

t    ::= return a                       -- simple value
      |  p((a1,...,an))                 -- primitive call
      |  b(a1,...,an)                   -- block call
      |  C(a1,...,an)                   -- data constructor
      |  k{a1,...,an}                   -- closure constructor
      |  f @ a                          -- enter closure
      |  m[a1,...,an]                   -- thunk constructor
      |  invoke a                       -- invoke closure

c    ::= v <- t; c                      -- monadic bind
      |  t                              -- (generalized) tail call
      |  case v of alt1; ...; altn      -- conditional branch

alt  ::= C(v1,...,vn) -> b(a1,...,an)   -- constructor match
      |  _            -> b(a1,...,an)   -- default match

def  ::= b(v1, ..., vn)   = c           -- block entry point
      |  k{v1, ..., vn} v = c           -- closure definition
      |  v <- t                         -- top-level constant
```


MIL, a monadic intermediate language

```
a    ::= v                              -- variable
      |  n                              -- integer literal

t    ::= return a                       -- simple value
      |  p((a1,...,an))                 -- primitive call
      |  b(a1,...,an)                   -- block call
      |  C(a1,...,an)                   -- data constructor
      |  k{a1,...,an}                   -- closure constructor
      |  f @ a                          -- enter closure
      |  m[a1,...,an]                   -- thunk constructor
      |  invoke a                       -- invoke closure

c    ::= v <- t; c                      -- monadic bind
      |  t                              -- (generalized) tail call
      |  case v of alt1; ...; altn      -- conditional branch

alt  ::= C(v1,...,vn) -> b(a1,...,an)   -- constructor match
      |  _            -> b(a1,...,an)   -- default match

def  ::= b(v1, ..., vn)   = c           -- block entry point
      |  k{v1, ..., vn} v = c           -- closure definition
      |  v <- t                         -- top-level constant
```

Definitions

MIL, a monadic intermediate language

```
a    ::= v                              -- variable
      |  n                              -- integer literal

t    ::= return a                       -- simple value
      |  p((a1,...,an))                 -- primitive call
      |  b(a1,...,an)                   -- block call
      |  C(a1,...,an)                   -- data constructor
      |  k{a1,...,an}                   -- closure constructor
      |  f @ a                          -- enter closure
      |  m[a1,...,an]                   -- thunk constructor
      |  invoke a                       -- invoke closure

c    ::= v <- t; c                      -- monadic bind
      |  t                              -- (generalized) tail call
      |  case v of alt1; ...; altn      -- conditional branch

alt  ::= C(v1,...,vn) -> b(a1,...,an)   -- constructor match
      |  _            -> b(a1,...,an)   -- default match

def  ::= b(v1, ..., vn)   = c           -- block entry point
      |  k{v1, ..., vn} v = c           -- closure definition
      |  v <- t                         -- top-level constant
```

Code

MIL, a monadic intermediate language

Tails

```
a    ::= v                          -- variable
       | n                          -- integer literal

t    ::= return a                   -- simple value
       | p((a_1,...,a_n))           -- primitive call
       | b(a_1,...,a_n)             -- block call
       | C(a_1,...,a_n)             -- data constructor
       | k{a_1,...,a_n}             -- closure constructor
       | f @ a                      -- enter closure
       | m[a_1,...,a_n]             -- thunk constructor
       | invoke a                   -- invoke closure

c    ::= v <- t; c                  -- monadic bind
       | t                          -- (generalized) tail call
       | case v of alt_1; ...; alt_n  -- conditional branch

alt  ::= C(v_1,...,v_n) -> b(a_1,...,a_n)  -- constructor match
       | _              -> b(a_1,...,a_n)  -- default match

def  ::= b(v_1, ..., v_n)   = c     -- block entry point
       | k{v_1, ..., v_n} v = c     -- closure definition
       | v <- t                     -- top-level constant
```


MIL, a monadic intermediate language

Atoms

```
a    ::= v                          -- variable
       | n                          -- integer literal

t    ::= return a                   -- simple value
       | p((a_1,...,a_n))           -- primitive call
       | b(a_1,...,a_n)             -- block call
       | C(a_1,...,a_n)             -- data constructor
       | k{a_1,...,a_n}             -- closure constructor
       | f @ a                      -- enter closure
       | m[a_1,...,a_n]             -- thunk constructor
       | invoke a                   -- invoke closure

c    ::= v <- t; c                  -- monadic bind
       | t                          -- (generalized) tail call
       | case v of alt_1; ...; alt_n  -- conditional branch

alt  ::= C(v_1,...,v_n) -> b(a_1,...,a_n)  -- constructor match
       | _              -> b(a_1,...,a_n)  -- default match

def  ::= b(v_1, ..., v_n)   = c     -- block entry point
       | k{v_1, ..., v_n} v = c     -- closure definition
       | v <- t                     -- top-level constant
```


... to MIL programs

```
id       ← k_0{}
k_0{} x  = b_0(x)
b_0(x)   = return x

map            ← k_4{}
k_4{} f        = k_5{f}
k_5{f} xs      = b_2(f,xs)
b_2(f,xs)      = case xs of
                   Nil() → b_3()
                   Cons(y,ys) → b_4(f,y,ys)

b_3()          = Nil()
b_4(f,y,ys)    = z ← f @ y
                 m ← map @ f
                 zs ← m @ ys
                 Cons(z,zs)
```

unknown function call

known function call

```
compose     ← k_1{}
k_1{} f     = k_2{f}
k_2{f} g    = k_3{f,g}
k_3{f,g} x  = b_1(f,g,x)
b_1(f,g,x)  = u ← g @ x
              f @ u
```

... to MIL programs

```
id        ← k0{}
k0{} x  = b0(x)
b0(x)   = return x

compose     ← k1{}
k1{} f      = k2{f}
k2{f} g     = k3{f,g}
k3{f,g} x   = b1(f,g,x)
b1(f,g,x)   = u ← g @ x
              f @ u

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

... to optimized MIL programs

```
id        ← k0{}
k0{} x  = b0(x)
b0(x)   = return x

compose     ← k1{}
k1{} f      = k2{f}
k2{f} g     = k3{f,g}
k3{f,g} x   = b1(f,g,x)
b1(f,g,x)   = u ← g @ x
              f @ u

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)
```

unknown function call

known function call

... to optimized MIL programs

```
id        ← k0{}
k0{} x  = b0(x)
b0(x)   = return x

compose     ← k1{}
k1{} f      = k2{f}
k2{f} g     = k3{f,g}
k3{f,g} x   = b1(f,g,x)
b1(f,g,x)   = u ← g @ x
              f @ u

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)
```

unknown function call

... to optimized 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 ← k5{f}
               zs ← b2(f,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)    = u ← g @ x
               f @ u
```

pure, dead code

... to optimized 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
               zs ← b2(f,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)    = u ← g @ x
               f @ u
```


Constant/copy propagation / left monad law:

`x ← return a; c` ⟹ `[a/x] c`

Tail call introduction / right monad law:

`x ← t; return x` ⟹ `t`

Prefix inlining: `b(x) = v₁←t₁; ...; vₙ←tₙ; t`
(monad associativity law)

`v←b(x); c` ⇒ `v₁←t₁;...;vₙ←tₙ;v←t;c`

Suffix inlining:

`v←t₀; b(x)` ⇒ `v←t₀;v₁←t₁;...;vₙ←tₙ;t`

Wildcard introduction:

`v←t; c` ⇒ `_ ←t; c` (`v` not free in `c`)

Dead tail elimination:

`_←t; c` ⇒ `c` (`t` pure)

Identity laws:

`or((x,0))` ⇒ `return x`

Idempotence:

`or((x,x))` ⇒ `return x`

Common subexpression elimination:

```
{y = t}
v ←t
```

⇒ `v ←return y`

Constant folding:

(M, N constants)

```
{x=M, y=N}
v ← and((x,y))   ⇒   v ← return (M&N)
```

Argument ordering:

(M, N constants)

```
v←and((M,x))   ⇒   v←and((x,M))
```

Associative folding:

```
                     (u&M)&N  =  u&(M&N)
{v=and((w,M))}
x ← and((v,N))   ⇒   x←and((w,M&N))
```


Distributive folding:

(M, N, P constants)

```
{v=or((u,M))}        t←and((u,N))
x ← and((v,N))   ⇒   x←or((t,M&N))
```

(t new)

```
(u|M)&N  =  (u&N) | (M&N)
```

```
{v=or((u,M)),
 w=and((v,N))}       t←and((u,N))
x ← or((w,P))    ⇒   x←or((w,(M&N)|P))
```

```
((u|M)&N)|P  =  (u&N) | ((M&N)|P)
```


Known closure:

```
{f = k{x}}
v ← f @ y   ⇒   v ← t
```

if `k{x} y = t`

Known thunk:

```
{t = m[x]}
v ← invoke t   ⇒   v ← m(x)
```

Known constructor:

```
{v = C(x)}
case v of      ⇒   b(x,y)
  C(x) ⟶ b(x,y)
  ...
```

Derived blocks - trailing enter:

$v \leftarrow b(x);\ v\ @\ a$ ⇒ $b1(x,a)$

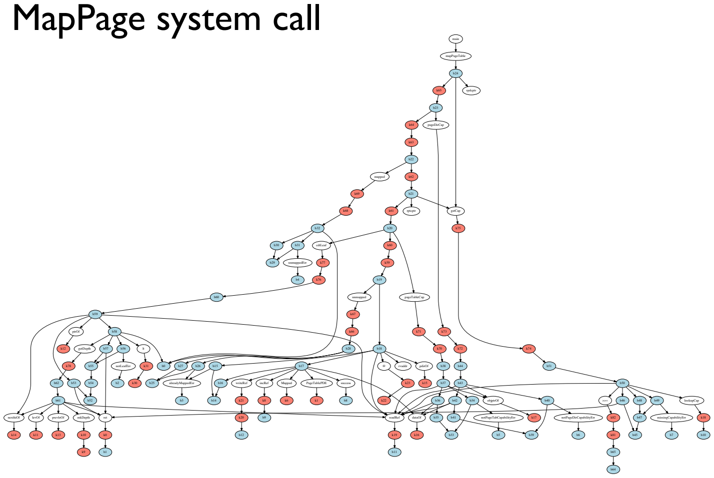
$$b(x) = \begin{array}{l} v_1 \leftarrow t_1 \\ \ldots \\ v_n \leftarrow t_n \\ t \end{array} \quad\Rightarrow\quad \begin{array}{l} b1(x,a) \\ \quad = v_1 \leftarrow t_1 \\ \qquad \ldots \\ \qquad v_n \leftarrow t_n \\ \qquad v \leftarrow t \\ \qquad v\ @\ a \end{array}$$

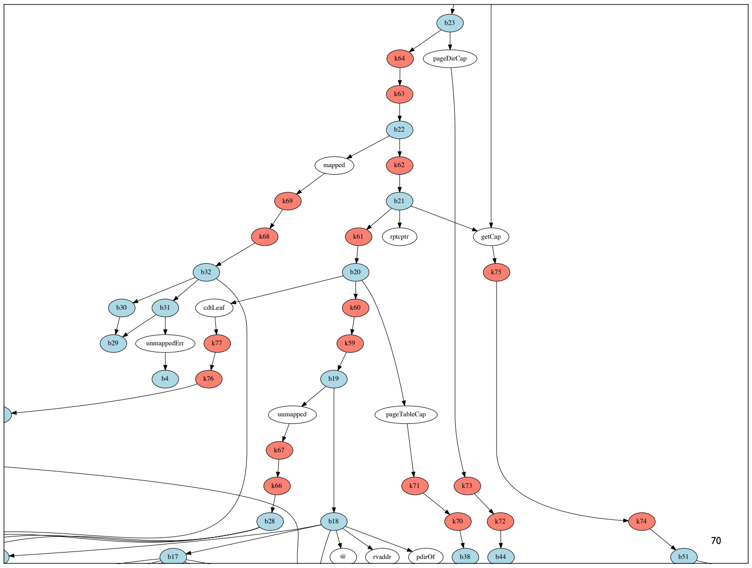
Derived blocks - known constructors:

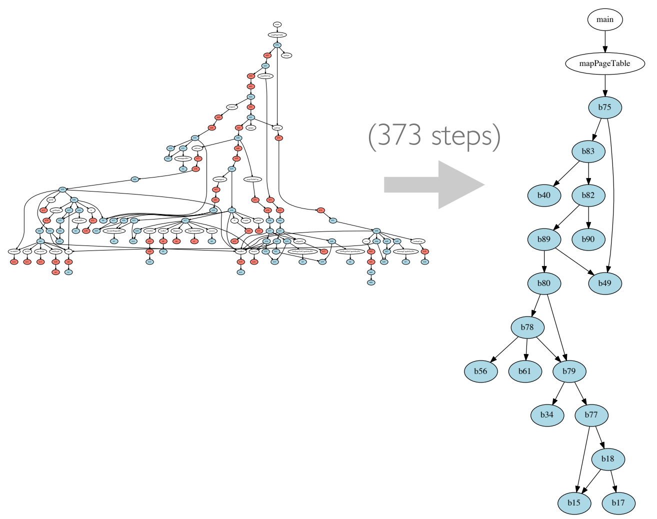
$\{u=C(p),\ f=k\{x\}\}$
$b(u,f,r)$ ⇒ $b1(p,x,r)$

$$b(u,f,r) = c \quad\Rightarrow\quad \begin{array}{l} b1(p,x,r) \\ \quad = u \leftarrow C(p) \\ \qquad f \leftarrow k\{x\} \\ \qquad c \end{array}$$

MapPage system call



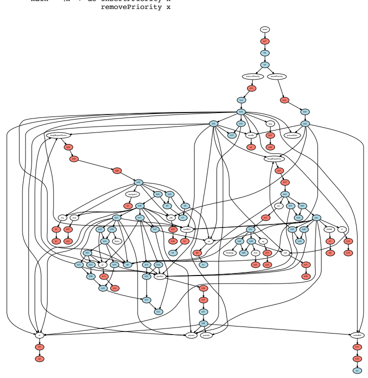




Prioset

```
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
```

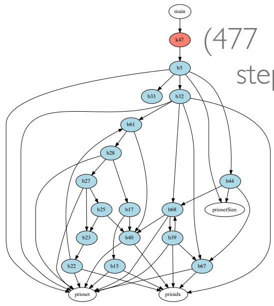
```
main = \x -> do insertPriority x
                removePriority x
```



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
```

```
main = \x -> do insertPriority x
                removePriority x
```

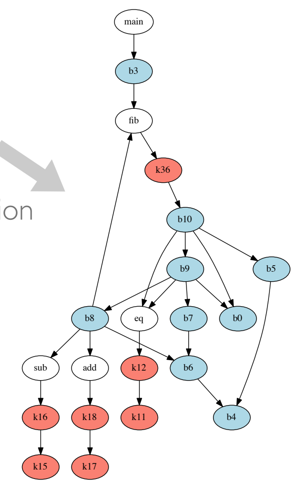


(477 steps)

Fibonacci

```
fib = \n -> if eq n 0 then 0
            else if eq n 1 then 1
            else add (fib (sub n 1))
                     (fib (sub n 2))
main = fib 12
```

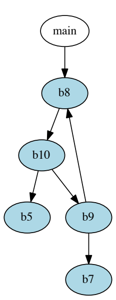
initial compilation



Fibonacci

```
fib = \n -> if eq n 0 then 0
            else if eq n 1 then 1
            else add (fib (sub n 1))
                     (fib (sub n 2))
main = fib 12
```

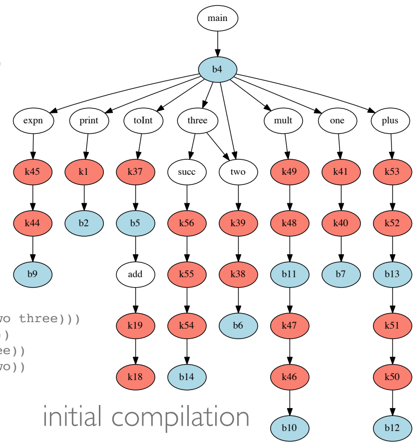
47 transformation steps later ...




```
b10(n) =                     b8(n) =
  t39 <- eq((n, 0))            t47 <- add((n, -1))
  case t39 of                  t48 <- b10(t47)
    True() -> b5()             t52 <- add((n, -2))
    False() -> b9(n)           t53 <- b10(t52)
                               add((t48, t53))
b9(n) =
  t43 <- eq((n, 1))          b7() = return 1
  case t43 of                b5() = return 0
    True() -> b7()
    False() -> b8(n)         main <-  b8(12)
```

Church

```
succ = \n f x -> n f (f x)
plus = \n m -> \f x -> n f (m f x)
mult = \n m -> \f x -> n (m f) x
expn = \n m -> m n


zero = \f x -> x
one  = \f x -> f x
two  = \f x -> f (f x)
three = succ two


toInt = \n -> n (add 1) 0


main
  = do print (toInt (plus one
                          (mult two three)))
       print (toInt (mult one two))
       print (toInt (expn two three))
       print (toInt (expn three two))
```

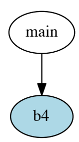


initial compilation

Church

```
succ = \n f x -> n f (f x)
plus = \n m -> \f x -> n f (m f x)
mult = \n m -> \f x -> n (m f) x
expn = \n m -> m n


zero = \f x -> x
one  = \f x -> f x
two  = \f x -> f (f x)
three = succ two


toInt = \n -> n (add 1) 0


main
  = do print (toInt (plus one
                          (mult two three)))
       print (toInt (mult one two))
       print (toInt (expn two three))
       print (toInt (expn three two))
```

```
main <- b4[]

b4() =
   _ <- print((7))
   _ <- print((2))
   _ <- print((8))
   print((9))
```



570 transformation steps later ...

PREPRINT: To be presented at ICFP '14, September 1–6, 2014, Gothenburg, Sweden.

Practical and Effective Higher-Order Optimizations

Lars Bergstrom
Mozilla Research *
larsberg@mozilla.com

Matthew Fluet
Matthew Le
Rochester Institute of Technology
{mtf,ml9951}@cs.rit.edu

John Reppy
Nora Sandler
University of Chicago
{jhr,nlsandler}@cs.uchicago.edu

Abstract

Inlining is an optimization that replaces a call to a function with that function's body. This optimization not only reduces the overhead of a function call, but can expose additional optimization opportunities to the compiler, such as removing redundant operations or unused conditional branches. Another optimization, copy propagation, replaces a redundant copy of a still-live variable with the original. Copy propagation can reduce the total number of live variables, reducing register pressure and memory usage, and possibly eliminating redundant memory-to-memory copies. In practice, both of these optimizations are implemented in nearly every modern compiler.

These two optimizations are practical to implement and effective in first-order languages, but in languages with lexically-scoped first-class functions (aka, closures), these optimizations are not available to code programmed in a higher-order style. With higher-order functions, the analysis challenge has been that the environment at the call site must be the same as at the closure capture location, up to the free variables, or the meaning of the program may change. Olin Shivers' 1991 dissertation called this family of optimizations *Super-β* and he proposed one analysis technique, called *reflow*, to support these optimizations. Unfortunately, reflow has proven too expensive to implement in practice. Because these higher-order optimizations are not available in functional-language compilers, programmers studiously avoid uses of higher-order values that cannot be optimized (particularly in compiler benchmarks).

This paper provides the first practical and effective technique for Super-β (higher-order) inlining and copy propagation, which we call unchanged variable analysis. We show that this technique is practical by implementing it in the context of a real compiler for an ML-family language and showing that the required analyses have costs below 3% of the total compilation time. This technique's effectiveness is shown through a set of benchmarks and example programs, where this analysis exposes additional potential optimization sites.

* Portions of this work were performed while the author was at the University of Chicago.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.3.4 [*Programming Languages*]: Processors—Optimization

Keywords control-flow analysis, inlining, optimization

1. Introduction

All high level programming languages rely on compiler optimizations to transform a language that is convenient for software developers into one that runs efficiently on target hardware. Two such common compiler optimizations are copy propagation and function inlining. Copy propagation in a language like ML is a simple substitution. Given a program of the form:

```
let val x = y
in
   x*2+y
end
```

We want to propagate the definition of x to its uses, resulting in

```
let val x = y
in
   y*2+y
end
```

At this point, we can eliminate the now unused x, resulting in

```
y*2+y
```

This optimization can reduce the resource requirements (*i.e.*, register pressure) of a program, and it may open the possibility for further simplifications in later optimization phases.

Inlining replaces a lexically inferior application of a function with the body of that function by performing straightforward β-substitution. For example, given the program

```
let fun f x = 2*x
in
   f 3
end
```

inlining f and removing the unused definition results in

```
2*3
```

This optimization removes the cost of the function call and opens the possibility of further optimizations, such as constant folding. Inlining does require some care, however, since it can increase the program size, which can negatively affect the instruction cache performance, negating the benefits of eliminating call overhead. The importance of inlining for functional languages and techniques for providing predictable performance are well-covered in the context of GHC by Peyton Jones and Marlow [PM02].

Both copy propagation and function inlining have been well-studied and are widely implemented in modern compilers for both first-order and higher-order programming languages. In this paper, PREPRINT: To be presented at ICFP '14, September 1–6, 2014, Gothenburg, Sweden.

Practical and Effective Higher-Order Optimizations

Lars Bergstrom
Mozilla Research *
larsberg@mozilla.com

Matthew Fluet
Matthew Le
Rochester Institute of Technology
{mtf,ml9951}@cs.rit.edu

John Reppy
Nora Sandler
University of Chicago
{jhr,nlsandler}@cs.uchicago.edu

Abstract

Inlining is an optimization that replaces a call to a function with that function's body. This optimization not only reduces the overhead of a function call, but can expose additional optimization opportunities to the compiler, such as removing redundant operations or unused conditional branches. Another optimization, copy propagation, replaces a redundant copy of a still-live variable with the original. Copy propagation can reduce the total number of live variables, reducing register pressure and memory usage, and possibly eliminating redundant memory-to-memory copies. In practice, both of these optimizations are implemented in nearly every modern compiler.

These two optimizations are practical to implement and effective in first-order languages, but in languages with lexically-scoped first-class functions (aka, closures), these optimizations are not available to code programmed in a higher-order style. With higher-order functions, the analysis challenge has been that the environment at the call site must be the same as at the closure capture location, up to the free variables, or the meaning of the program may change. Olin Shivers' 1991 dissertation called this family of

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.3.4 [*Programming Languages*]: Processors—Optimization

Keywords control-flow analysis, inlining, optimization

1. Introduction

All high level programming languages rely on compiler optimizations to transform a language that is convenient for software developers into one that runs efficiently on target hardware. Two such common compiler optimizations are copy propagation and function inlining. Copy propagation in a language like ML is a simple substitution. Given a program of the form:

```
let val x = y
in
    x*2+y
end
```

We want to propagate the definition of x to its uses, resulting in

```
let val x = y
```


A "challenging example"

```
fun mk i =
  let
    fun g j = j + i
    fun f (h : int -> int, k)=
      (h (k * i))
  in
    (f, g)
  end
val (f1, g1) = mk 1
val (f2, g2) = mk 2
val res = f1 (g2, 3)
```

```
res <- return 4
```

```
fun mk i =
  let
    fun g j = j + i
    fun f (h : int -> int, k) =
      ((k * i) + i)
  in
val (f1, g1) = mk 1
val (f2, g2) = mk 2
val res = f1 (g2, 3)
```

"While this example is obviously contrived, this situation occurs regularly in idiomatic higher-order programs and the inability to handle the environment problem in general is a limit in most compilers, leading developers to avoid higher-order language features in performance-critical code."

An earlier example in the paper

```
let
  fun emit x = print (Int.toString x)
  fun fact i m k =
    if i=0 then k m
    else fact (i-1) (m*i) k
in
  fact 6 1 emit
end
```

```
b2() =
  t46 <- intToStr((720))
  print((t46))

main <-
  b2()
```

```
let
  fun emit x = print (Int.toString x)
  fun fact i m k =
    if i=0 then emit m
    else fact (i-1) (m*i) emit
in
  fact 6 1 emit
end
```

... generalized

```
emit x = print (intToStr x)

fact i m k
  = if i==0
      then k m
      else fact (i-1) (m*i) k

main m n = fact m n emit
```

```
b7(x) =
  t14 <- intToStr((x))
  print((t14))

b10(m, i) =
  t9 <- add((i, -1))
  t12 <- mul((m, i))
  t15 <- eq((t9, 0))
  case t15 of
    True -> b7(t12)
    False -> b10(t12, t9)

b8(i, m) =
  t5 <- eq((i, 0))
  case t5 of
    True -> b7(m)
    False -> b10(m, i)

k36{m} n = b8(m, n)

k37{} m = k36{m}

main <-
  k37{}
```

+ Third arg for fact has gone!

+ Direct call to (inlined) emit

- Some duplication of code ...

Very good optimization, for low effort?

- 50% chance that I just got lucky with my choice of examples ...
- 50% chance that this is a result of aggressive inlining and specialization
- 1% chance that there is something fundamentally new about this approach to optimization …
- 101% chance that I don't understand percentages :-)

Summary

- Back to 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 ...)