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 ...

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 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 ...

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 ]
  ```

- 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

Here is how we describe page directory entries in Habit:

```
bitdata PDE /WordSize
| = UnmappedPDE | unusedP0 :: Bit 11 | B0 :: Bit 0 | unused entry (present bit reset)
| + PageTablePDE | ptab :: Bit 4 | attr : readWrite :: PagingAttr | -- paging attributes
| + SuperPagePDE | super :: Bit 13 | attr : PagingAttr | -- present bit set

bitdata PagingAttr /6
| = PagingAttr | dirty :: Bit 0 | bit 1 | -- Dirty: 1 => data written to page
| | accessed :: Bit 0 | bit 1 | -- Accessed: 1 => page accessed
| | caching :: 0 | bit 1 | -- Caching
| | us :: 0 | bit 1 | -- User/supervision: 1 => user access allowed
| | rw :: 0 | 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 ...
    UnmappedPDE -> ... 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

19

First-class functions

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell</td>
<td>\x \rightarrow x + 1</td>
</tr>
<tr>
<td>LISP</td>
<td>\text{(lambda } (x) \text{ } (+ x 1))</td>
</tr>
<tr>
<td>Python</td>
<td>\text{lambda x: x + 1}</td>
</tr>
<tr>
<td>Javascript</td>
<td>\text{function } (x) \text{ } x + 1</td>
</tr>
<tr>
<td>C++ 11</td>
<td>[] \text{(int } x) \rightarrow \text{int} { \text{return } x + 1; }</td>
</tr>
<tr>
<td>Rust</td>
<td></td>
</tr>
</tbody>
</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

20

Simple examples

• The identity function:

  \text{id} = \text{\x \rightarrow x}

• The "successor" function

  \text{succ} = \text{\x \rightarrow x + 1}

• The "add" function

  \text{add} = \text{\x \rightarrow (y \rightarrow x + y)}

• The "compose" function

  \text{compose} = \text{\f \rightarrow \text{\g} \rightarrow \text{\x \rightarrow (g \circ f)(x)}}

21

Higher-order functions

• **Higher-order functions** are functions that take other functions as inputs or return functions as outputs

• \text{compose} and \text{map} are classic examples of higher-order functions:

  \text{map} = \text{\f} \text{xs} \rightarrow
  \text{case } \text{xs} \text{ of}
    \text{Nil} \rightarrow \text{Nil}
    \text{Cons } y \text{ ys} \rightarrow \text{Cons } (f \text{ y}) \text{ (map } f \text{ ys)}

• For example:

  \text{map } (\text{x \rightarrow x + 1}) \text{ [1,2,3,4]} = [2,3,4,5]
  \text{map } (\text{x \rightarrow 2 * x}) \text{ [1,2,3,4]} = [2,4,6,8]

• Good for capturing recurring patterns as reusable functions

22

First-class functions using classes

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

  \text{abstract class Func}\langle\text{A}, \text{B}\rangle \{ 
  \text{abstract B applyTo(A arg);} 
\}

  \text{class Id}\langle\text{A}\rangle \text{ extends Func}\langle\text{A}, \text{A}\rangle \{ 
  \text{A applyTo(A arg)} \{ \text{return arg}; \}
\}

  \text{class Succ extends Func}\langle\text{int}, \text{int}\rangle \{ 
  \text{int applyTo(int arg)} \{ \text{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"

23

First-class functions using classes, continued

• We can build closures for functions with multiple arguments:

  \text{class Add1 extends Func}\langle\text{int}, \text{int}\rangle \{ 
  \text{private int n;} 
  \text{new Add1(int n)} \{ \text{this.n = n; } \}
  \text{int applyTo(int arg)} \{ \text{return arg + n}; \}
\}

  \text{class Add extends Func}\langle\text{int}, \text{int}\rangle \{ 
  \text{Func}\langle\text{int}, \text{int}\rangle \text{ applyTo(int arg)} \{ \text{return new Add1(n)}; \}
\}

• Sample use:

  \text{new Add()}.applyTo(1).applyTo(2) \implies \text{returns 3}

• A single class can have many methods, which might require multiple functions

• But the verbose notation can discourage users ...

24
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 \( a \rightarrow b \) 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 \( \text{Proc} \ a \) 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:

  \[
  \begin{align*}
  A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow R \\
  A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow \text{Proc} \ R
  \end{align*}
  \]

  - A typical C prototype for a function like this:
    \[ R \ f(A_1 \ \text{arg}_1, \ A_2 \ \text{arg}_2, \ldots) \]  

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:
   - \( \text{Proc} \ a \) for regular procedures.
   - \( \text{Init} \ a \) 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.

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?

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

Chipping away ...

A small case study:
The Multiboot Information Structure

HaL4: A Capability-Enhanced Microkernel Implemented in Habit

based on Haskell
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) into memory
- GRUB searches the disk for a configuration file and interprets the commands there to load a full featured OS into memory
- The OS configures itself using information passed in from GRUB via a "Multiboot Information Structure"

The Multiboot Information Structure

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 (0th) 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, in C

```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;
};
```

Mark end of list with a special value
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:
  \[ \text{next} : \text{AsciiZ} \rightarrow \text{Proc} (\text{Maybe} (\text{Char}, \text{AsciiZ})) \]

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

Notes

• The \text{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 ...

The compiled version of putStr

putStr <- k54()
k54() t64 = 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))
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

foldCursor :: (ItemRef -> Word -> Proc Word)
foldCursor i a = fnmap (add a) (itemData i)

foldCursor i a = if c
  then foldCursor c a
  else return a

foldCursor i a = if c
  then foldCursor c a
  else return a

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

Things to ignore: everything else!
Another example: CursorSum in Habit

```haskell
main <- k59()
k59{}[] = b95{}
b95{}[] =
t618 <- getCursor({})
t619 <- Cursor 0 t618
t620 <- Cursor 1 t618
t621 <- primGet(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 <- primGet((t613, 0))
if t617
  then b120[t614, t613, t616]
  else b121[

b121[] = return 0
```

Another Case Study: System Call Validators

Using types ...

HaL4: A Capability-Enhanced Microkernel Implemented in Habit

Using lambda ...

HaL4: A Capability-Enhanced Microkernel Implemented in Habit

Using lambda ...

HaL4: A Capability-Enhanced Microkernel Implemented in Habit

Using lambda ...

HaL4: A Capability-Enhanced Microkernel Implemented in Habit

based on Haskell

based on Haskell

based on Haskell

?
Using lambda ...

System Calls Interrupt Handlers Exception Handlers

Using lambda ...

System Calls Interrupt Handlers Exception Handlers

Shared (Kernel) State

Validate Parameters

Return with error code

Perform Action

Return with success code

Imperative Functional Programming

• Traditional sequential control flow

\[
\text{do } \ f \leftarrow \text{openFile "file.txt"} \\
\text{l}_1 \leftarrow \text{readLine } f \\
\text{l}_2 \leftarrow \text{readLine } f \\
\text{out } \left(\text{l}_1, \text{l}_2\right) \\
\text{closeFile } f
\]

• 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

\[
\text{openFile} :: \text{String} \to \text{Proc FileHandle}
\]

• Try:

\[
\text{openFile} :: \text{String} \\
\to (\text{ErrorCode} \to \text{Proc } a) \\
\to (\text{FileHandle} \to \text{Proc } a) \\
\to \text{Proc } a
\]

• It’s as if we’ve given \text{openFile} two return addresses: one to use when an error occurs, and one to use when the call is successful.

higher-order, or first-class functions
Programming with continuations

• Our original program using continuations:

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

• Could we do the same for `readLine`?

```
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:

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

• Looking better …

Programming with continuations

• Reformat:

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

• Not always applicable …

Programming with continuations

• Add an infix operator: $f \$ x = f x$

```haskell
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:

```haskell
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
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

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

**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 $ set slot.pdir (Ref pdir)
    set slot.count 1
    setCapdata pdcap MappedPD[asid]
    success curr
```
The MapPage System Call

The MapPage System Call

heapRepairDown l prio last
prioSet i prio
if gt prio lprio && gt prio rprio then
do rprio <- readRef (at prioset r)
else
prioSet l prio
prioSet i lprio
if gt lprio prio then
Nothing ->                            // i has no right child

case leq (add (mul 2 u) 2) last of      // Look for a right child
Nothing -> prioSet i prio                    // i has no children

let u = unsigned i  // <- ret (unsigned i)
heapRepairDown = \i prio last ->
nprio <- readRef (at prioset i)
heapRepairDown i rprio (modIx (sub s 2))
i <- readRef (at prioidx prio)
rprio <- readRef (at prioset (modIx (sub s 1)))
prioSet i prio
if lt pprio prio then
Nothing -> prioSet 0 prio

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

Current status

• For the three main questions for CEMaBS:
  • 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 ...)

Wrapping Up ...
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 Kieburstz
- 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

Front end → Back end

Lambda Case

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: \( k\{x_1, \ldots, x_n\} \)
  - code pointer: \( k \)
  - stored fields: \( x_1, \ldots, x_n \)
- 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: \( k\{x_1, \ldots, x_n\} \ a = t \)
  - The code in \( t \) describes the result that is produced when you enter the closure with argument \( a \)

From Functional Source Code ...

\[
\begin{align*}
\text{id} & = \lambda x \rightarrow x \\
\text{compose} & = \lambda f \ g \ x \rightarrow f \ (g \ x) \\
\text{map} & = \lambda f \ xs \rightarrow \\
& \quad \text{case} \ xs \ of \\
& \quad \text{Nil} \quad \rightarrow \text{Nil} \\
& \quad \text{Cons} \ y \ ys \quad \rightarrow \text{Cons} \ (f \ y) \ (\text{map} \ f \ ys)
\end{align*}
\]
... to MIL Programs

\[
\begin{align*}
\text{id} & \leftarrow k_0() \\
k_0(x) & = b_0(x) \\
b_0[x] & = \text{return } x \\
\text{map} & \leftarrow k_1() \\
k_1(f) & = k_2(f) \\
k_2(f, g) & = k_3(f, g, x) \\
b_1[f, g, x] & = g \circ x \\
k_3(f) & = k_4(f, g, x) \\
k_4(f) & = k_5(f) \\
b_2[f, xs] & = \text{case } xs \text{ of } \\
& \quad \text{Nil}() \rightarrow b_3() \\
& \quad \text{Cons}(y, ys) \rightarrow b_4[f, y, ys] \\
b_3[] & = \text{Nil}() \\
b_4[f, y, ys] & = z \leftarrow f \circ y \\
m & \leftarrow \text{map } @ f \\
zs & \leftarrow m \circ ys \\
\text{Cons}(z, zs)
\end{align*}
\]

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

... to Optimized MIL Programs

\[
\begin{align*}
\text{compose} & \leftarrow k_1() \\
k_1(false) & = k_2() \\
k_2(f, x) & = b_3(f, x) \\
b_1[f, x] & = \text{case } x \text{ of } \\
& \quad \text{Nil}() \rightarrow b_3[] \\
& \quad \text{Cons}(y, ys) \rightarrow b_4[f, y, ys] \\
b_3[] & = \text{Nil}() \\
b_4[f, y, ys] & = z \leftarrow f \circ y \\
m & \leftarrow \text{map } @ f \\
zs & \leftarrow m \circ ys \\
\text{Cons}(z, zs)
\end{align*}
\]

unknown function call

known function call

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