

Writing Systems Software in a Functional Language

An Experience Report

Iavor Diatchki, Thomas Hallgren, Mark
Jones, **Rebekah Leslie**, Andrew Tolmach



Background:

- House: a demonstration kernel
 - Kernel, scheduler, device drivers (Video, Keyboard, Mouse, Network), etc...
 - Applications: GUI, terminal, calculator, ...
 - Execution of arbitrary user programs

- L4 microkernel implementation (incomplete)
 - Scheduling, IPC, memory management
 - Intended target for formal verification, focusing on security properties such as separation



The “Systems Haskell” Project:

- Preserve critical, distinguishing features of the current Haskell design
 - Expressive type system
 - Support for abstraction
 - Strong mathematical foundations
- Address the shortcomings of Haskell for writing high-assurance systems software
 - Safe access to low-level operations
 - Small, configurable, trustworthy run-time system
 - Ability to manage resource usage explicitly



Low-level Operations in Haskell:

- The Foreign Function Interface (FFI) provides a way to access low-level machine features
 - Some features (e.g., reading and writing memory, manipulating pointers) are already included
 - Others can be coded in C and/or assembly and then packaged as new Haskell primitives
- But many of these functions are unsafe
- Requires careful and disciplined use to ensure that type and memory safety are preserved



Safe Low-level Operations Using the H-Interface:

- A hardware abstraction layer that encapsulates the use of unsafe low-level primitives
- Goals: small, reusable, safe
- Semantics and safety guarantees captured by a collection of formal properties
- Still requires careful and disciplined construction



Observations on Safe Low-level Operations:

- Many low-level operations *are* safe, but the FFI provides no way to enforce safety constraints
 - Example: restricting memory accesses to a particular region of memory
 - Solution: typed memory areas and bitdata primitives
- Truly untrusted low-level operations should not be able to break orthogonal safety guarantees
 - Example: preventing foreign function calls from accessing the heap



RTS Support in Haskell:

- Both of our kernels rely on a port of the GHC run-time system that runs on bare metal*
- Characteristics of GHC:
 - Open source, active development community
 - 50-75kloc, many configurations, unsupported legacy code
 - Full featured (support for powerful extensions)
 - Many of these features aren't needed by our systems
 - Highly optimized
 - Difficult to understand, let alone reason about

* Initial port done by Carlier and Bobio's hOp project.



Observations on RTS Design:

- A monolithic run-time system cannot be both:
 - Generic enough to support all of the different forms of memory allocation and/or concurrency that are required in a range of systems applications
 - Simple enough to enable a manageable, verifiable implementation
- A highly modular run-time system can support:
 - Independent certification of components
 - Configuration/customization of application-specific run-time systems that include only the components that are needed



Resource Management in Haskell:

- High-level languages are designed to hide low-level implementation details from programmers
- Example: garbage collection presents the illusion of an infinite memory
 - Allocate without checking, avoid nasty pointer bugs
 - GC algorithm quietly identifies & recycles garbage
- But real computers do have real limits:
 - Programs cannot run if the heap is exhausted
 - Separate threads can starve one another by consuming resources



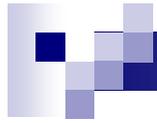
Observations on Resource Management:

- New language mechanisms, backed by run-time support, are needed:
 - To enable fine-grained accounting for memory usage
 - To support dynamic partitioning and reallocation of memory between threads
 - To provide facilities for detecting and recovering from memory overflow
- What lessons can we learn from the systems community, where such issues receive much greater attention?



Summary:

- Developing systems software is challenging
- Higher-level languages can help programmers to manage the complexity of low-level details while also producing code that is amenable to high-level analysis and verification
- Critical focus areas for Systems Haskell include:
 - safe and direct manipulation of low-level structures
 - run-time system configuration
 - reflective resource management
 - performance



Questions?