TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

A Principled Approach to Operating System Construction in Haskell

Thomas Hallgren, Mark P. Jones, Rebekah Leslie, Andrew Tolmach

Dresden, 2008-03-26

What is it all about?

- Writing operating systems in Haskell
- Provide a Haskell framework, which abstracts hardware access, consisting of:
  - Haskell Runtime System
  - Hardware monad
- **Thesis**: using pure functional languages facilitates construction of high assurance systems

OS construction in Haskell ?

Haskell's features:

- Strong static typing, memory safe
- Neither pointer arithmetic, casting nor deallocation
- Pure functional language

Pure functional languages

- Data flow is made explicit
- Value of expression depends only on its free variables
- Substitution of equals for equals is always valid
- Computational order is irrelevant

➔ Eases up evaluation especially with respect to property verification

Example divisor

- Take divide one term by another:

```
data Term = Con Int | Div Term Term

eval :: Term -> Int
eval (Con a) = a
eval (Div t u) = eval t ÷ eval u
```

When loosing state ...

- Example: exception

```
data M a = Raise Exception | Return a
type Exception = String

eval :: Term -> M Int
eval (Con a) = Return a
eval (Div t u)= case eval t of
                Raise e -> Raise e
                Return a -> case eval u of
                  Raise e -> Raise e
                  Return b -> if b = 0
                    then Raise "divide by zero"
                    else Return (a ÷ b)
```

➔ **Introduction of Monads**

TECHNISCHE UNIVERSITÄT DRESDEN

Monads

- Represent states resp. stores function results and side-effect representations
- Sequence operations
- are triples consisting of:
  - *type construction* defining for every underlying type the corresponding monadic type
  - *unit function* mapping values to values of the corresponding monadic type
  - *binding operation* $(M\ t) \rightarrow (t \rightarrow M\ u) \rightarrow (M\ u)$

Error monad

```
data M a = Raise Exception | Return a
type Exception = String

unit   :: a -> M a
unit a = Return a
(*) :: M a -> (a -> M b) -> M b
m * k = case m of
          Raise e -> Raise e
          Return a -> k a

raise :: Exception -> M  a
raise e = Raise e

eval :: Term -> M Int
eval (Con a)   = unit a
eval (Div t u) = eval t * λa.eval u * λb.
    (if b=0 then raise "..." else unit(a÷b))
```

OS construction

- I/O monad
- Foreign Function Interface (monads needed)

Problem:

- FFI insecure by using raw pointers (-) arithmetic and unsafe type casts

➔ Introduction of the 'hardware monad' H

➔ Specification & verification of certain monad's properties

Hardware monad – physical memory

- Physical pages abstraction (excluding heap) and allocation mechanism (garbage collected)

```
type Paddr = (PhysPage, Poffset)
type PhysPage
type POffset = Word12

pageSize = 4096 :: Int

allocPhysPage :: H (MayBe PhysPage)
getPAddr :: PAddr -> H Word8
setPAddr :: Paddr -> Word8 -> H ()
```

Hardware monad – virtual memory

- Allocating, writing and reading of (one-level) page maps including entries with typical properties

```
type PageMap
allocPageMap :: H (Maybe PageMap)

data PageInfo
  = PageInfo { physPage::PhysPage,
               writable, dirty,
               accessed::Bool }
    deriving (Eq,Show)
```

Hardware monad – 'Threads'

- Execution of code:

```
execContext :: PageMap -> Context
   -> H (Interrupt,Context)

data Context
   = Context { edi, esi, ebp, esp, ebx,
               edx, ecx, eax, eip,
               eflags :: Word32 }

data Interrupt = I_DivideError | ...
```

Hardware monad - I/O

- I/O ports and memory mapped I/O are supported
- Interrupt handling

```
data IRQ = IRQ0 | IRQ1 | ... | IRQ15
                deriving (Bounded, Enum)

enableIRQ, disableIRQ :: IRQ -> H()
enableInterrupts,disableInterrupts :: H()
```

Hardware monad - concurrency

- In Haskell RTS code can only be paused at 'safe points' (heap check points)
- When interrupts are raised in supervisor mode, two interrupt handling routines possible
  - explicit model: Haskell threads poll for interrupts
  - implicit model: using the signal handling mechanisms in the RTS
- Implicit concurrency breaks certain state-based assertions

P-Logic

- Extension of the Haskell language
- Used to describe properties of H and the underlying hardware, that can be verified by a theorem prover (Isabelle):
  - Allocations of pages, page maps etc. deliver distinct values
  - Bounds checking (e.g.: address space bound.)
  - Several non-interference properties (e.g.: page map entries, physical addresses, I/O ports ...)

Using H – Osker

- Oregon Separation Kernel: implementation of L4X2 using the hardware monad framework
- Goal: proving non-interference of concurrent processes
- Separating state in distinct components (different monads) limits the space one has to reason about (e.g. yield() example)

TECHNISCHE UNIVERSITÄT DRESDEN

Conclusions

- It works somehow, but we don't know how good
- It's far away from being formally verified
  - RTS: 90 000 LOC + libc
  - 7 MB kernel image
- Lack of evaluation at all, especially on the facilitation of verification
- Very x86 specific
- Nevertheless I like the idea and approach (OSKit)

Discussion

- Garbage collection within the kernel is it really a problem ?
- Don't we have so much state here, that the functional part gets negligible, maybe a pointer-safe language like Java is enough to facilitate formal proof ?

Further Reading

- 'Monads for functional programming' by Philip Wadler (Glasgow) 1993
  http://citeseer.ist.psu.edu/wadler95monads.html