CS 410/510

Languages & Low-Level Programming

Mark P Jones
Portland State University

Spring 2016

Week 9: Language design

Language design

- We've spent eight weeks studying bare metal development and microkernel design and implementation

- Now we turn our attention to language design

- How do you design a new language?
- What are the general goals of language design?
- What are the specific goals for this domain?
- How do you evaluate a proposed design?

- There are many valid answers to these questions!

Designing Habit: a Functional Language for Bare-Metal Programming

High Assurance Systems Programming Project (HASP)
Department of Computer Science
Portland State University

HASP Members, Past & Present

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

Languages for bare metal programming

Programming Languages

Machine Language/Assembly Language

- Primitive, access to hardware, very powerful, verbose, unsafe, use with care!

C, C++, Ada, …

- Higher-level, more portable, access to hardware, unsafe, popular, use with care!

Haskell, ML, …

- Very high-level, functional, concise, promotes reuse, expressive types/interfaces, safe

Programming Languages

Machine Language/Assembly Language

Primitive, access to hardware, very powerful, verbose, unsafe, use with care!

C, C++, Ada, …

Higher-level, more portable, access to hardware, unsafe, popular, use with care!

Haskell, ML, …

Very high-level, functional, concise, promotes reuse, expressive types/interfaces, safe

Programming Languages

Machine Language/Assembly Language

Primitive, access to hardware, very powerful, verbose, unsafe, use with care!

C, C++, Ada, …

Higher-level, more portable, access to hardware, unsafe, popular, use with care!

Haskell, ML, …

Very high-level, functional, concise, promotes reuse, expressive types/interfaces, safe

Programming Languages

Machine Language/Assembly Language

Primitive, access to hardware, very powerful, verbose, unsafe, use with care!

C, C++, Ada, …

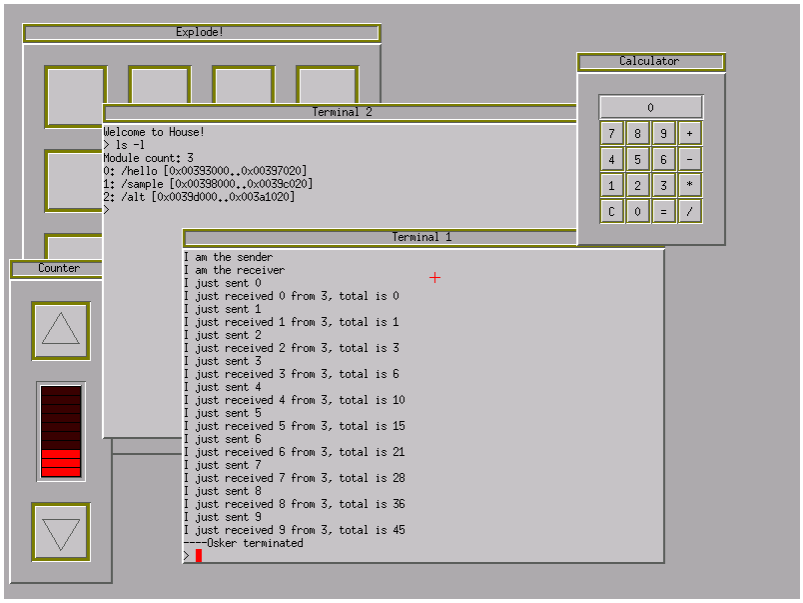
Higher-level, more portable, access to hardware, unsafe, popular, use with care!

Haskell, ML, …

Very high-level, functional, concise, promotes reuse, expressive types/interfaces, safe

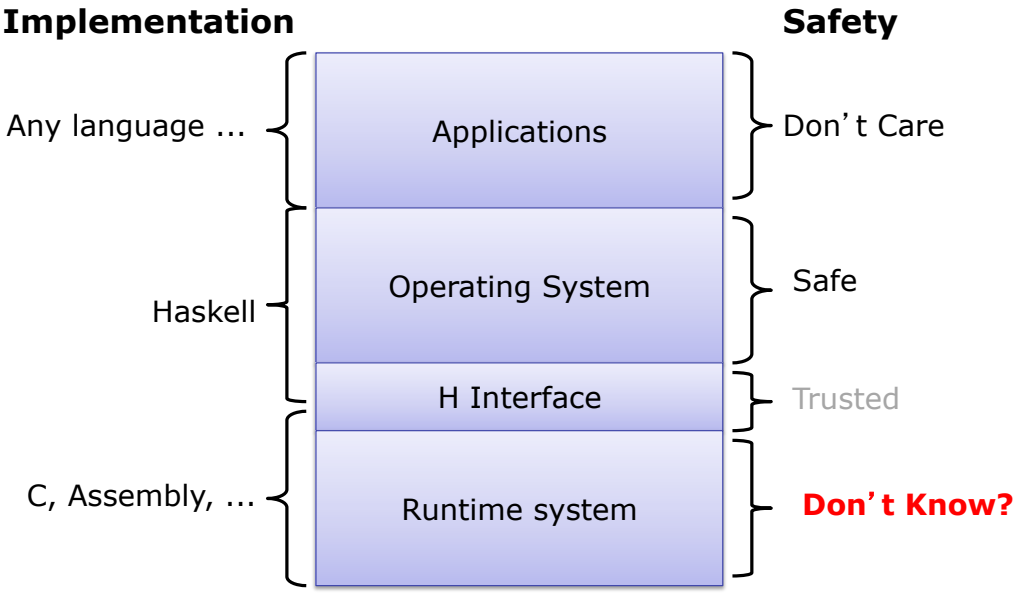
Some History: Experiments with bare metal programming in Haskell

Welcome to our House



(Building on hOp by Sébastian Carlier and Jérémy Bobbio, and Gadgets by Rob Noble)

House architecture



Managing pages in House

```
zeroPage p = sequence_ [poke ((castPtr p) `plusPtr` i) (0::Word8) | i <- [0..pageSize]]
```

It all type checks, it all runs, it must be right ...

Managing pages in House

Unsafe

Unsafe

Gratuitously inefficient

```
zeroPage p
  = sequence_ [poke ((castPtr p) `plusPtr` i)(0::Word8)
              | i <- [0..pageSize]]
```

Gratuitously inefficient

Wrong

type checks, it all runs, it must
ht ...

Separation in House

- There are no cross thread/process communication system calls in House
- So every process runs in perfect isolation from all of the others, right?
- Alas no, subject to DOS attacks because of the shared Haskell heap

Summary of issues raised

- Large, untrusted runtime system
- Reliance on unsafe operations for essential low-level primitives
- Weak type system
- Resource management issues
- Performance concerns

The seL4 Project

seL4: A formally correct OS

- A security enhanced version of the L4 microkernel, developed at NICTA/UNSW
- A C implementation that provably implements its Isabelle specification
- Main corollaries:
  - No buffer overflows
  - No null pointer dereferences
  - No pointer type errors
  - No memory leaks
  - No division by zero

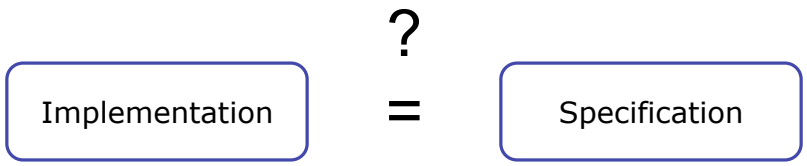
Details of overall effort

- 8,700 lines of C (7,500 lines verified) + 600 lines (unverified) ARM assembly code
- 200,000 lines of Isabelle proof script
- Relies on assumptions about compiler, assembly, hardware management, virtual memory, boot code
- Cost: 25-30 person years (estimate < 10 to repeat)
- A landmark accomplishment in formal verification
- But there are concerns about scalability, and the costs of maintaining proof scripts to track evolving code

Scalability and reuse

- Can we achieve similar results by focusing on general, reusable tools instead of a single application?
- Key idea: instead of

  verifying correctness between an implementation and a specification



Scalability and reuse

- Can we achieve similar results by focusing on general, reusable tools instead of a single application?

- Key idea: why not

  write the implementation in an executable specification language that <u>automatically</u> enforces key correctness properties?

Implementation $\hat{=}$ Specification

seL4: A formally correct OS

- A security enhanced version of the L4 microkernel, developed at NICTA/UNSW

- A C implementation that provably implements its Isabelle specification

- Main corollaries:
  - No buffer overflows
  - No null pointer dereferences
  - No pointer type errors
  - No memory leaks
  - No division by zero

  You get many of these properties for free by using a type/memory safe language ...

Challenges

- Can you get the low-level control over representation that you need in a high-level language?
- Can you deliver the predictable performance that is required in a typical systems programming context?

The HASP project

High
Assurance
Systems
Programming

Wiring 101

- Take Care!
- Avoid Shorts!
- Match Voltages!
- Follow Color Codes!

Plug and Play

- Simple, fast connections
- Enforce correct usage
- Guarantee safety
- Higher-level interfaces

Can you trust this?

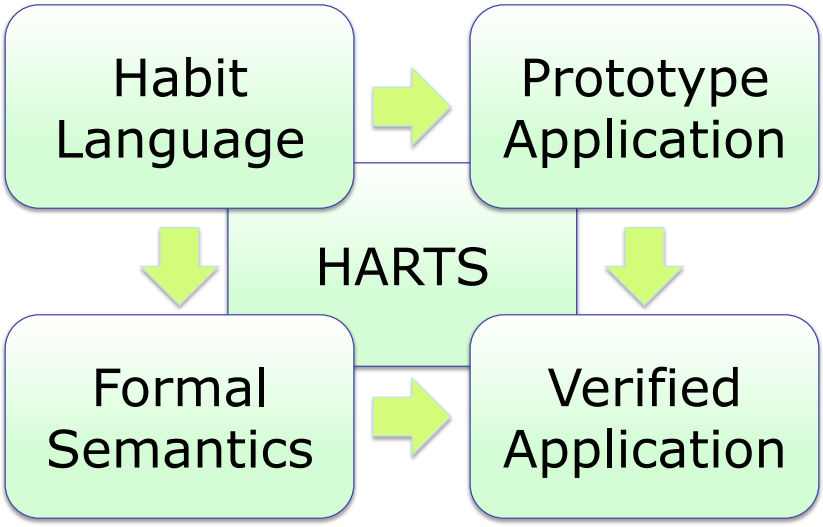
Does it work?

Is it safe?

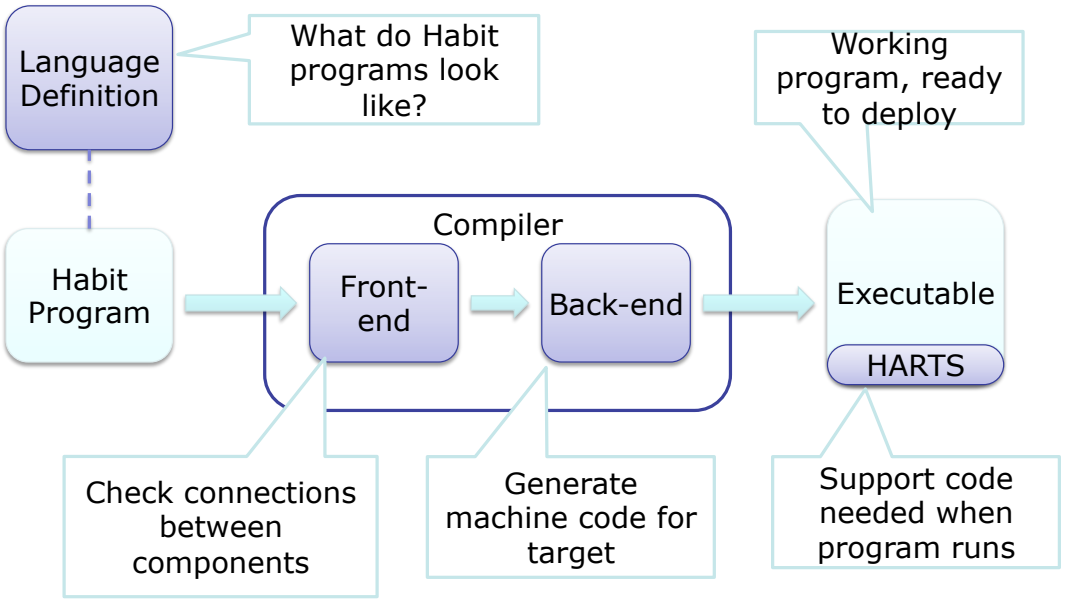
Is it certified/tested/verified?

What does that even mean?

HASP project vision



Habit infrastructure



A Habit for Bare-Metal Programming

Habit

Originally called
"Systems Haskell"

- A dialect of Haskell that is designed to meet the needs of high assurance systems programming

Habit

Haskell + **bit**s
High **a**ssurance + **bit**s

- A dialect of Haskell that is designed to meet the needs of high assurance systems programming

Habit

Purity and
Higher Orders

- A dialect of Haskell that is designed to meet the needs of high assurance systems programming

Habit

An excellent source of puns …

- A dialect of Haskell that is designed to meet the needs of high assurance systems programming

Habit

- A dialect of Haskell that is designed to meet the needs of high assurance systems programming
- Primary Commitments:
  - Systems Programming
  - Trading Control and Abstraction
  - High Assurance

Habit

- A dialect of Haskell that is designed to meet the needs of high assurance systems programming
- Systems (Bare Metal) Programming:
  - Standalone embedded applications
  - Operating systems, microkernels, device drivers, …
  - Not a general purpose programming language

Habit

- A dialect of Haskell that is designed to meet the needs of high assurance systems programming
- Provide programmers with the ability to choose and make informed trade-offs between:
  - Control over data representation and performance
  - Abstraction and use of higher-level language mechanisms

Habit

- A dialect of Haskell that is designed to meet the needs of high assurance systems programming
- High Assurance: a full and formal semantics that provides a basis for:
  - Mechanized reasoning
  - Meaningful assurance arguments
  - Verification of Habit programs and implementations

Habit

- A dialect of Haskell that is designed to meet the needs of high assurance systems programming
- High Assurance Runtime System (HARTS):
  - Services for memory management, garbage collection, foreign function interface, …
  - Designed to be "as simple as possible", modular, formally verified

Habit

- A dialect of Haskell that is designed to meet the needs of high assurance systems programming
  - Productivity: higher-level abstractions, genericity, reuse
  - Safety: built-in type and memory safety guarantees
  - Tractability: purity, referential transparency, encapsulation of effects, semantic foundations

Habit

- A dialect of Haskell that is designed to meet the needs of high assurance systems programming
  - Increasing interest & adoption
  - Strong community
  - Usability (the barrier to entry is already high)
  - Avoid reinventing the wheel:
    - Syntax: familiar notations and concepts
    - Semantics: powerful, expressive type system

Habit

- A dialect of Haskell that is designed to meet the needs of high assurance systems programming
- Aspects of the "House" experience to avoid:
  - Low level features via unsafe interfaces
  - Unpredictable performance
  - Large, feature rich runtime system
  - Abstraction from resource management

Design influences

General areas/ application domains
- Operating systems
- Microkernels
- VMMs
- Hypervisors
- Device drivers

Languages
- Haskell, ML, …
- C, C++, assembler, …

Previous PSU/OGI work
- House, H, L4, pork
- Bitdata and memory areas

Previous Galois work
- TSE, especially the Block Access Controller (BAC)
- Haskell file system
- HaLVM
- Cryptol
- AIM debugger

Requirements

- Representation/Control
  - Code: optimization, implementation
  - Data: layout, initialization, conversion
- Ease of use
  - Notation, type inference, user-defined control structures
- Verification
  - Semantic foundations, type and memory safety

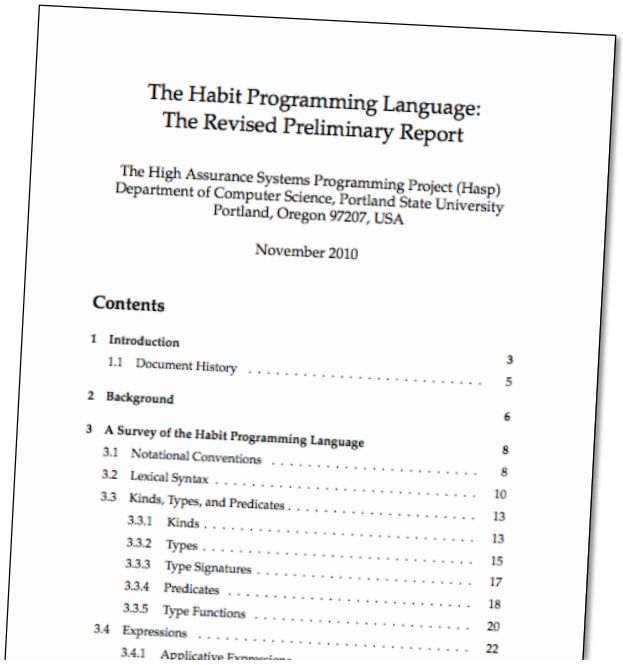
The Habit Language Report

November 2010:

- Revised Preliminary Report (95 pages)
- Available from D2L reference section

Today:

- A Quick Overview



The Habit Programming Language:
The Revised Preliminary Report

The High Assurance Systems Programming Project (Hasp)
Department of Computer Science, Portland State University
Portland, Oregon 97207, USA

November 2010

Contents

1 Introduction 3
1.1 Document History 5
2 Background 6
3 A Survey of the Habit Programming Language 8
3.1 Notational Conventions 8
3.2 Lexical Syntax 10
3.3 Kinds, Types, and Predicates 13
3.3.1 Kinds 13
3.3.2 Types 15
3.3.3 Type Signatures 17
3.3.4 Predicates 18
3.3.5 Type Functions 20
3.4 Expressions 22
3.4.1 Applicative Expressions

Habit design: summary

- "Simplified" "dialect" of Haskell
  - Foundations: pure, higher-order, typed
  - Syntax: definitional style, lightweight notation
- Omitted features
  - Module system (at least for now); fancy patterns; misc. syntactic sugar; strictness annotations; newtype; …
- Changes/additions
  - Strict evaluation; bitdata; memory areas; type-level numbers; functional dependencies & notation; instance chains; unpointed types; monadic sugar; …

When unsafe is the only option

- Some low-level features are inherently unsafe
- In these cases, we strive:
  - To wrap them in safe interfaces (A key aspect in the design of House)
  - To ensure that they are easily located and identified for audit purposes

Types in Habit

Types in Habit

The Habit type system is used:

- to document behavior
- to ensure correct usage
- to detect errors
- to guide code generation
- to support assurance

Types in Habit

Not a fundamentally new type system:

New Syntax: bitdata, structure, and memory area declarations
New Primitives: kinds, predicates, types, functions
Established Foundation: Haskell style type system (kinds, polymorphism, type classes)

Example: kinds in Haskell

Haskell uses kinds to classify types

*	standard types: Unsigned, Bool, etc…
$k_1 \rightarrow k_2$	parameterized type constructors

Example: kinds in Habit

Habit builds on this foundation:

*	standard types: Unsigned, Bool, etc…
$k_1 \rightarrow k_2$	parameterized type constructors
nat	type-level natural numbers
area	layout of data blocks in memory

Type-level naturals (kind nat)

Natural numbers as components of types:

- Array bounds, bit vector widths, alignments, literals, memory areas sizes, etc…
- Examples: Bit 3, Ix 256, ARef 4K a, …
- Simple syntax, efficient type inference (avoids encodings used in some Haskell libraries)
- Weaker than full dependent types; retains tractability

Memory areas (kind area)

Layouts for blocks of memory:

Stored t	In memory representation for values of type t (limited domain)
Array n a	An array comprising n elements of type a
Pad n a	Padding n elements of type a
structures	In memory records/structures

Type classes and type functions

- Ad-hoc polymorphism:

```
(+) :: Num a => a -> a -> a
```

- Type functions (e.g., arithmetic on type-level natural numbers):

```
(#) :: Bit n -> Bit m -> Bit (n+m)
```


Conventional FP

```
data List a            = Nil | Cons a (List a)
data Maybe a           = Nothing | Just a

map                   :: (a -> b) -> List a -> List b
map f Nil              = Nil
map f (Cons x xs)      = Cons (f x) (map f xs)

foldr                 :: (a -> b -> b) -> b -> List a -> b
foldr f a Nil          = a
foldr f a (Cons x xs) = f x (foldr f a xs)
```

Conventional FP

```
inside            :: Fpage -> Fpage -> Bool
fp1 `inside` fp2   = (fpageStart fp1 <= fpageStart fp2)
                     && (fpageEnd fp2 <= fpageEnd fp1)

overlaps          :: Fpage -> Fpage -> Bool
fp1 `overlaps` fp2 = not (fp1 `disjoint` fp2)

disjoint          :: Fpage -> Fpage -> Bool
fp1` disjoint` fp2 = (fp1 `before` fp2) || (fp2 `before` fp1)

before            :: Fpage -> Fpage -> Bool
fp1 `before` fp2   =  fpageEnd fp1 < fpageStart fp2
```


Dogmatic design?

- In Haskell, Strings are represented by lists of characters, String = [Char]
- In Habit, there are no strings
  - There are no (built-in) lists
  - There are no characters
- Reinforces choice of target domain
- (Our implementation isn't so dogmatic)

Leveraging (Abstract) Types

Leveraging types

Fine-grained control over:

- representation
- layout
- alignment

Safety/correctness

- no out of bounds array accesses
- no out of range numeric literals
- no unchecked division by zero

Scoping of effects

- access to state, privileged operations, …
- documenting & enforcing correct usage
- ensuring correct initialization

Controlling representation

```
bitdata Bool  = False [ B0 ] | True [ B1 ]
bitdata Perms = Perms [ r, w, x :: Bool ]
bitdata Fpage
       = Fpage [ base :: Bit 22 | size :: Bit 6
               | reserved :: Bit 1 | perms :: Perms ]
```

Bit-level data specifications

Type-level numbers

Mimics familiar box layout notation

$base_{22}$	$size_6$	~	r	w	x

Compare with C

```
class mempage_t {
public:
    union {
        struct {
            BITFIELD7(word_t,
                      execute          : 1,
                      write            : 1,
                      read             : 1,
                      reserved         : 1,
                      size             : 6,
                      base             : L4_FPAGE_BASE_BITS,
                      : BITS_WORD - L4_FPAGE_BASE_BITS - 10
                );
        } x __attribute__((packed));
        word_t raw;
    };
};
```

BITFIELD macro adjusts for variations between C compilers …

From L4Ka::Pistachio, a mature L4 implementation in C++ from the University of Karlsruhe, Germany

Permission values inlined

Macro-level numbers

gcc specific attribute: "a variable or structure field should have the smallest possible alignment"

Compare with seL4 solution

```
block pte {
    field_high  page_base_address   20
    field       avl                 3
    field       global              1
    field       pat                 1
    field       dirty               1
    field       accessed            1
    field       cache_disabled      1
    field       write_through       1
    field       super_user          1
    field       read_write          1
    field       present             1
}
```

One example from structures.bf in the seL4 sources

Simple domain specific language for describing bit-level layout

python code parses .bf file and generates C code for manipulating data structures

(More) conventional FP

```
fpageStart        :: Fpage -> Unsigned
fpageStart fp      = (fp.base # 0) .&. not (fpageMask fp)

fpageEnd          :: Fpage -> Unsigned
fpageEnd fp        = (fp.base # 0) .|. fpageMask fp

fpageMask         :: Fpage -> Unsigned
fpageMask fp       = fpmask fp.size

fpmask            :: Bit 6 -> Unsigned
fpmask n
  | n==1 || n==32 = not 0
  | n<12 || n>32  = 0
  | otherwise     = (1 << n) - 1
```

shift inferred from type

Could be compiled as a lookup table …

Controlling memory layout

In C:

```
struct Mapping {
  struct Space*   space;
  struct Mapping* next;
  struct Mapping* prev;
  unsigned        level;
  Fpage           vfp;
  unsigned        phys;
  struct Mapping* left;
  struct Mapping* right;
};
```

Exact layout is not guaranteed without compiler-specific annotations

In Habit:

```
struct Mapping/32
  [ space       :: Stored (Ptr Space)
  | next, prev  :: Stored (Ptr Mapping)
  | level       :: Stored (Bit 32)
  | vfp         :: Stored Fpage
  | phys        :: Stored (Bit 32)
  | left, right :: Stored (Ptr Mapping) ]
```

Fine-control of memory layout, endianness, etc… No arbitrary pointer arithmetic

Area alignment & allocation

Allocating an initial page directory:

- In C/C++ (from L4Ka::Pistachio):

```
static word_t init_pdir[1024]
        __attribute__((aligned(4096)))
        SECTION(".init.data");
```

- In assembly code (from pork):

```
            .align  (1<<PAGESIZE)
init_pdir:  .space  4096
```

- In Habit:

```
area init_pdir :: ARef 4K (Array 1024 PDE)
```

Strong typing of array elements

Example: arrays

- The type `Ix n` contains only in-bound indices for an array of length `n`
- Array lookup can be fast (no bounds check) and safe: `(@) :: Ref (Array n t) -> Ix n -> Ref t`
- Amortized construction of safe indices with comparisons that are already required

```
(<=?) :: Unsigned -> Ix n -> Maybe (Ix n)
incIx :: Ix n -> Maybe (Ix n)
decIx :: Ix n -> Maybe (Ix n)
```


Example: numeric literals

- It is convenient to allow `0` to be used as a value of many types: `Bit n`, `Ix n`, `Unsigned`, …
- Haskell: interpret as value of type `Num a => a`
  - Requires bignum arithmetic at run-time
  - Does not test validity (e.g., `5` is not a valid `Bit 2` or `Ix 3`)
- Habit: introduce a class `Lit n t`, indicating `n` is a valid literal of type `t`
  - Requires bignum arithmetic at compile-time
  - `0 :: NumLit 0 t => t` can be used at expected types
  - `5 :: NumLit 5 t => t` rejects invalid uses

Example: division

- Division has type: `t -> NonZero t -> t`
- Only two ways to construct a `NonZero t` value:
  - Runtime check (cost can be amortized):

    ```
    nonZero :: t -> Maybe (NonZero t)
    ```

  - Literal divisor checked at compile-time:

    ```
    instance (Lit n t, 0<n) => Lit n (NonZero t)
    ```

- Simple, safe, low-cost, generic
- (But what is: MININT / (-1) ?)

Refining types with control flow

- Compare:

```
if n == 0
     then ...
     else ...
```

The Habit type system is unable to exploit the fact that `n != 0` here

- With:

```
case nonZero n of
     Nothing -> ...
     Just m   -> ...
```

But we can be sure that `m` is non zero here

Example: purity

- The output of a function of type `A -> B` depends only on the value of its input
- No hidden dependence on global variables or privileged state
- Explicit data flow; simplified reasoning
- The features that we omit can sometimes be as important as the features that we include

Example: initialization

- How to ensure deterministic initialization of memory areas/ global data?
- Answer: The abstract type `Init a`, with a family of operations for constructing initializers:

```
initArray :: (Ix n -> Init a) -> Init (Array n a)
```

- Initializers specified in memory area declarations:

```
area name <- init :: type
```

- Operations for the `Init` type can write (initialize), but not read, or perform side effects; execution of initializers, in any order, produces deterministic effect

Imperative Programming

Imperative programming

- Traditional systems programming typically has a very "imperative feel"
- In Haskell, "imperative feel" = monads
- Back in the days before monads:
  - Streams, Fudgets, Fran and FRP (functional reactive programming), ...
- Are there better, more "functional" abstractions for bare-metal programming?

Imperative programming

- Traditional systems programming typically has a very "imperative feel"
- In Habit, "imperative feel" = monads ☹
- But we hope to find inspiration for new, more functional abstractions as we gain more experience with Habit ...
- In the meantime, there are still benefits to be had ...

Imperative programming

A page fault handler, transcribed from pork:

```
pageFault :: K a
pageFault  = do curr      <- getCurrent
                faultCode <- getCR2
                writeRef curr.faultCode faultCode
                space <- readRef curr.space
                if space==sigma0Space then
                  sigma0map faultCode
                else
                  utcb    <- readRef curr.utcb
                  pagerId <- readRef utcb.tcrs.pager
                  deliverException PageFault curr pagerId
```

Looks mostly similar to C code in pork

But there is a program error here!

... which is found automatically by the type checker!

Monads / procedure types

Monads are a kind of abstract datatype that distinguish:

P V

Computational features — from — Values produced

Procedure types

- Typical C function prototype:

  $$\texttt{R f(T}_1\texttt{, T}_2\texttt{, …, T}_n\texttt{)}$$

- Corresponding Haskell type:

  $$\texttt{f :: T}_1 \rightarrow \texttt{T}_2 \rightarrow \ldots \rightarrow \texttt{T}_n \rightarrow \texttt{R}$$

  Pure

- Or, with effects:

  $$\texttt{f :: T}_1 \rightarrow \texttt{T}_2 \rightarrow \ldots \rightarrow \texttt{T}_n \rightarrow \texttt{IO R}$$

  Effectful

What is IO?

```
putChar   :: Char -> IO ()
getChar   :: IO Char
readFile  :: FilePath -> IO String
createDir :: FilePath -> IO ()
system    :: String -> IO ExitCode
getArgs   :: IO [String]
socket    :: Family -> SocketNo -> IO Socket
accept    :: Socket -> IO (Socket, SockAddr)
readRef   :: Ref a -> IO a
writeRef  :: Ref a -> a -> IO ()
poke      :: Storable a => Ptr a -> a -> IO ()
peek      :: Storable a => Ptr a -> IO a
```


IO, the “kitchen sink monad”

Console I/O	Port I/O
FileSystem I/O	Read Only
Environment I/O	Exceptions
Network I/O	Initialization
Mutable References	Interruptable
Memory Access	Sleep/Block
Database I/O	Allocation

Procedure types

- What does a type tell us about behavior?

$$R\ f(T_1,\ T_2,\ \ldots,\ T_n)$$

- Even in Haskell:

$$f :: T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow IO\ R$$

- A huge, missed opportunity

Particular relevance to the bare metal domain where some sections of code are required to run in special "modes"

Privileged? Blocking? Allocating?

Preemptable? Exceptions? Transactional?

Paging on? Protection on? Segmentation on?

- Correct usage traditionally depends on programmer discipline
- Can we use the type system to enforce correct usage at compile-time?

Goodbye kitchen sink!

```
putChar   :: Char -> IO ()
getChar   :: IO Char
readFile  :: FilePath -> IO String
createDir :: FilePath -> IO ()
system    :: String -> IO ExitCode
getArgs   :: IO [String]
socket    :: Family -> SocketNo -> IO Socket
accept    :: Socket -> IO (Socket, SockAddr)
readRef   :: Ref a -> IO a
writeRef  :: Ref a -> a -> IO
poke      :: Storable a => Ptr a -> a -> IO ()
peek      :: Storable a => Ptr a -> IO a
```


Focus on interfaces

```
class Console p
  where   putChar   :: Char -> p ()
          getChar   :: p Char

class File p
  where   readFile  :: String -> p String
          writeFile :: String -> String -> p String

class Sleep p
  where   sleep     :: MicroSecs -> p ()

class X86Priv p
  where   getCR3    :: p CR3

…
```

Scalable, flexible

constraints

variable

$$f :: P \Rightarrow T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow p\ R$$

Now types reveal a great deal:

```
          (File p) => Int → p Int
(Console p, File p) => Int → p Int
        (X86Priv p) => Int → p Int
                       Int → p Int
```


Monadic Sugar

Common patterns:

```
do b <- expr; if b then s1 else s2
do b <- expr; case b of alts
```

Monadic Sugar

Common patterns:

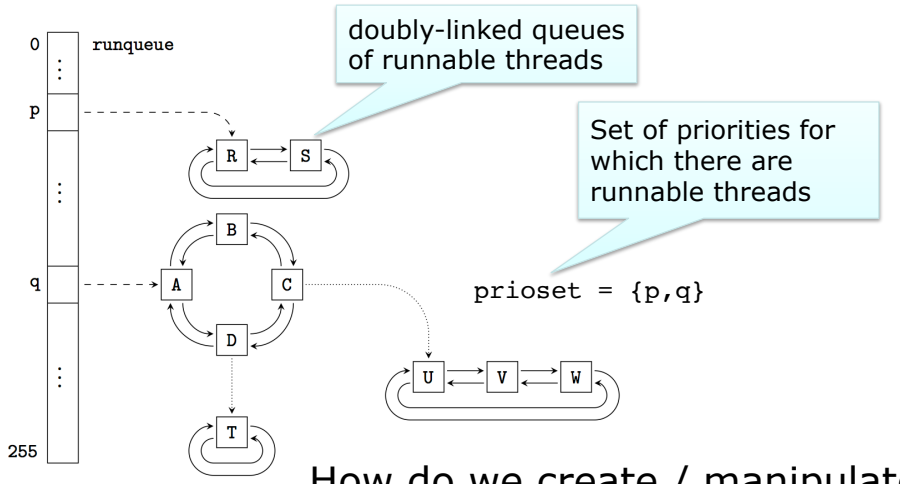
```
if<- expr then s1 else s2
case<- expr of alts
```

Example using `if<-` and `case<-`:

```
recvBlock :: IPCType -> Ref TCB -> K ()
recvBlock recvtype recv
           = if<- recvCanBlock recvtype recv
               then case<- get recv.status of
                      Runnable -> removeRunnable recv
                                  set recv.status Blocked
               else recvError NoPartner recvtype recv
```


The Priority Set Case Study

Imperative structures



How do we create / manipulate structures like these?

Example: timer interrupt handler

Code from the timer interrupt handler:

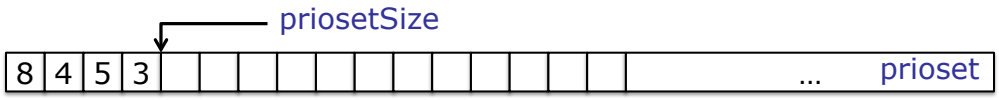
```
timerInterrupt
  = do ...
       ...
       case<- highestPriority of
         Just prio -> hprio <- readRef (holder.prio)
                      if prio > hprio then
                        reschedule
       resume
```

Preempt current timeslice holder with a higher priority thread?

Finding highest priority should be fast …

The priority set

Represent as a maximum heap structure




```
type NumPrio      = 256  -- Number of priority levels
type Priority     = Ix NumPrio

area prioset     :: Ref (Array NumPrio (Stored Priority))
area priosetSize :: Ref (Stored Unsigned)
```

name

alignment

layout

Finding highest runnable priority

Possibility of empty runqueue reflected in type

```
highestPriority :: M (Maybe Priority)
highestPriority  = do s <- readRef priosetSize
                      if s==0 then
                        return Nothing
                      else
                        prio <- readRef (prioset @ 0)
                        return (Just prio)
```

"do notation"

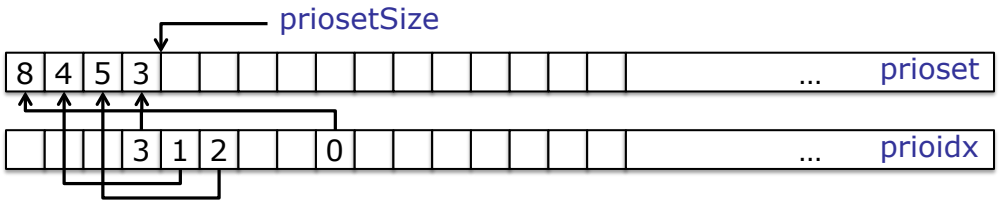
Valid array index, checked statically

Constant time execution ...

...or does this require allocation?

The priority set

Represent as a maximum heap structure




```
type NumPrio      = 256  -- Number of priority levels
type Priority     = Ix NumPrio

area prioset     :: Ref (Array NumPrio (Stored Priority))
area priosetSize :: Ref (Stored Unsigned)
area prioidx     :: Ref (Array NumPrio (Stored (Ix NumPrio)))
```


Updating the priority set

```
setPrio         :: Ix NumPrio -> Priority -> M ()
setPrio i prio  = do writeRef (prioset @ i) prio
                     writeRef (prioidx @ prio) i


insertPriority     :: Priority -> M ()
insertPriority prio = do s <- readRef priosetSize
                         writeRef priosetSize (s+1)
                         heapRepairUp (modIx s) prio
```

runtime check to ensure validity

Restore maximum heap

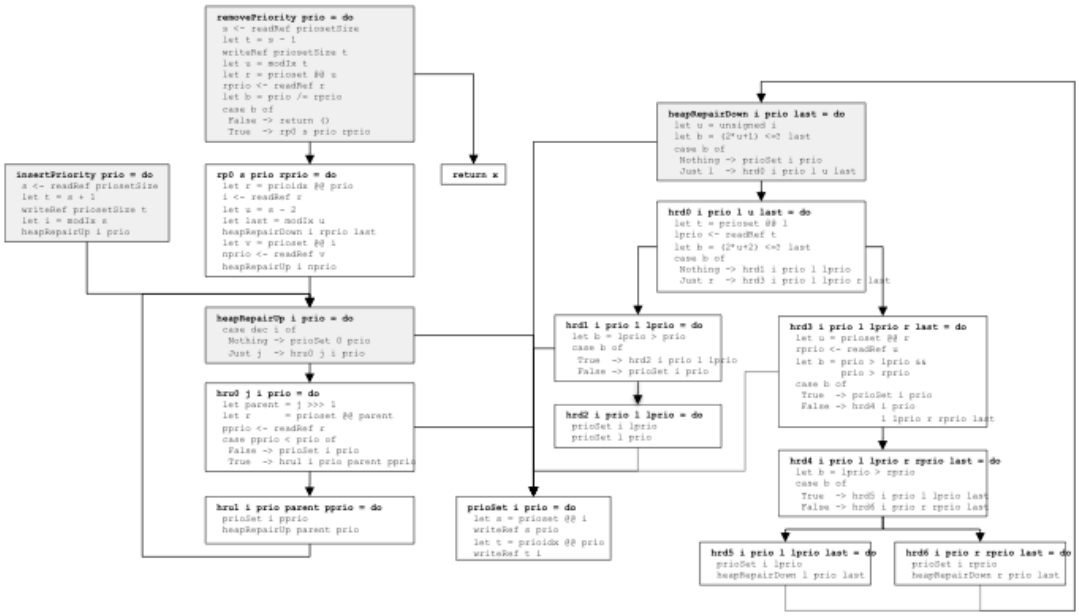
```
heapRepairUp :: Ix NumPrio -> Priority -> K ()
heapRepairUp i prio
 = case dec i of
      Nothing -> setPrio 0 prio  -- at the root
      Just j  -> do let parent = j>>1
                    pprio <- readRef (prioset @ parent)
                    if pprio < prio then
                      setPrio i pprio
                      heapRepairUp parent prio
                    else
                      setPrio i prio
```

Type safe manipulation of array indices

Syntactic sugar for monadic programming

"monadic tail recursion" can compile to simple loop

Can performance be addressed?



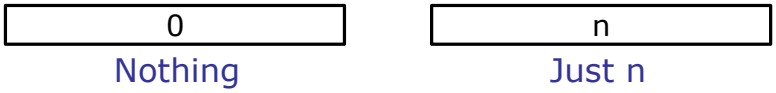
Implementation Overview

Compilation strategy

- Specialization eliminates polymorphism, type classes
  - Reduce runtime overhead
  - Type-specific/customized implementations
  - Based on experiments, code explosion is not expected to be a problem
- Whole program compilation
  - Enables whole program optimization
  - Appropriate for bare metal domain

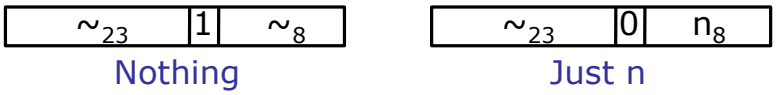
Type specific representations

- data Maybe a = Nothing | Just a
- For Maybe (NonZero t), we can use:



nonZero :: t -> Maybe (NonZero t) is now a nop

- For Maybe (Ix 256), we can use:



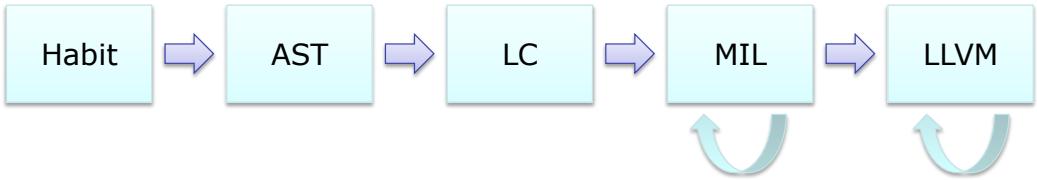
incIx :: Ix n -> Maybe (Ix n)
decIx :: Ix n -> Maybe (Ix n)

Implemented by simple inc/dec

Current status

- The HASP project built a working compiler pipeline, capable of running small test apps
  - Of course, it's (still) not feature complete ☺
  - No sense of performance yet
  - Hard to build and run … bit rot has already set in ☹
- Semantics-preserving back-end
  - Leverages portions of Leroy et al.'s CompCert certifying compiler for C
  - Written using the Coq proof assistant
  - Compiled code augmented by small RTS with verified collector
  - Targets include ARM & x86

A new back-end?



- Translate output from front end to a "monadic intermediate language", MIL, which is a good target for optimizations on functional language abstractions
- Translate MIL code to LLVM, which has strong support for imperative language optimizations, and can generate machine code for multiple architectures

Conclusions

Conclusions

- Goals for Habit:
  - build on successes in the design of Haskell
  - reflect requirements and feature set for the bare metal programming domain
  - leverage type system
  - provide foundations for formal verification
- How are we getting there?
  - careful selection of primitive kinds, classes types, and functions
  - focus on features for bare-metal programming

Still on the table ...

- Can we deliver acceptable performance?
- Can we guarantee correctness of our compiler?
- Resource management
  - Types? analysis? pragmas?
- New functional abstractions for bare metal?
- Concurrency: it's hard to ignore multicore ...