Developing Good Habits for Bare-Metal Programming

Mark P Jones, Iavor Diatchki (Galois),
Garrett Morris, Creighton Hogg, Justin Bailey
April 2010

Portland State University

Emphasis on Develop**ing**

- This is a talk about work in progress
- The language **design** is substantially complete (for now), but not all of the details have been written down, and some have not been tested in practice
- A prototype **implementation** is in progress, but it is substantially incomplete and lags the design

Habit

Formerly
"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, …

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
  - Avoid reinventing the wheel:
    - Syntax: familiar notations and concepts
    - Semantics: powerful, expressive type system

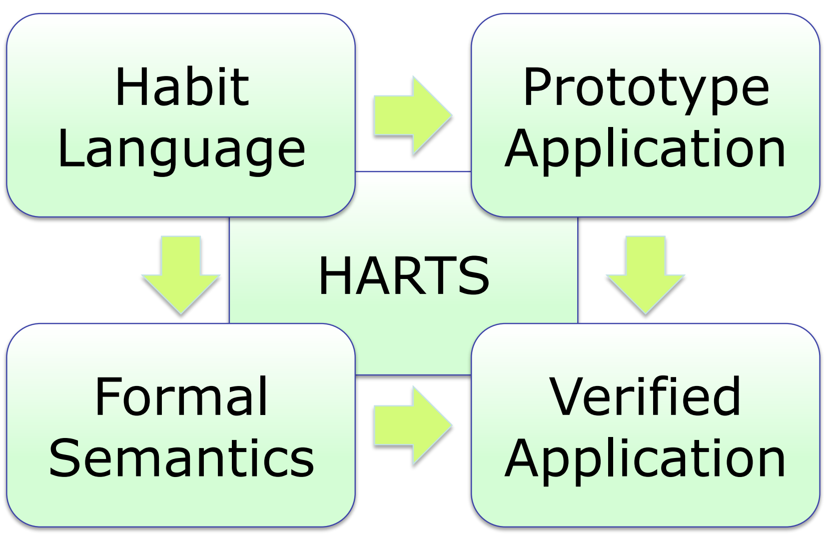
Habit

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

HASP Project Overview

High
Assurance
Systems
Programming

HASP Project Overview



Design Influences

General areas/application domains

- Operating systems
- Microkernels
- VMMs
- Hypervisors
- Device drivers

Languages

- Haskell, ML, BlueSpec, Erlang, Cryptol, …
- C, C++, Ada, assembler, …

Previous PSU/OGI work

- Programatica
- House, H, L4, pork
- Bitdata and memory areas (Hobbit)

Previous Galois work

- TSE, especially the Block Access Controller (BAC)
- Haskell file system
- HaLVM
- 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

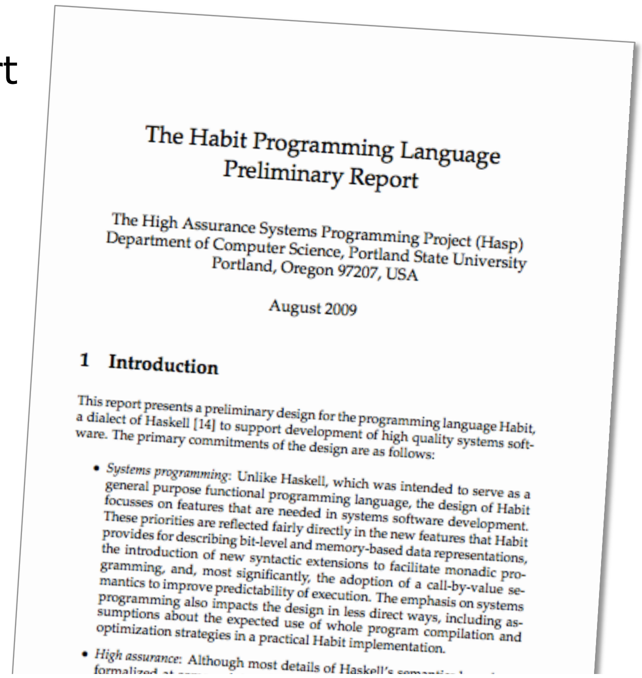
The Habit Language Report

Last Year:

- Preliminary Report (~70 pages)

Today:

- A Quick Overview



The Habit Programming Language
Preliminary Report

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

August 2009

1 Introduction

This report presents a preliminary design for the programming language Habit, a dialect of Haskell [14] to support development of high quality systems software. The primary commitments of the design are as follows:

- *Systems programming*: Unlike Haskell, which was intended to serve as a general purpose functional programming language, the design of Habit focusses on features that are needed in systems software development. These priorities are reflected fairly directly in the new features that Habit provides for describing bit-level and memory-based data representations, the introduction of new syntactic extensions to facilitate monadic programming, and, most significantly, the adoption of a call-by-value semantics to improve predictability of execution. The emphasis on systems programming also impacts the design in less direct ways, including assumptions about the expected use of whole program compilation and optimization strategies in a practical Habit implementation.
- *High assurance*: Although most details of Haskell's

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; …

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)
```

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
```

Controlling Representation

Bit-level data specifications

```
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 ]
```

Type-level numbers

Mimics familiar box layout notation

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

Types in Habit

Not a fundamentally new type system:

New Syntax: bitdata, structure, and memory area declarations
New Primitives: kinds, classes, 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…
- $k_1 \to k_2$ parameterized type constructors

Example: Kinds in Habit

Habit builds on this foundation

*	standard types: Unsigned, Bool, etc…
$k_1 \to k_2$	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, but surprisingly effective in practice

Memory Areas (kind area)

Primitive type constructors

Stored, LE, BE :: * → area (partial)

Array, Pad :: nat → area → area

Structures (special syntax, dot notation)

References and Pointers

Ref, Ptr :: area → *

ARef, APtr :: nat → area → *

Type Classes

- Ad-hoc polymorphism:

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

- Functional dependencies and notation:

```
(#) :: Bit n -> Bit m -> Bit (n+m)
instance ByteSize (Array n t) = n * ByteSize t
```

Type Classes

- Ad-hoc polymorphism:

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

- Functional dependencies and notation:

```
class (+) (n::nat) (m::nat) (p::nat)
  | n m -> p, m p -> n, p n -> m
(#) :: (n + m = p) => Bit n -> Bit m -> Bit p

class ByteSize (a::area) (n::nat) | a -> n
instance ByteSize (Array n t) p
      if ByteSize t m, n * m = p
```

Type Classes

- Ad-hoc polymorphism:

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

- Functional dependencies and notation:

```
(#) :: Bit n -> Bit m -> Bit (n+m)
instance ByteSize (Array n t) = n * ByteSize t
```

- Instance chains, explicit failure:

```
instance AESKey Word128
else     AESKey Word192
else     AESKey Word256
else     AESKey a fails
```

Unpointed Types

- Every type in Haskell is **pointed**:
  - Includes a bottom element denoting failure to terminate
  - Enables general recursion, complicates reasoning
- But many types in systems programming (e.g., bit fields, references,…) are naturally viewed as **unpointed**:
  - No bottom element, stronger termination properties, manipulated via primitive recursion or "fold" operations
- Could be modeled by lifting to attach "false bottom"
  - Better to handle directly; more expressive types

Integrating Unpointed Types

- Strategy for integrating unpointed types in Haskell proposed by Launchbury and Paterson in 1996
- Key idea: use type classes to identify dependencies on pointed types/general recursion

```
class Pointed t
   where fix :: (t -> t) -> t
```

- Previous experiments to explore how this would scale to a full language design are encouraging
- Providing appropriate semantic foundations is challenging, arguably less interesting for a call-by-value language

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

(More) Conventional FP

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

shift inferred from type

```
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
```

Could be compiled as a lookup table …

Leveraging Types: 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)
```

Leveraging Types: 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

Leveraging Types: 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

Leveraging Types: 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

Compilation Strategy

Habit → AST → HIL → Fidget → PPC/ARM

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

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

Watch this space ...