Practical Haskell Programming:

Scripting With Types

Don Stewart | Amazon | 2011-02-04

Scripting with Types : Themes

- Introduction to thinking in functions
- Designing with types
  - An example: scripting privileged syscalls
- Engineering tools for reliability
- Further reading
- How Galois uses FP

$$(\text{Abs}) \quad \frac{\Gamma \vdash_{\mathrm{TY}} \sigma_x : \star \qquad \Gamma, x : \sigma_x \vdash_e e : \sigma}{\Gamma \vdash_e \lambda x{:}\sigma_x .\, e : \sigma_x \rightarrow \sigma}$$

Part 1. Shell scripting

Motivation: shell scripting

Shell scripts are the bread and butter of programming

- Don’t do manually what the computer could do for you
  - #!/bin/sh

- Often quick and dirty: get something running immediately
  - Developed in a compositional style: f | g | h > foo.txt
  - Often little attention paid to error handling
  - All data simply typed as strings

- Great for simple problems
- But fragile, slow and unwieldy as requirements change

Haskell is ...

- A functional language
- Strongly statically typed
- 21 years old
- Open source
- Compiled and interpreted
- Used in research, open source and industry

http://haskell.org
http://haskell.org/platform
http://hackage.haskell.org

Download Haskell

Haskell for shell scripting

In some ways, the opposite of shell scripting

- Built on generic, reusable abstractions
- Error handling up front
- Strong, static types with flexible type system

- But still concise! f . g . h

- Other benefits
  - Native code optimization (LLVM) and compilation
  - Parallel and concurrent runtime
  - Debuggers, profiling tools, refactoring, coverage, testing tools
  - Library support, distribution mechanisms

Goal: robust, maintainable scripting

Automate grunt work, to free you to tackle harder problems

Use solid software engineering practices for “scripting”

- Improve robustness of scripts over time
- Improve maintainability of scripts
- Improve performance of scripts

Focus is on scripts with an eye to long term use

- Use abstractions in code to model problem domain in scripts

Similar techniques apply for text encodings, SQL injections, string interpolation, ...

Simple example: CPU frequency scripting

Modern laptops have variable frequency CPUs

- Low clock speed: cooler machine, longer battery life
- High clock speed: fast code!

For benchmarking, I need to turn the CPU up to 11: no auto-scaling.

A shell script and Haskell program to toggle this behavior:

```
$ cpuperf                      $ cpuperf
 cpu: 0 -> 100                  cpu: 100 -> 0
  clock: 2.66 Ghz                clock: 1 Ghz
```

A shell program for frequency pinning

Host OS: OpenBSD

No /proc filesystem, instead “sysctls” are used

- Mutable variables in the kernel
- Read or set via the “sysctl” program:

```
$ sysctl kern.ostype
  kern.ostype=OpenBSD
$ sysctl hw.cpuspeed
  hw.cpuspeed=600
$ sysctl hw.setperf
  hw.setperf=0
```

Privileged mode

Setting sysctl variables means mutating a value in the running kernel process

This is a privileged operation: you must be root at the time.

Use “sudo” to gain elevated privileges.

Set these as only password-less operations:

```
dons NOPASSWD: /bin/sysctl –w hw.setperf=0
dons NOPASSWD: /bin/sysctl –w hw.setperf=100
```

A shell implementation

```
#!/bin/sh

s=`sysctl hw.setperf`
old=`echo $s | sed ‘s/.*=//’`
if [ “100” = $old ] ; then
    new=0
else
    new=100
fi

sudo sysctl –w hw.setperf=$new > /dev/null

printf “cpu: %d -> %d\n” $old $new
speed=`sysctl hw.cpuspeed`
clock=`echo $speed | sed ‘s/.*=//’`
clock2=`bc –l –e “$clock / 1000” –e quit`
printf “clock: %0.1f Ghz\n” %clock
```

Straight forward design

- Read the state of the world
- Perform pure logical operations on the data
- Render new data, driving external services:
  - Setting kernel variables
- Check that state of world matches what we think it is

- Red flags:
  - regular expressions used for parsing
  - no error handling
  - type confusion: numbers and strings mixed up
  - floating point math needs external services
  - root privileges are taken

Direct Haskell translation (imperative style)

```haskell
import Text.Printf
import Process

main = do
   s <- run “sysctl hw.setperf”
   let old = clean s
       new | old == 100  = 0
           | otherwise   = 100
   run $ “sudo sysctl -w hw.setperf=“ ++ show new
   printf “cpu: %d -> %d\n” old new

   s’ <- run “sysctl hw.setperf”
   let clock = fromIntegral (clean s’) / 1000
   printf “clock: %f Ghz\n” clock
 where
   clean = read . init . tail . dropWhile (/= ‘=‘)
```

Notes on imperative Haskell

- Regular expressions replaced with list manipulations
- Failure translated into (unhandled) exceptions
- IO operations and pure math are interleaved
- *Some* additional typing: strings, integers and doubles are distinguished

- Bytecode interpreter:

```
$ runhaskell naïve.hs
```

- Compiled:

```
$ ghc -O2 -make naïve.hs -o cpuperf
$ ./cpuperf
```

Part 2. Doing a better job: a DSL for sysctls

Code smells

A number of code smells are present

- The semantics of “sysctl” isn't obvious in the code
- No reuse of code possible
- No error handling strategy
- Fast and loose with privileges – could bite us in the future

General rule: **be domain specific**

- the abstractions in the program should model the abstractions of the problem domain

Domain specific shell code

Abstraction 1: sysctls are “variables” – mutable memory cells

And mutable cells have a well-known API:

Read a value from a box holding a’s:

```
get :: Box a -> m a
```

Write a value into a box:

```
set :: Box a -> a -> m ()
```

Take a box and a function over a’s, and mutate the box with that function

```
modify :: Box a -> (a -> a) -> m (a,a)
```

Concrete API for sysctls

|galois|

sysctls act as mutable cells storing integers, keyed by strings.

The cell API can be implemented as:

```
  get :: String -> IO Integer
  set :: String -> Integer -> Priv ()
```

And:

```
  modify :: String
         -> (Integer -> Integer)
         -> IO (Integer, Integer)
```

Implementation in Haskell

```
get s = do
  v <- run $ “sysctl” <+> s
  readM (parse v)
 where
  parse = tail . dropWhile (/= ‘=‘) . init

set s v = run $
  printf “sysctl -w %s=%s” s (show v)
```

Implementation … continued

```
modify s f = do
  v <- get s
  let u = f v
  set s u
  return (v, u)
```

Simpler code: a DSL for sysctls

We can now rewrite our script as:

```
toggle v   | v == 100    = 0
           | otherwise   = 100

main = do
  (old, new) <- modify "hw.setperf" toggle
  clock      <- get "hw.cpuspeed"
  printf "cpu: %d -> %d\n" old new
  printf "clock: %f Ghz\n" (clock / 1000)
```

Part 3: improving error handling

Exception handling

So far, both shell code and Haskell have ignored exceptions

E.g. data is parsed with a list processing pipeline:

```
parse
    = read
    . init
    . tail
    . dropWhile (/= '=')
```

However, malformed data will cause a parsing exception to be thrown.

Capturing errors in a monad

Rather than terminating our program with an exception, we'll lift errors into an abstraction: the Error monad

And then add custom error handling

Explicit parse errors:

```
readM s    | [x] <- parse       = return x
           | otherwise          = throwError (show s)
  where
      parse = [ x | (x,t) <- reads s ]
```

Now parse failures from malformed data will call the throwError method.

Error handling is overloaded

There are many different notions of error, so a type class is provided so that different sorts of errors can have a common interface

```
instance MonadError IO where
  throwError = ioError

instance MonadError e (Either e) where
  throwError = Left
```

Now, depending on the return type we choose, we can get custom error handling behavior

Checking for all possible errors

Now, we really should check the return value of every partial function – anything that might fail.

In Haskell, possibly-error values are tagged with the Either type: Right a | Left b

```
modify s f = do
  ev <- get s
  case ev of
      Left e    -> return (Left e)
      Right v   -> do
         let u = f v
         ev <- set s u
         case ev of
            Left e   -> return (Left e)
            Right _ -> return (Right (v,u))
```

Scrap your boilerplate!

Our beautiful program is now wrapped in obfuscating error handling boilerplate.

After each possibly-failing computation:

- check the result
  - then either propagate the failure up,
  - or pass a value further on.

- We have a particular blob of glue to run between every step.
- We probably have a monad!

There will be monads

Either forms a monad: the glue in our code

```
instance Monad (Either e) where
  return   = Right

  Left l  >>= _  = Left l
  Right r >>= k = k r
```

If one step ends in Left, discard the rest of the program, and return that Left value.

If one step ends in a Right, pass its result to the rest of the computation.

We have a programmable semi-colon!

Programmable semi-colons

Replace generic IO with custom error handling ';'

```
newtype Shell a =
     Shell { runShell :: ErrorT String IO a }

instance MonadError String Shell where
  throwError = error . ("Shell failed: " ++ )
```

Now, let's rewrite `modify' to handle errors better….

Abstracting out the error handling

```
modify s f = do
  ev <- get s
  case ev of
     Left e  -> return (Left e)
     Right v -> do
        let u = f v
        ev <- set s u
        case ev of
           Left e  -> return (Left e)
           Right - -> return (Right (v,u) )
```

Replacing with bind...

```
modify s f = do
  get s
     >>= \v -> do


        let u = f v
        set s u
          >>= \v -> do

                    return (v,u)
```

Using do-notation syntax

```
modify s f = do {
  v <- get s    ;
  let u = f v   ;
  v <- set s u ;
  return (v,u) ;
}
```

And using whitespace layout…

```
modify s f = do
  v <- get s
  let u = f v
  v <- set s u
  return (v,u)
```

Same code, but smarter

So we now have the same code as before for ‘modify’

- instead of unchecked errors in IO
- it has checked errors in a custom Shell monad

And different behavior:

- Before: parse error.
- After: cpuperf: checked errors:
  - `Shell failed: Failed parse: “hw.setperf=On”`

- All in the programmable ‘;’

Part 4. Privilege Separation

Embedding a problem in the type system

Escalating privileges is bad

Directly calling ‘sudo’ is a bit worrying.

In larger software abuse of root privileges leads to security holes, as it becomes hard to track what code fragments are running with elevated privileges.

Goal: full control of the scope of ‘sudo’

Goal 2: compile-time errors if ‘sudo’ is used in the wrong code

We will lean on the type system…

Standard trick: a tracking monad

Regular code has type:

```
Shell a
```

We'll make sure elevated code has type:

```
Priv a
```

A new monad for privileged actions:

```
newtype Priv a = Priv { priv :: Shell a }
```

And add custom error handling

A customized message for errors in privileged mode:

```
instance MonadError String Priv where
    throwError =
       error . ("Priv failed: " ++)
```

Smart constructors: only one way in

The key step now is to ensure the only way to construct a "Priv" value is to run sudo.

We will use a "smart constructor", and lift 'set' into Priv:

```
runPriv :: String -> Priv String
runPriv s = Priv $
  run ("/usr/bin/sudo" <+> s)

set :: String -> Integer -> Priv String
set s v = runPriv $
  printf "sysctl -w %s=%s\n" s (show v)
```

Now the type checker does the security audit!

Now calls to privileged code can only be made inside a 'priv' tag:

```
Main.hs: 66:4:
  Couldn’t match expected type `Shell t’
     against inferred type `Priv String’
```

Unsafe calls to ‘set’ are type errors:

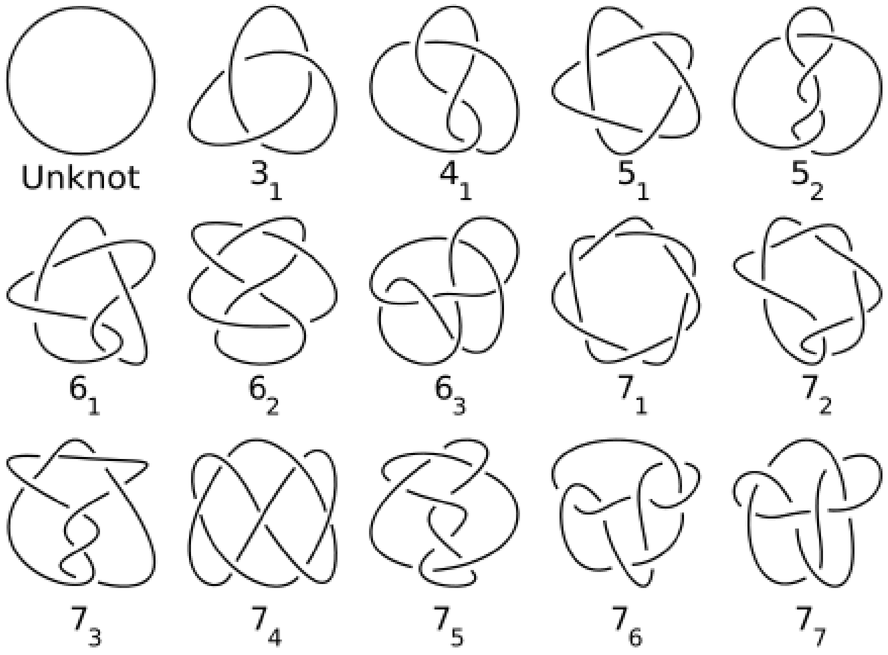
```
  set s u
```

becomes

```
  priv (set s u)
```

And the scope of privileges is lexically visible

Part 5. Summary



Shell programming as case-study of DSLs

A throw-away script may get used over and over.

Even simple code contains many subtle behaviors

Good languages let us embed the problem domain semantics in the host language semantics.

And we can use standard tools:

- types
- libraries
- abstractions
- compilation, optimizations…

to make code more precise, safer and more reliable

Similar tools

Other functional languages

- Statically Typed: OCaml – everything in the IO monad
- Dynamically Typed: Erlang, Scheme, Clojure

Hybrid languages:

- JVM: Scala – statically typed, with OO features
- .NET: F# – OCaml subset with some Haskell on .NET.

Further reading

Real World Haskell:

http://book.realworldhaskell.org/

Example embedded domain specific languages:

- Orc - concurrent workflow scripting
- Accelerate – GPU array processing
- atom – hard real-time, safety critical controllers
- HJScript – typed JavaScript in Haskell
- haskore – music notation in Haskell
- Feldspar – digital signal processing

All on http://hackage.haskell.org

And what does Galois do?

Portland-based research and development contracting for critical systems. We love FP languages. http://galois.com

Virtualization: HaLVM (GHC runtime on Xen for tiny, domain-specific operating systems)

– mobile devices, cloud nodes

Identity and Security: IdM systems for grid systems, and networks with multiple security levels

Embedded Systems: CoPilot, a DSL for writing hard real-time control systems code

– aerospace, automotive, industrial apps

Cryptography: Cryptol, a DSL for doing cryptography

– prove algorithms correct, generate fast code