Verifying an Operating System Kernel

Michael Norrish

Windows

An exception 06 has occured at 0028:C11B3ADC in VxD DiskTSD(03) + 00001660. This was called from 0028:C11B40C8 in VxD voltrack(04) + 00000000. It may be possible to continue normally.

* Press any key to attempt to continue.
* Press CTRL+ALT+RESET to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue

seL4 + L4.verified

Goals:

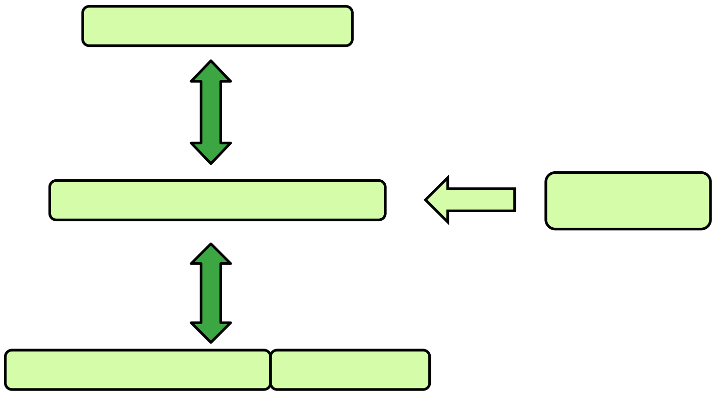
- Formal specification of kernel and machine
- Verified production quality, high-performance kernel

Address problems in older L4s:

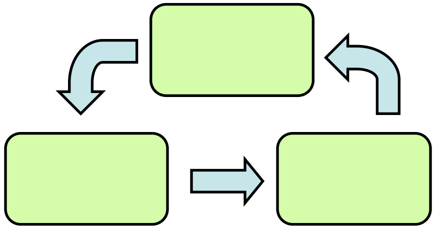
- Communication control
- Kernel resource accounting
- No performance penalty for new features
  - 30 cycles per syscall ok. Maybe.

Overview

- The seL4 Kernel
  - Interface
  - State
  - Kernel Objects



- Interesting Problems
  - Designing and formalizing an OS kernel
  - Coping with C
  - Refinement on monadic functional programs



Credit Where It's Due

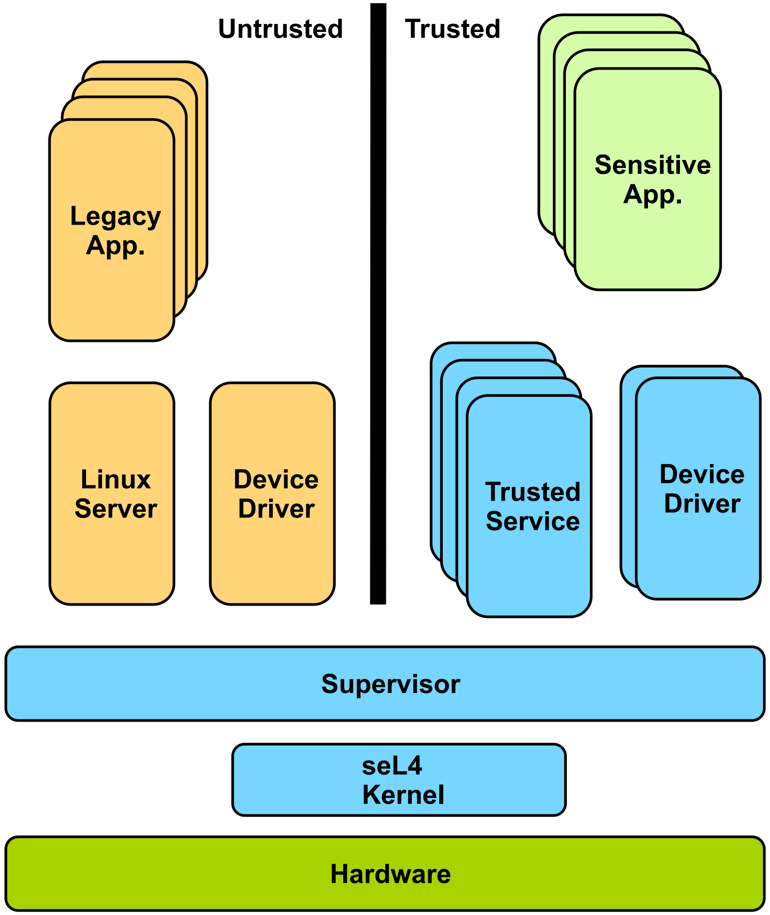
- L4.verified:
  - Led by Gerwin Klein
  - Verifiers (Sydney): four research assistants, three PhD students, under-grad projects, two/three researchers (including 100% of Gerwin)
  - Tool support (Canberra): three researchers
  - Project entering its fourth (and last) year
- L4 itself
  - "L4" is really a family of (open source) implementations
  - Many people over many years
    - including new spin-off company Open Kernel Labs

seL4

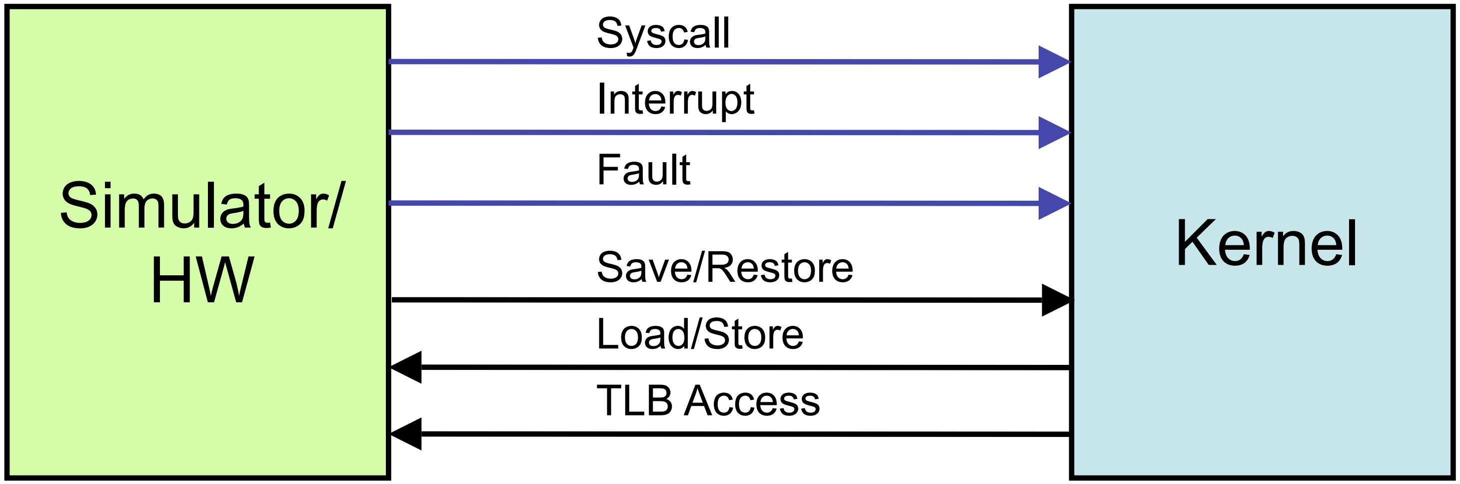
secure embedded L4

Small Kernels

- Smaller, more trustworthy foundation
  - Hypervisor, microkernel, nano-kernel, virtual machine monitor, isolation kernel, partitioning kernel, exokernel…..
  - Fault isolation, fault identification, IP protection, modularity…..
  - High assurance components in presence of other components



Kernel Interface



- Kernel is a state transformer:

```
kernel :: Event ⇒ KernelState ⇒ KernelState
```

Kernel State

- Physical memory

  Storage: `obj_ref` ⇒ `kernel_object option`

- Mapping database

  Capability derivations: `cte_ref` ⇒ `cte_ref option`

- Current thread

  Pointer: `obj_ref`

- Machine context

  Registers, caches, etc

Kernel Objects (simplified)

- Capability Table

```
cap_ref ⇒ capability
```

- Thread Control Block (TCB)

```
record        ctable, vtable :: capability
                  state :: thread_state
                  result_endpoint, fault_endpoint ::
    cap_ref
                ipc_buffer :: vpage_ref
                context :: user_context
```

- Endpoint:

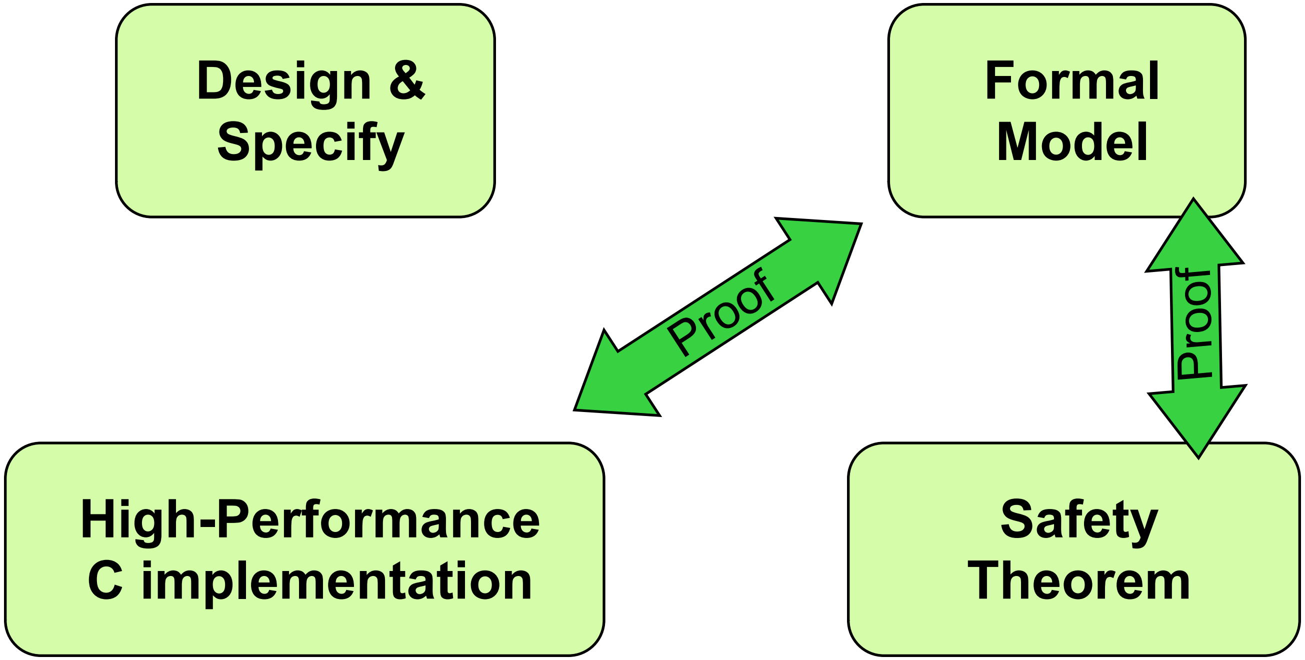
```
Idle | Receive (obj_ref list) | Send (obj_ref list)
```

- Data Page

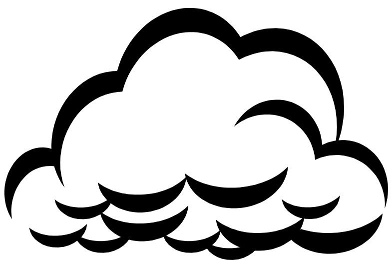
Designing and Formalising

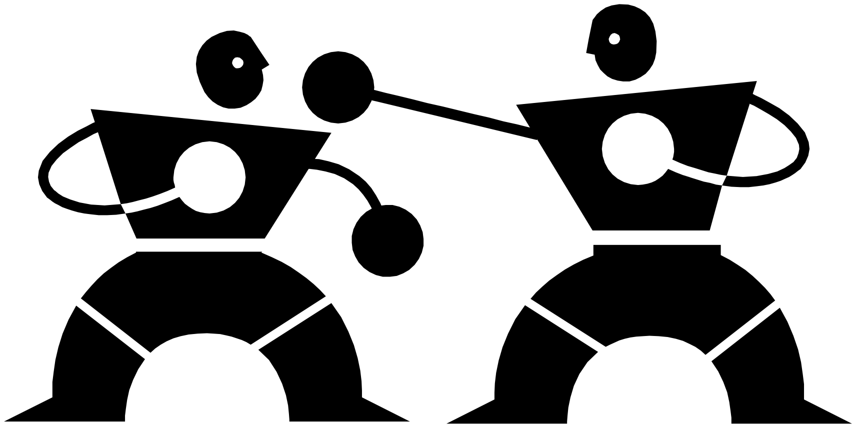
concrete syntax is everything

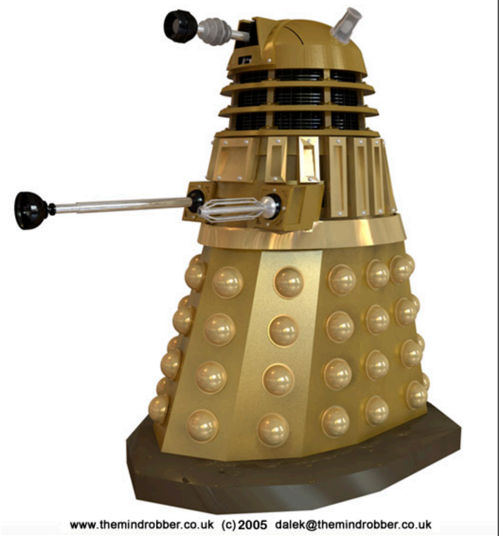
Designing and Formalising a New Kernel



Kernel Developers
Versus
Formal Methods Practitioners

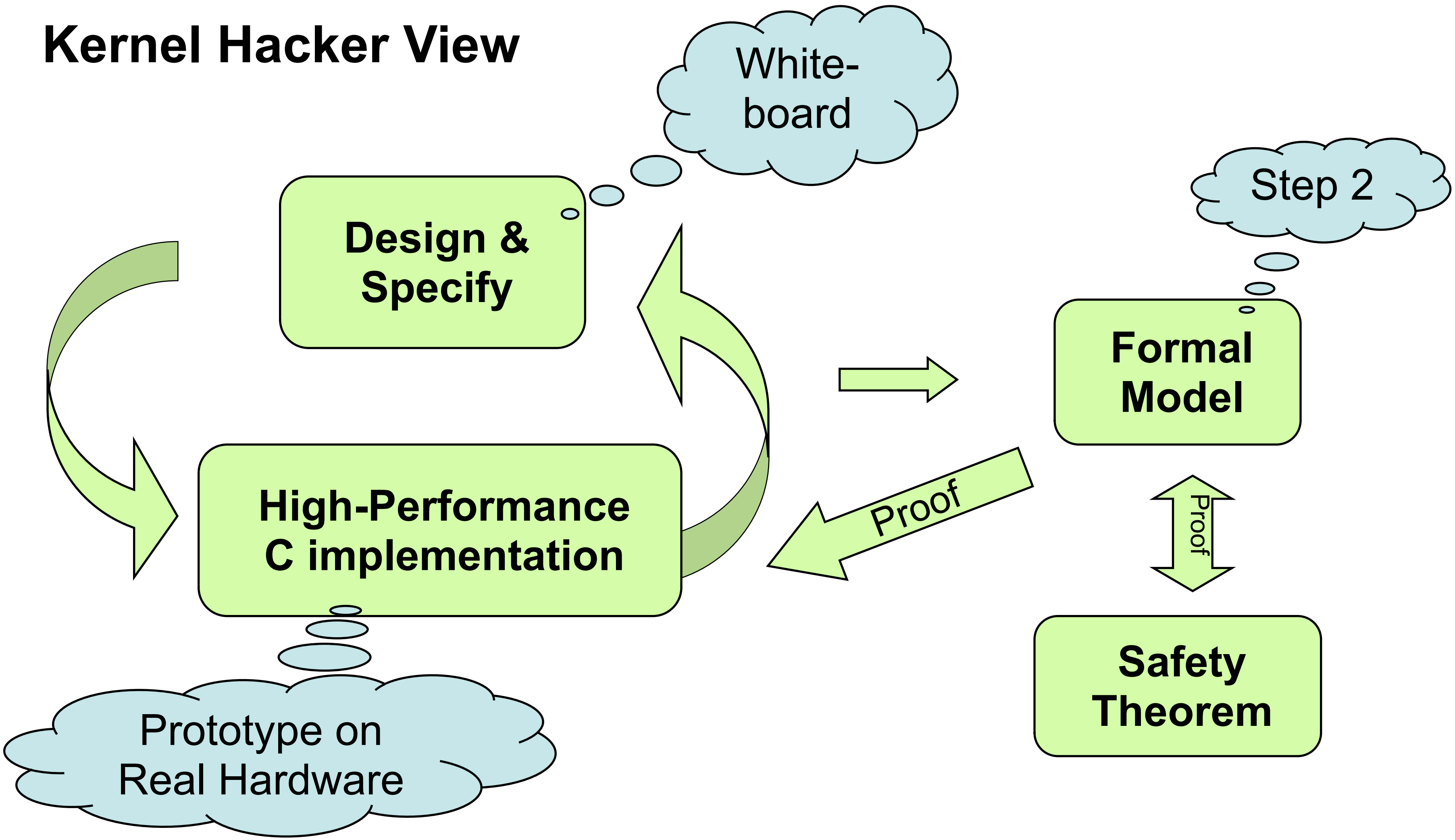






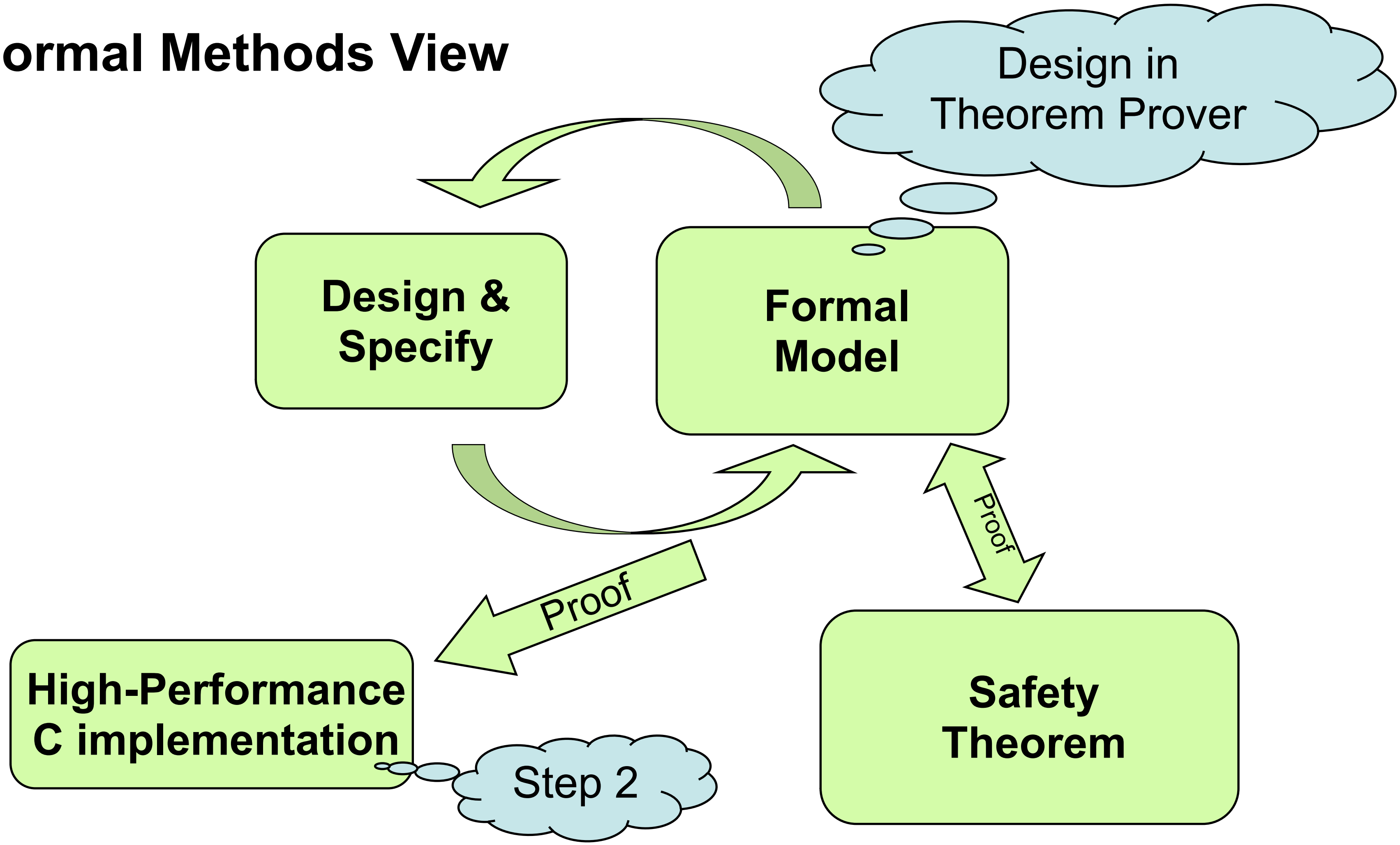
Standard Kernel Design

Kernel Hacker View

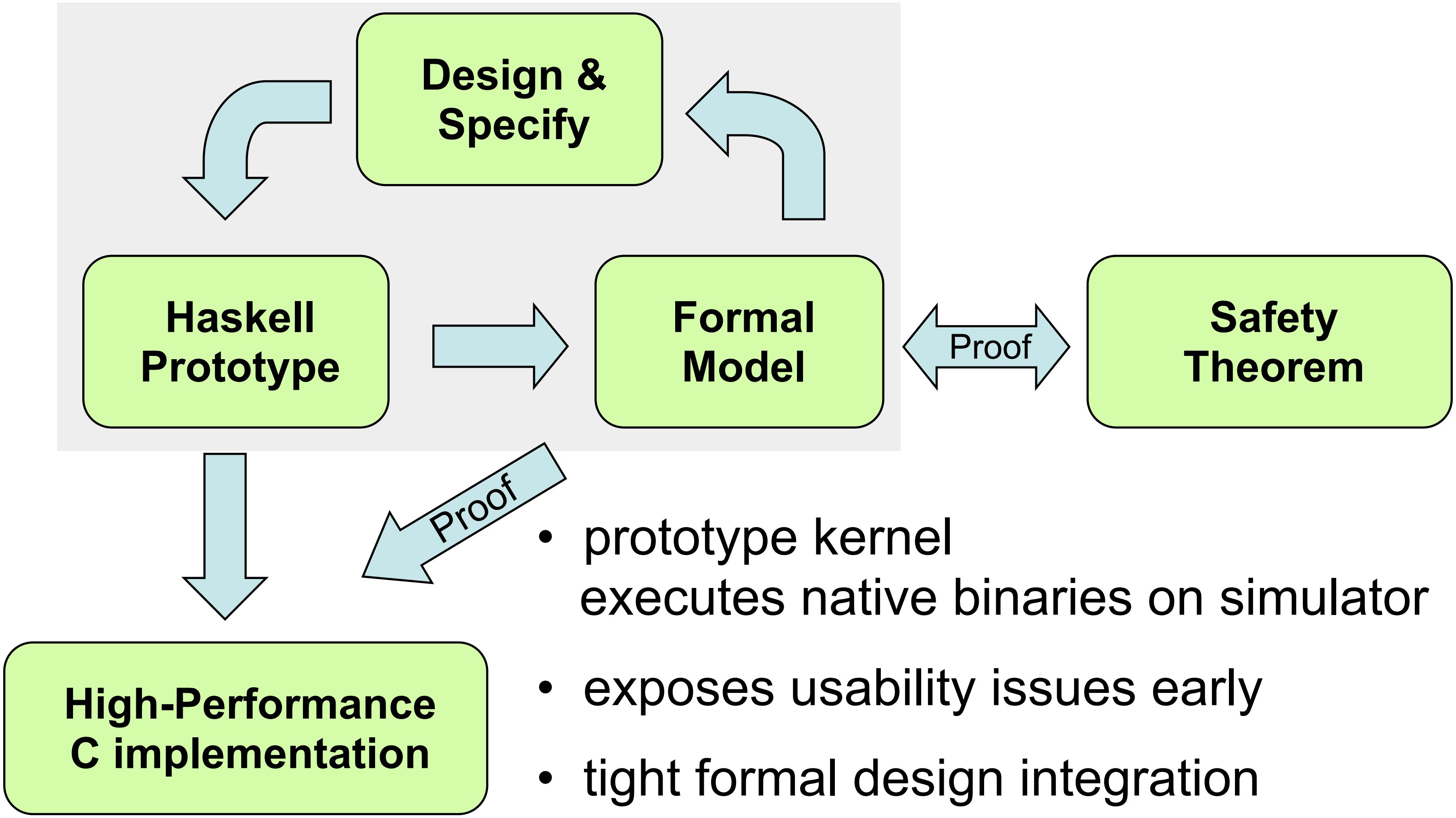


Formal Design

Formal Methods View

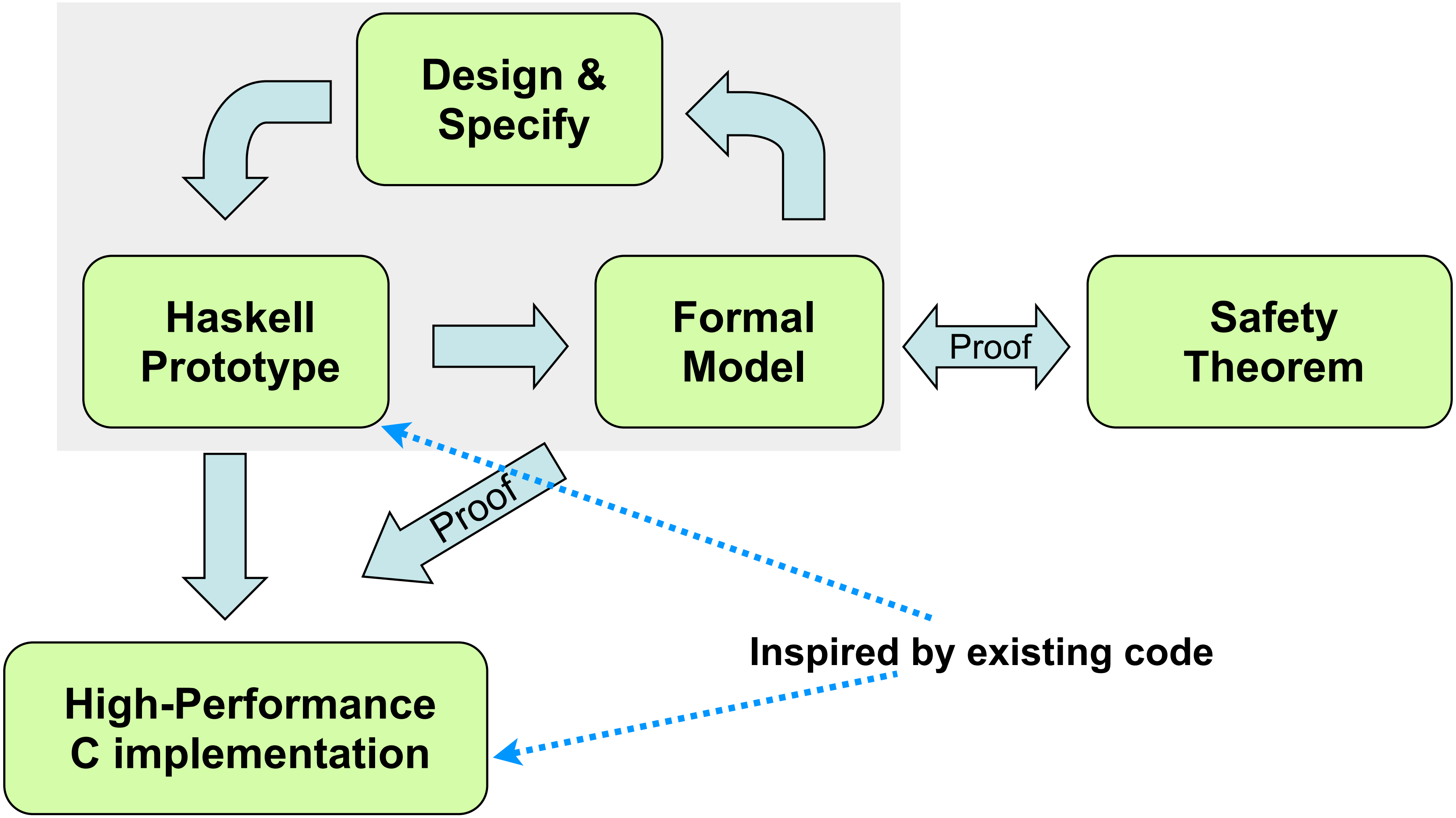


Iterative Design and Formalisation



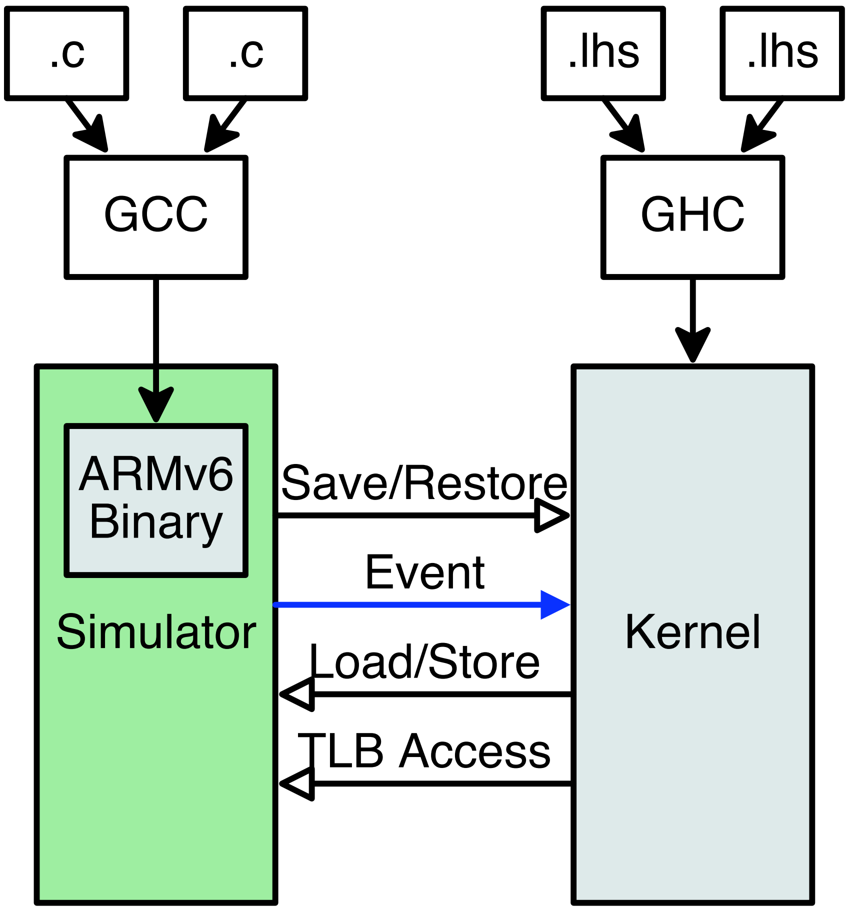
- prototype kernel
  executes native binaries on simulator
- exposes usability issues early
- tight formal design integration

Iterative Design and Formalisation



User-Level Simulation

- User-level CPU simulator
  - M5 Alpha simulator
  - Locally-developed ARMv6 simulator
  - QEMU
- Executes compiled user-level binaries
- Sends events to the Haskell kernel



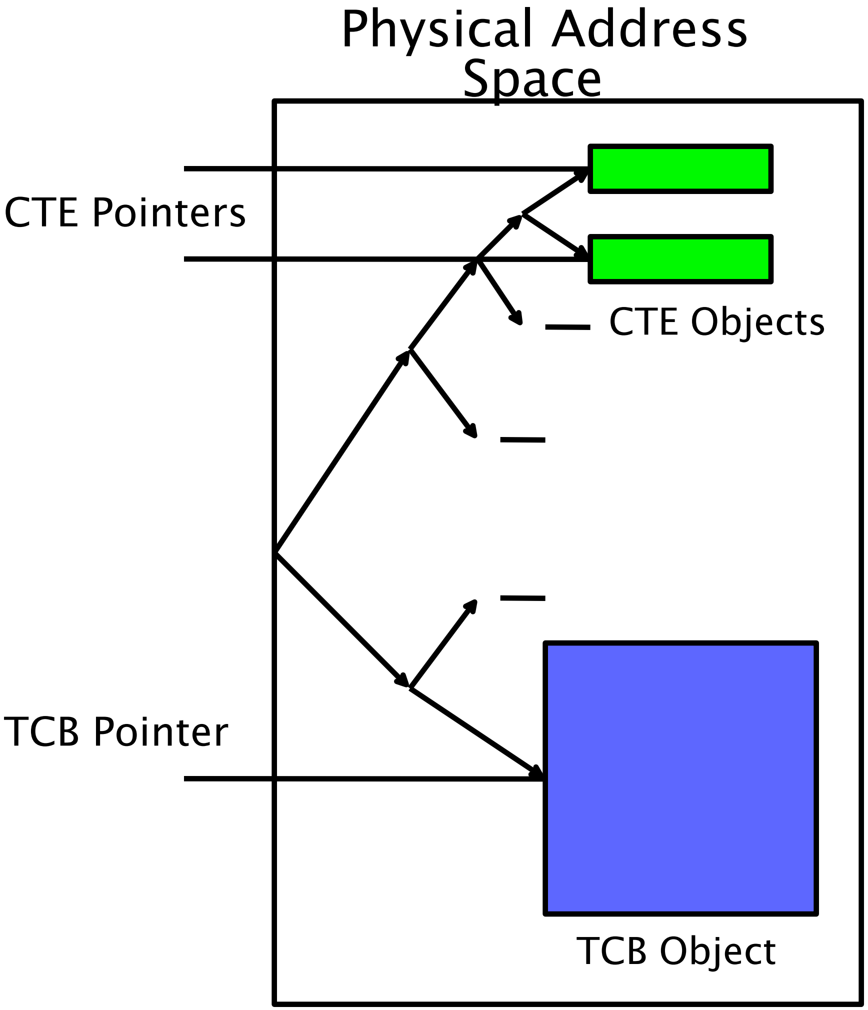
Machine Monad - Lowest Level of Model

- `getMemoryTop :: MachineMonad (PPtr ())`
- `getDeviceRegions :: MachineMonad [(PPtr (), Int)]`
- `loadWord :: PPtr Word -> MachineMonad Word`
- `storeWord :: PPtr Word -> Word -> MachineMonad ()`
- `insertMapping :: PPtr Word -> VPtr -> Int -> Bool -> MachineMonad ()`
- `flushCaches :: MachineMonad ()`
- `getActiveIRQ :: MachineMonad (Maybe IRQ)`
- `maskInterrupt :: Bool -> IRQ -> MachineMonad ()`
- `ackInterrupt :: IRQ -> MachineMonad ()`
- `waitForInterrupt :: MachineMonad IRQ`
- `configureTimer :: MachineMonad IRQ`
- `resetTimer :: MachineMonad ()`

- Foreign Function Interface (FFI)
- Approximate machine-level C functions
- Close to “real” as possible
  - Forces us to manage “hardware”

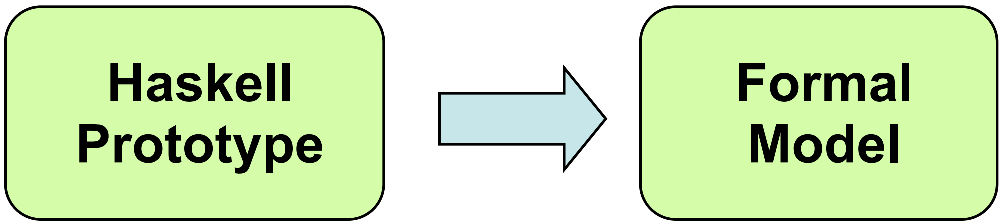
Kernel-State Monad

- Physical memory model
  - Contents of dynamically-allocated memory
  - Typed kernel data
    - Thread control blocks
    - Capability and page tables
  - Indexed by physical memory address
- Forces us to model memory management (30% of kernel)
- Reduces the gap to C
  - Pointers,not Haskell's



Haskell to Isabelle/HOL

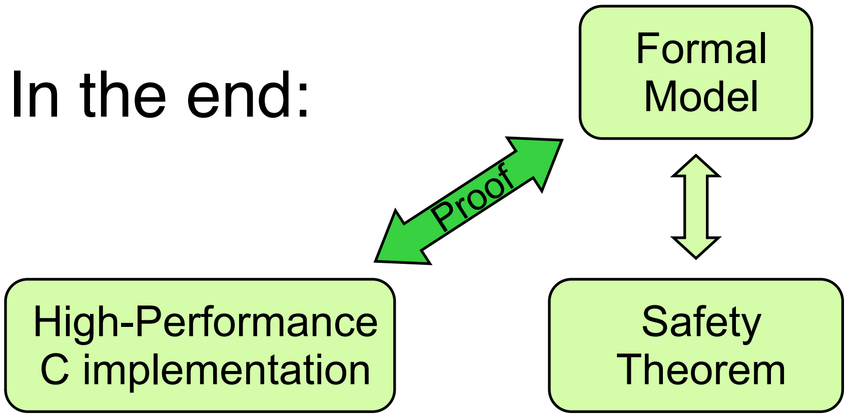
- Needs to be quick and easy:



- Problems:
  - Size (3000 loc)
  - Real-life code (GHC extensions, no nice formal model)
  - Want Isabelle/HOL for safety and refinement proofs
  - Existing tools do not parse the code

Approach: Quick and Dirty

- In the end:



- No “hard” translation correctness guarantee
- Remaining issues:
  - Special features (“Dynamic”)
  - Termination
  - Monads

Termination

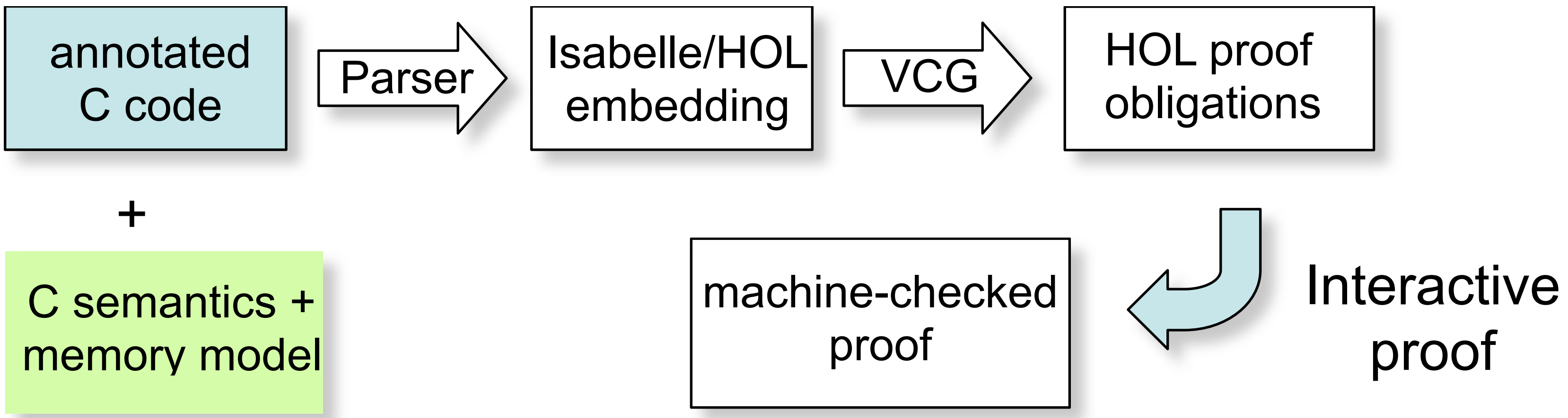
- Haskell:
  - Lazy evaluation
  - Non-terminating recursion possible
- Isabelle/HOL:
  - Logic of total functions
- But:
  - All system calls terminate
  - We prove termination
  - So far: done, relatively easy, not much recursion
    (one proof required ugly, but true, assumption)

Monads

- Haskell kernel:
  - Imperative, monadic style throughout
- Isabelle/HOL:
  - Type system too weak to implement monads in the abstract
- But:
  - Strong enough to implement concrete monads (state, exception)
  - Nice do-style syntax in theorem prover
  - So far: needed more concrete than abstract properties for proofs

Dropping Down to C

- Refinement target is C code
  - C code whose “shape” we know in advance
  - C code we will be able to optimise and refine later
  - C code that will include assembler for fast paths



C Problems Avoided

- Abstract machine details can be implementation-defined.
  - E.g., endian-ness, behaviour of arithmetic operations, size of bytes(!)
    - For a particular verification, treat these choices as given (know that bytes are 8-bits long, know that ARM chips are run little-endian, etc.)
- Other aspects can be unspecified.
  - E.g., order of evaluation of expressions (significant in presence of side effects)
    - Make expressions with side effects illegal.
- Worst: lots of behaviour is undefined.

C Problems Avoided

- Undefined behaviour is illegal behaviour:
  - dividing by zero, accessing memory at bad alignments, writing to unallocated memory...
- When translating from C input, annotate possibly badly behaved expressions with guards.
- For example, when translating

  ```
  *(p + 1) = 3;
  ```

  add guard requiring that address `(p + 1)` be a valid address for an integer.

Guards

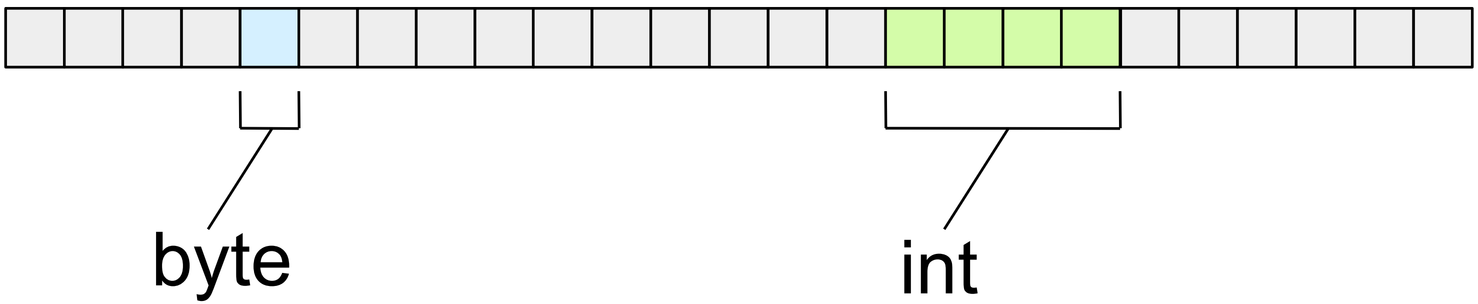
- A guard is an arbitrary boolean expression over program states. If true of a state, the program is allowed to continue. (Otherwise, implicitly, it aborts.)
- Guards can be used to simulate arbitrarily complicated run-time checks.
- The verification environment (Isabelle/HOL) requires the verifier to prove that the guard is true whenever the attached statement is about to be run.

Guard Flexibility

- Generation of guards can be customised to suit particular verification tasks.
  - In some OS environments, address zero holds the start of a exception vector; reading and writing the null pointer has to be allowed.
  - Writing unallocated memory may be a necessary part of the action of a memory management module
    - reading uninitialised memory might still be treated as a run-time error
- The underlying verification environment can be simple and language-agnostic.

C Memory Model

- Underlying model must be *addr* ⇒ *byte*



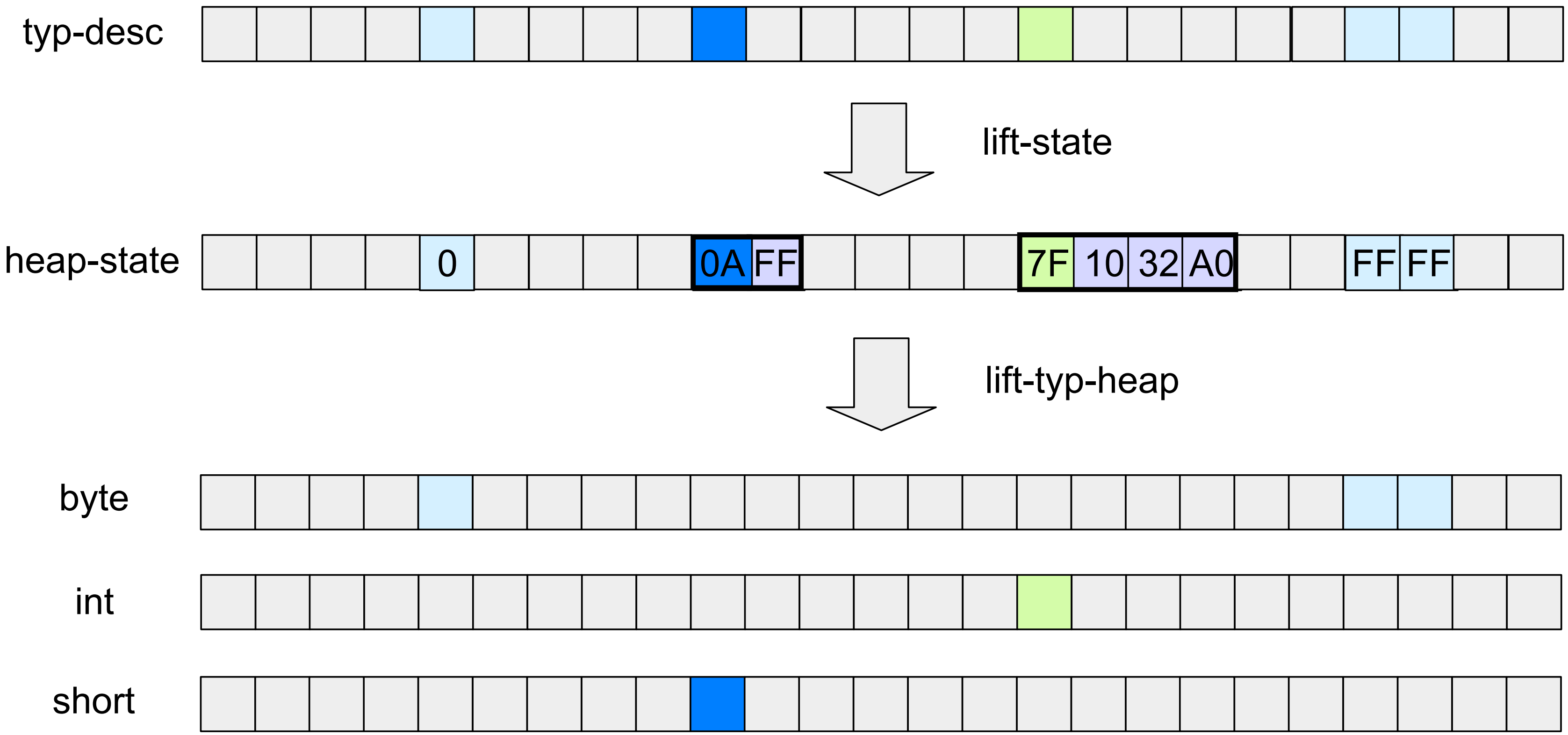
- This is unusable

- Lots of work by Harvey Tuch has allowed layering of higher-level, typed abstractions over this when code is “well-behaved” (which is often)

Multiple Typed Heaps Co-Existing

A type description gives one type to each address.

Accesses that respect it can happen independently.



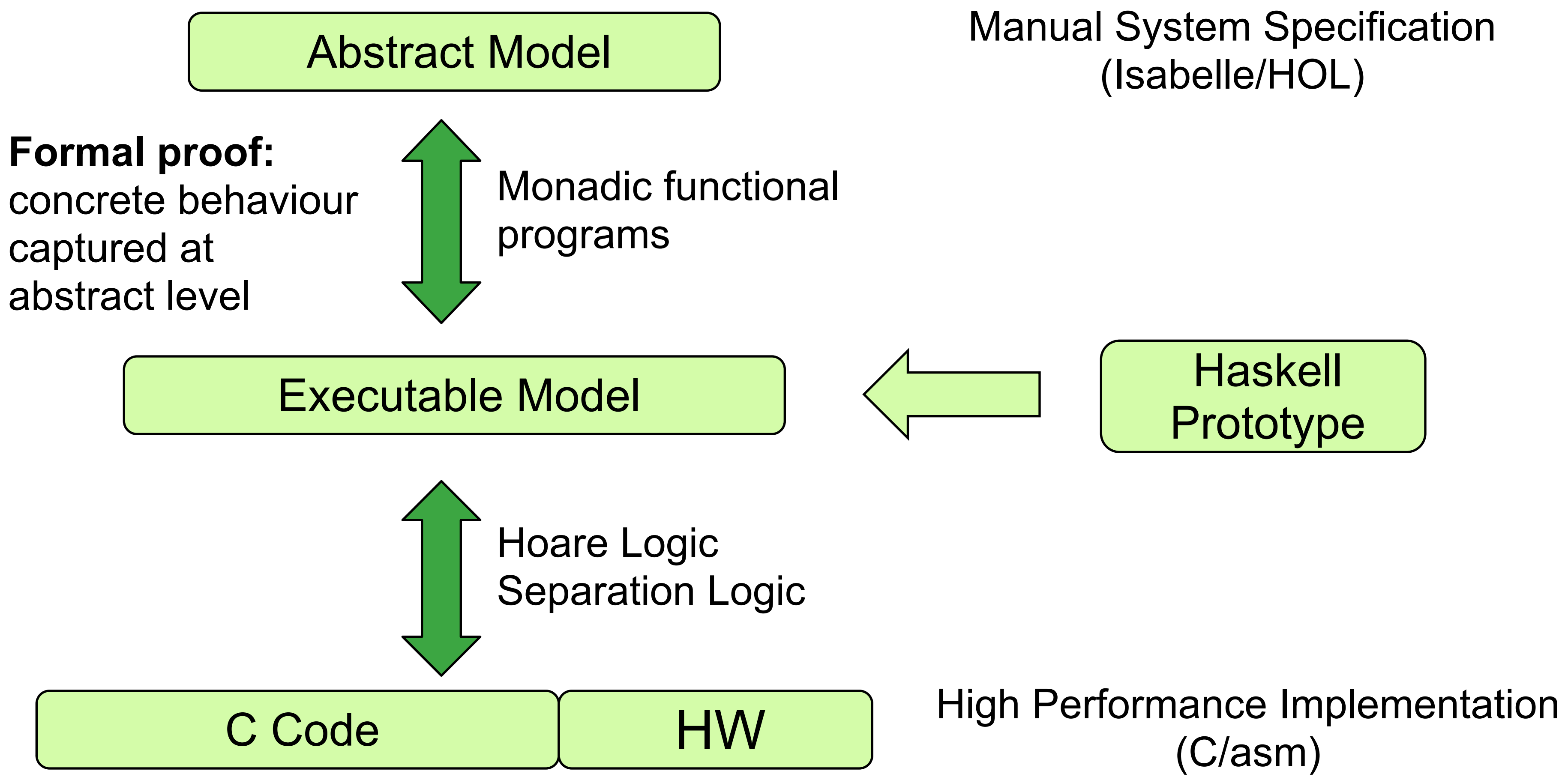
C Experience

- Technology and theory prototypes have been developed
  - parser, VCG, typed heaps
- Case studies (independent of any refinement) have established that the basic verification environment is usable
- The proof of the pudding will soon be upon us...

The Proof

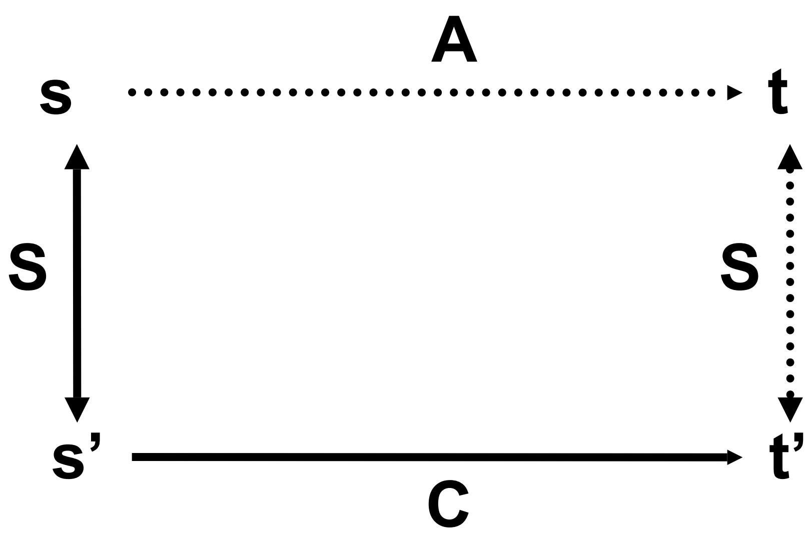
Refinement on monadic functional programs

Overview


Abstract Model
Manual System Specification
(Isabelle/HOL)
Formal proof:
concrete behaviour
captured at
abstract level
Monadic functional
programs
Executable Model
Haskell
Prototype
Hoare Logic
Separation Logic
C Code
HW
High Performance Implementation
(C/asm)

Refinement

- The old story:
  - C refines A if all behaviors of C are contained in A
- Sufficient: forward simulation



State Monad in Isabelle

- Nondeterministic state monad:

```
types  (σ,α) monad  =  σ ⇒ (α * σ) set

return :: α ⇒ (σ, α) monad
return x s == {(x,s)}

bind (>>=) :: (σ, α) monad ⇒ (α ⇒ (σ, β) monad) ⇒
                              (σ, β) monad
f >>= g == λs. U (λ(v,t). g v t) ` (f s)

fail :: (σ, α) monad
fail s = {}
```

Hoare Logic for the State Monad

- Hoare triples with result values:

  `{P} f {Q} == ∀s. P s → (∀(r,s') ∈ f s. Q r s')`

- WP-Rules:

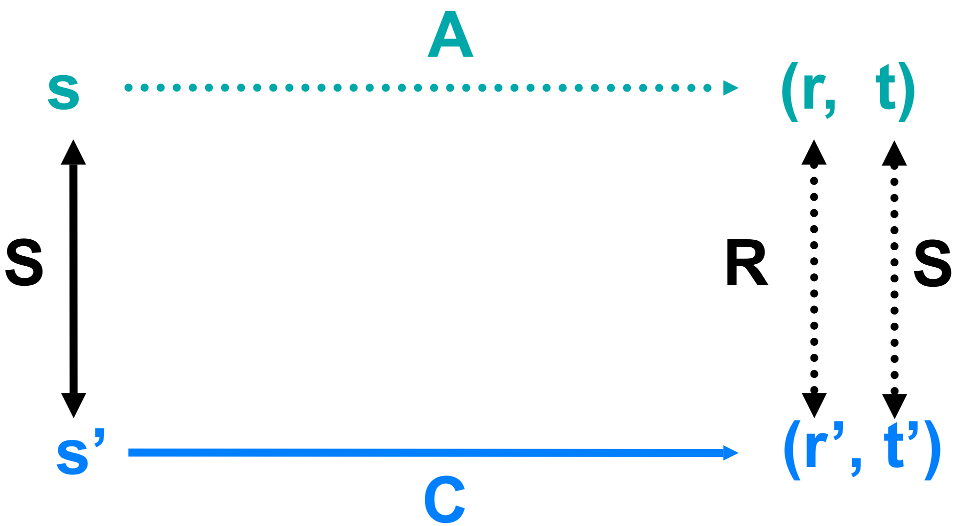
$$\frac{}{\{P\ x\}\ \text{return}\ x\ \{P\}}$$

$$\frac{\{\,P\,\}\,f\,\{\,Q\,\} \qquad \forall x.\,\{\,Q\ x\,\}\,g\ x\,\{\,R\,\}}{\{P\,\}\,f >\!\!>\!= g\,\{R\,\}}$$

$$\frac{}{\{\,P\,\}\ \text{fail}\ \{\,Q\,\}}$$

State Monad Refinement

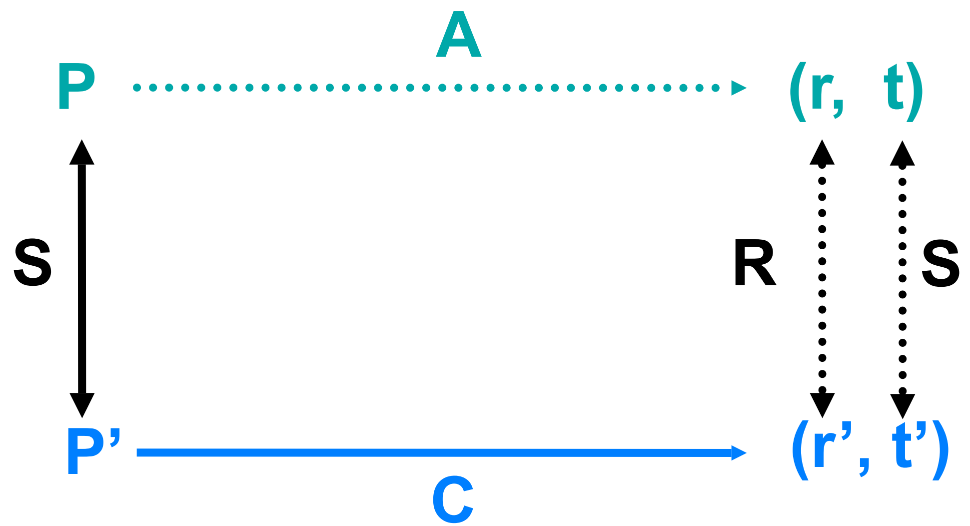
- Forward Simulation




```
corres S R A C ==
  ∀(s,s') ∈ S.
      ∀(r', t') ∈ C s'.
    ∃(r, t) ∈ A s.  (t, t') ∈ S ∧  (r, r') ∈ R
```

State Monad Refinement

- Forward Simulation




```
corres S R P P' A C ==
  ∀(s,s') ∈ S. P s ∧ P' s'
      ∀(r', t') ∈ C s'.
        ∃(r, t) ∈ A s. (t, t') ∈ S ∧ (r, r') ∈ R
```

A Small Refinement Calculus

$$\frac{}{\texttt{corres S R P P' A fail}}$$

$$\frac{\texttt{(x,y)} \in \texttt{R}}{\texttt{corres S R P P' (return x) (return y)}}$$

$$\frac{\begin{array}{c}\texttt{corres S R' P P' f f'} \\ \forall\, \texttt{x y. (x,y)} \in \texttt{R'} \to \texttt{corres S R (Q x) (Q' y) (g x) (g' y)} \\ \texttt{\{P\} f \{Q\}} \\ \texttt{\{P'\} f' \{Q'\}}\end{array}}{\texttt{corres S R P P' (f >>= g) (f' >>= g')}}$$

Summary

- Monadic style supports Refinement and Hoare Logic nicely
  - get, put, modify, select, or, assert, when, if, case, etc analogous

- Statistics:
  - 3.5kloc abstract, 7kloc concrete spec (about 3k Haskell)
  - 35kloc proof so far (estm. 50kloc final, about 10kloc/month)
  - 22 patches to Haskell kernel, 90 to abstract spec
  - 7-10kloc of C/asm expected for final product

- Invariants:
  - well typed references, aligned objects
  - thread states and endpoint queues
  - well formed current thread, scheduler queues

Future Work

- From http://www.ok-labs.com

  We are collaborating closely with NICTA ... on developing the first fully verified, proven bug-free operating systems kernel within two years.

- Just advertising?

Thank You