The 7 Features of Habit for Highly Assured Systems Programming

Mark P. Jones, on behalf of the HASP project

May 2009

Portland State UNIVERSITY | galois |

About that title …

HASP: High Assurance Systems Programming

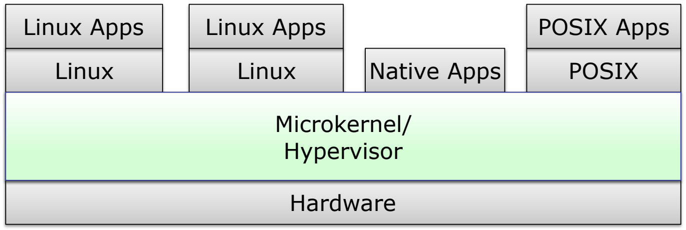
Tools and techniques to support the development of high assurance systems software:

- Operating system kernels
- Hypervisors
- VMMs
- Device drivers
- …

Vision

- A tool-chain for developing robust, reliable, and secure systems software that spans the full range of concerns:
  - From high-level analysis and verification
  - To low-level, performance-sensitive implementation

Example Application:



a working μ-kernel implementation with very high assurance of separation between domains

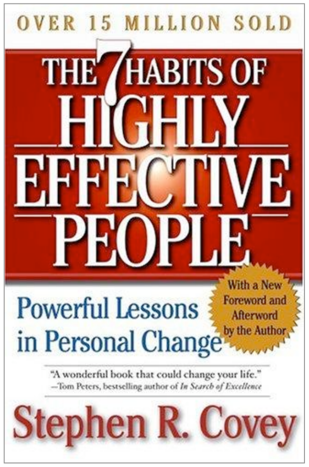
Habit (formerly "Systems Haskell")

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

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

The Book

- A well-known "self-help" book by Stephen Covey
- "Powerful lessons in Personal Change"
- First published in 1989
- Nothing to do with HASP...



OVER 15 MILLION SOLD

THE 7 HABITS OF HIGHLY EFFECTIVE PEOPLE

Powerful Lessons in Personal Change

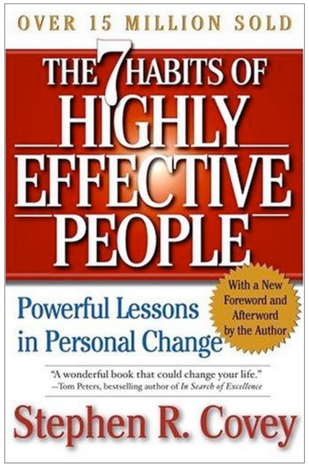
With a New Foreword and Afterword by the Author

"A wonderful book that could change your life."
—Tom Peters, bestselling author of *In Search of Excellence*

Stephen R. Covey

Original Talk Plan

- An introduction to the Habit language
- Showcase seven "cool" features of the design
- Only connection with the book: the number 7 and the word "Habit"



OVER 15 MILLION SOLD

THE 7 HABITS OF HIGHLY EFFECTIVE PEOPLE

Powerful Lessons in Personal Change

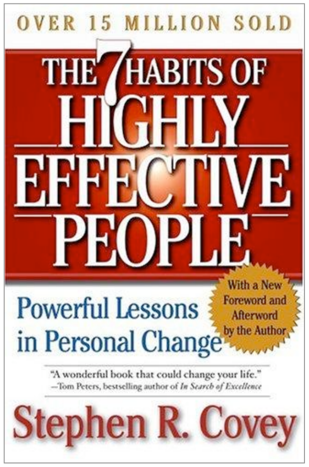
With a New Foreword and Afterword by the Author

"A wonderful book that could change your life."
—Tom Peters, bestselling author of *In Search of Excellence*

Stephen R. Covey

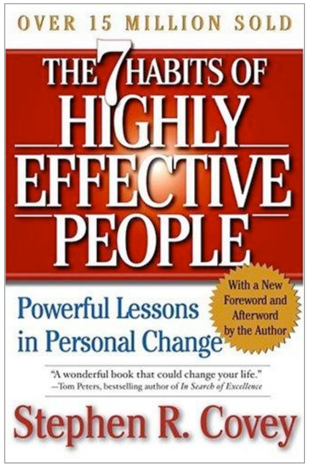
Maybe I should take a look …

- Personal Development
- Time Management
- Relationships
- Communication
- Leadership
- The Character Ethic
- The Abundance Mentality


OVER 15 MILLION SOLD
THE 7 HABITS OF HIGHLY EFFECTIVE PEOPLE
Powerful Lessons in Personal Change
With a New Foreword and Afterword by the Author
"A wonderful book that could change your life." —Tom Peters, bestselling author of In Search of Excellence
Stephen R. Covey

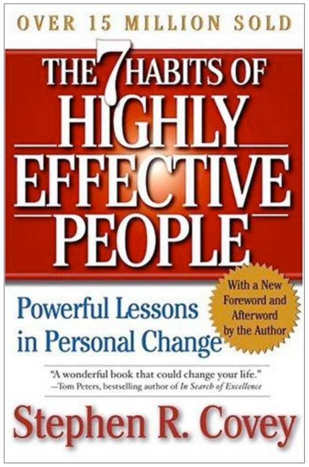
Principles & Values

"A holistic, integrated, **principle-centered** approach for solving personal and professional problems"


OVER 15 MILLION SOLD
THE 7 HABITS OF HIGHLY EFFECTIVE PEOPLE
Powerful Lessons in Personal Change
With a New Foreword and Afterword by the Author
"A wonderful book that could change your life." —Tom Peters, bestselling author of In Search of Excellence
Stephen R. Covey

Original Talk Plan

- ~~An introduction to the Habit language~~
- ~~Showcase seven "cool" features of the design~~
- ~~Only connection with the book: the number 7 and the word "Habit"~~


OVER 15 MILLION SOLD
THE 7 HABITS OF HIGHLY EFFECTIVE PEOPLE
Powerful Lessons in Personal Change
With a New Foreword and Afterword by the Author
"A wonderful book that could change your life." —Tom Peters, bestselling author of In Search of Excellence
Stephen R. Covey

The New Talk Plan

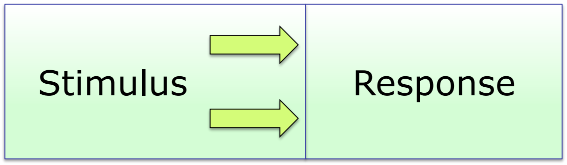
- An introduction to the Habit language
- Emphasis on underlying principles and values
- An occasional diversion into those "cool" technical features
- Use Dr Covey's names for the seven habits to structure the talk (sometimes with a very liberal reinterpretation)

Habit 1

Be Proactive

Principles of Personal Choice

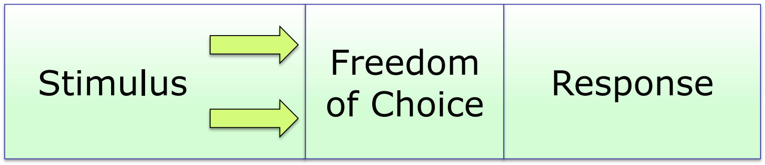
Between Stimulus & Response



Stimulus

Response

Between Stimulus & Response



Stimulus

Freedom of Choice

Response

- Response is a function of our decisions, not of our conditions
- We have the initiative and the responsibility to make things happen

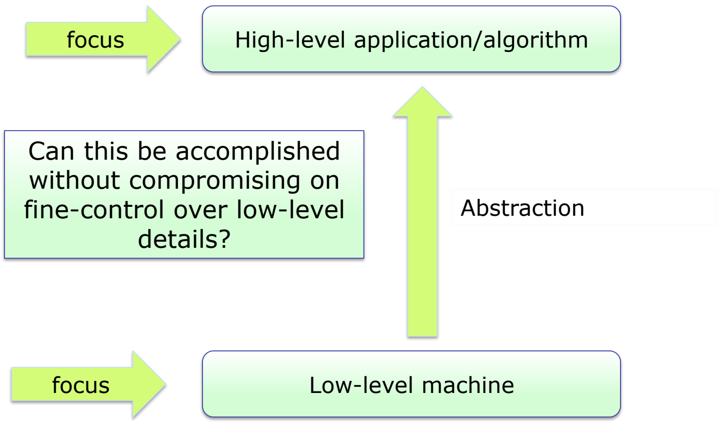
A Challenging Domain

Building high-assurance systems software is extremely challenging

1. Low-level operations, fine-grained control, performance sensitivity, …
2. Increasing functionality, increasing complexity, increasing need for assurance, …

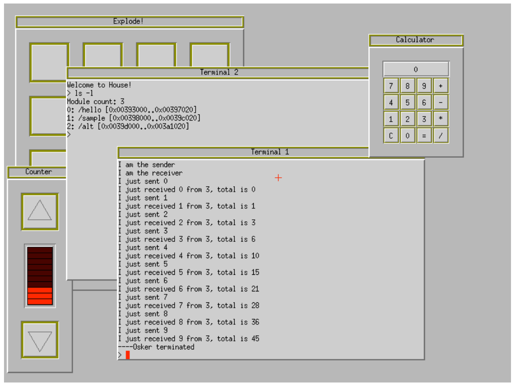
Current practice favors languages that are close to the metal, accommodating (1), neglecting (2)

Raising the Level of Abstraction


focus
High-level application/algorithm
Can this be accomplished without compromising on fine-control over low-level details?
Abstraction
focus
Low-level machine

The House Experience

- House is a proof-of-concept OS, written in Haskell:
  - Kernel + basic drivers (~5KLOC)
  - Network driver (~2KLOC)
  - GUI (~6KLOC)
  - Apps
  - User programs
- A starting point for the Galois HaLVM



```
Welcome to House!
> ls -l
Module count: 3
0: /hello [0x00393000..0x00397020]
1: /sample [0x00398000..0x0039c020]
2: /alt [0x0039d000..0x003a1020]
>
```

```
I am the sender
I am the receiver
I just sent 0
I just received 0 from 3, total is 0
I just sent 1
I just received 1 from 3, total is 1
I just sent 2
I just received 2 from 3, total is 3
I just sent 3
I just received 3 from 3, total is 6
I just sent 4
I just received 4 from 3, total is 10
I just sent 5
I just received 5 from 3, total is 15
I just sent 6
I just received 6 from 3, total is 21
I just sent 7
I just received 7 from 3, total is 28
I just sent 8
I just received 8 from 3, total is 36
I just sent 9
I just received 9 from 3, total is 45
----Osker terminated
>
```

Benefits of Using Haskell

Productivity: higher-level abstractions, genericity, reuse

Safety: built-in type and memory safety guarantees

Tractability: purity, referential transparency, encapsulation of effects, semantic foundations

The Price of Abstraction

- But for some systems programming applications, abstraction is a major barrier to adoption
- Examples:
  - Unknown data representation
  - Unpredictable execution behavior, performance, resource utilization, etc.
  - Runtime system: size and complexity
- Haskell has weaknesses in these areas

Habit 2

Begin with the End in Mind

Principles of Personal Vision

Habit

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

- Primary Commitments:
  - Systems Programming
  - High Assurance
  - Simplicity

Habit

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

- 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

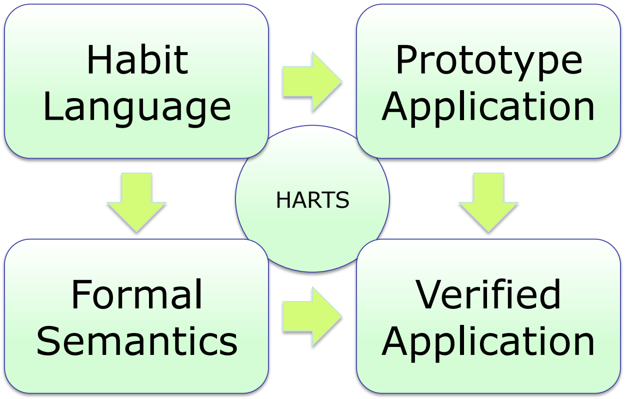
- A dialect of Haskell that is designed to meet the needs of high assurance systems programming

- High Assurance: a **full** and **formal** semantics that provides a basis for:
  - Mechanized reasoning
  - Meaningful assurance arguments
  - Verification of Habit programs (and, ultimately, Habit 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

The Big Picture



Habit Language

Prototype Application

HARTS

Formal Semantics

Verified Application

Realizing the Big Picture

- Each of these areas provides opportunities for local innovations and advances
- But our sights are set on the combination
  - Feasibility: Functionality, assurance, performance, cost
  - Technology Transfer: A toolset and a real world case study

Habit 3

Put First Things First

Principles of Integrity & Execution

The Time Management Matrix

	Urgent	Not urgent
Important	I	II
Not important	III	IV

The Time Management Matrix

	Urgent	Not urgent
Important	• Crisis management • Patching	• Prevention • New opportunities
Not important	• Interruptions	• Busy work

Why Build on Haskell?

- Increasingly broad adoption/use of Haskell
- Growing interest, strong community
- Avoid reinventing the wheel:
  - Syntax: familiar notations and concepts
  - Semantics: powerful, expressive type system
- Leave time to focus on what is new

Semantic Foundations

- Exploring a formal semantic framework using denotational techniques that can be expressed in well-developed domain theory
- Automated (for example, in Isabelle; see upcoming publication in TPHOLs conference)
- Expect to develop a corresponding operational semantics for treatment of resource sensitivity

Properties for Assurance

- Properties about the language
  - Examples: type and memory safety
- Equivalences between program fragments
  - Examples: for use in reasoning, transformation, optimization, synthesis
- Properties of implementations
  - Example: preservation of semantics
- Properties of applications
  - Example: separation properties of a hypervisor

Habit 4

Think Win/Win

Principles of Mutual Benefit

Example: Type Safety

Developer Win:

Earlier detection of bugs during development

User Win:

More secure deployed systems

Example: Type Safety

Developer Win:

Earlier detection of bugs during development

User Win:

More secure deployed systems

Certifier Win:

Many safety properties enforced automatically via types

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: 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

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: `(@) :: Ix n -> Ref (Array n t) -> Ref t`
- Amortized construction of safe indices with comparisons that are already required

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

Example: Side Effects

- Presence of potential side-effects (e.g., state, exceptions, …) is made explicit in types via monads: `A -> M B`
- A single program can use multiple monads
- Some operations are generic in the monad, and others that are specific to a particular monad

- Particular relevance to systems 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

- A monadic type system can document and enforce correct use of modes at compile-time

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

Habit 5

Seek First to Understand, Then to be Understood

Principles of Mutual Understanding

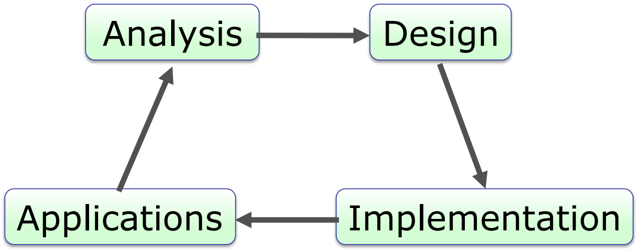
Understanding the Domain

- What are the driving needs of the systems programming domain?
- How can we best address those needs in our design?

Development Model

Prior work at PSU
and at Galois

Haskell, ML, …
Types, Monads



House, HaLVM, L4,
BAC, dombuilder, …

Hobbit, RML, …

Requirements

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

Initial Habit Design: Summary

- Simplified dialect of Haskell
  - Foundations: pure, higher-order, typed
  - Syntax: definitional style, lightweight notation
- Changes/additions
  - Strict evaluation; bitdata; memory areas; type-level numbers; unpointed types; monadic sugar

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

Familiar box layout notation

… continued

```
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”

Fine Control of 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:

```
type Mapping = struct
  [ 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..

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 (based on Hobbit prototype):

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

Portable Assembly Language

- Habit shares with C the goal of being a portable assembly language:
  - High-level (e.g., expressions not registers)
  - Intuitive (albeit approximate) mapping to machine; predictable performance/costs
- Except that Habit will:
  - Provide a formal semantics from the outset
  - Allow more precise control over data layout
  - Support higher-level programming features
  - Eschew the use of unsafe primitives

Performance Annotations

- How are resource sensitivity and performance expectations captured in code?
- Open research problem
- Initial approach for Habit:
  - Performance annotations to guide code generation
  - Compiler feedback to guide programmer refactoring of code

```
[noalloc]
sum [1..10]

[noalloc]
let loop t n
 = if n>10 then t
   else loop (t+n) (n+1)
in loop 0 1


[noalloc]
natfold 0 (+) 10
```

Habit 6

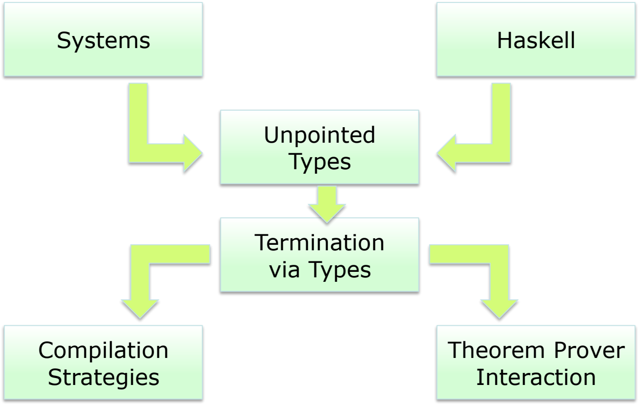
Synergize

Principles of Creative Cooperation

Synergy

- When 1 + 1 is more than 2
- Synergy via our collaboration with Galois
- Synergy via technical developments

Unpointed Types



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, primitive recursion still possible via "fold" operations
- Could be modeled by lifting to attach "false bottom"
  - Better to handle directly; more expressive types

Integrating Unpointed Types

- A strategy for integrating unpointed types in Haskell using type classes was proposed by Launchbury and Paterson
- We are scaling this to a full language design
- Example:

```
fpsize                      :: Bit 6 -> Bit 6
fpsize n | n==1              = 32
         | n<12              = 0
         | otherwise         = n
```

  Finite, pointed domain and range enables implementation as a lookup table, computed at compile-time

Habit 7

Sharpen the Saw

Principles of Balanced Self-Renewal

The Fable:

- Sawing down a tree will be easier if you pause from time to time to sharpen the saw
- Less time hacking
- Take some time to improve the tools

Paradigm Shifts

- According to Thomas Kuhn in “The Structure of Scientific Revolutions”:

- Almost every significant break through in the field of scientific endeavor is first a break with tradition, with the old ways of thinking

Time for a new Paradigm Shift?

- The systems programming community went through a major paradigm shift in the move from assembly to C**,** enabling:
  - New levels of functionality
  - New levels of portability

- Languages like Habit are positioning for a new paradigm shift that will enable:
  - New levels of assurance and security

Current Status

- On target to complete baseline design and implementation this summer:
  - Language design
  - Front-end implementation (parser, type checker, …)
  - Formal semantics
  - Small case studies

- In progress:
  - Prototype backend via Leroy’s Compcert framework for semantics preserving compilation
  - Integration with HARTS
  - Demonstration application

Conclusions

One Habit for Highly Effective High Assurance Systems Programming:

- Builds on critical successes in the design of Haskell

- Reflects requirements and feature set for the systems programming domain

- Provides foundations for formal verification

- Serves as a platform for future research and technology transfer activities