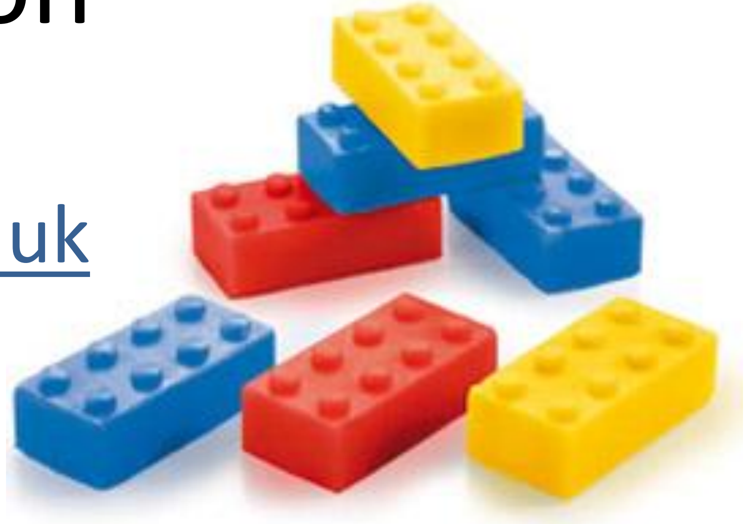
Building stuff with monadic dependencies + unchanging dependencies + polymorphic dependencies + abstraction

Neil Mitchell

http://nmitchell.co.uk

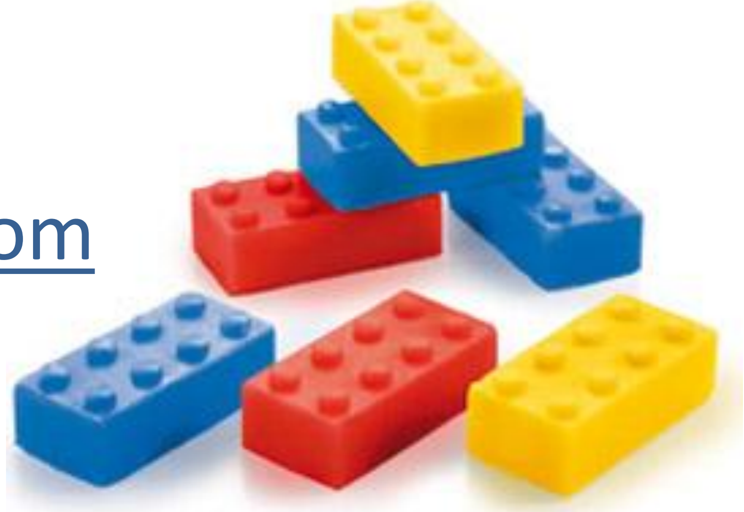


Building stuff with

Shake

Neil Mitchell

http://shakebuild.com



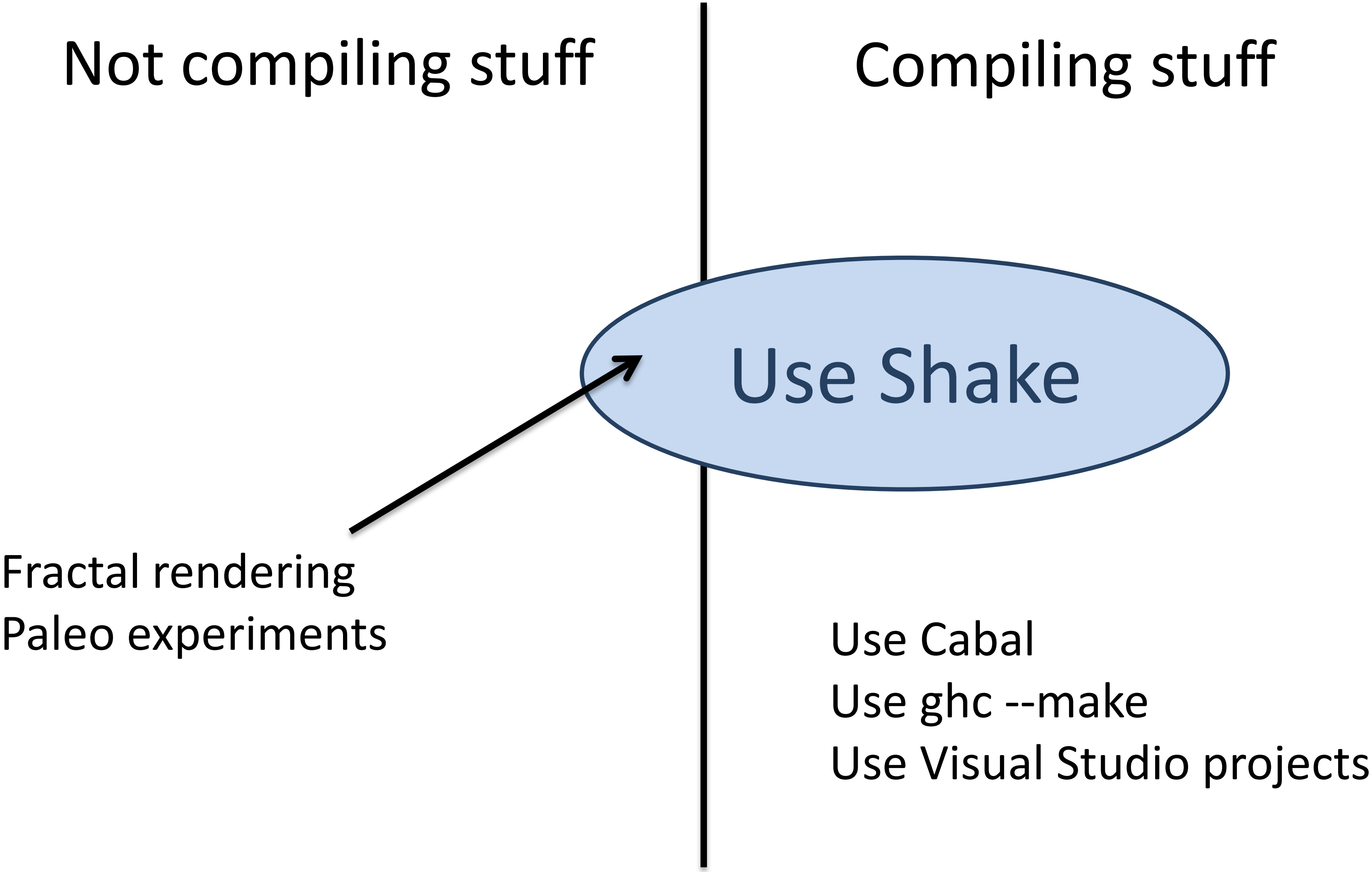
What is Shake?

- A Haskell library for writing build systems
  - Alternative to make, Scons, Ant, Waf…

- I wrote it at Standard Chartered in 2009
- I rewrote it open-source in 2012

Who has used Haskell? Shake?

When to use a build system



Not compiling stuff

Compiling stuff

Use Shake

Fractal rendering
Paleo experiments

Use Cabal
Use ghc --make
Use Visual Studio projects

Tutorial Overview

- Tutorial rules
  - Ask if you don’t understand
  - There is no end – I stop when the clock hits 0
  - All slides will be online
  - Not a “sales pitch”
  - Questions for you *in italic on most slides*.
- One main example (compiling a C file)
- Lots of independent extensions to that

Main example

Some C files

```c
/* main.c */
#include <stdio.h>
#include "a.h"
#include "b.h"
void main() {
    printf("%s %s\n",a,b);
}
```

```c
/* a.h */
char* a = "hello";

/* b.h */
char* b = "world";
```

What does this print?

Main example Compiling C

```
gcc -c main.c
gcc main.o -o main
```

What files are involved at each step?

Main example

Compiling C in Haskell

```
import Development.Shake

main = do
    () <- cmd "gcc -c main.c"
    () <- cmd "gcc main.o -o main"
    return ()
```

Why do we have the ugly () <- line noise?

Main example

A Shake system

Boilerplate

```
import Development.Shake
import Development.Shake.FilePath

main = shakeArgs shakeOptions $ do
   want ["main" <.> exe]
   "main" <.> exe %> \out -> do
      () <- cmd "gcc -c main.c"
      () <- cmd "gcc main.o -o main"
      return ()
```

When will main.exe rebuild?

Main example

With dependencies

```
want ["main" <.> exe]
"main" <.> exe %> \out -> do
    need ["main.c", "a.h", "b.h"]
    () <- cmd "gcc -c main.c"
    () <- cmd "gcc main.o -o main"
    return ()
```

Why is this a bad idea?

Main example

Asking gcc for depends

```
$ gcc -MM main.c
main.o: main.c a.h b.h
```

Anyone used that before?

Using gcc -MM

```
import Development.Shake.Util

"main" <.> exe %> \out -> do
    Stdout s <- cmd "gcc -c -MM main.c"
    need $ concatMap snd $ parseMakefile s
    () <- cmd "gcc main.o -o main"
    return ()
```

Did you know you can combine -c and -MM?

Main example

Two rules

```
"main.o" %> \out -> do
    Stdout s <- cmd "gcc -c -MM main.c"
    need $ concatMap snd $ parseMakefile s

"main" <.> exe %> \out -> do
    need ["main.o"]
    cmd "gcc main.o -o main"
```

Why are two rules better?

Main example

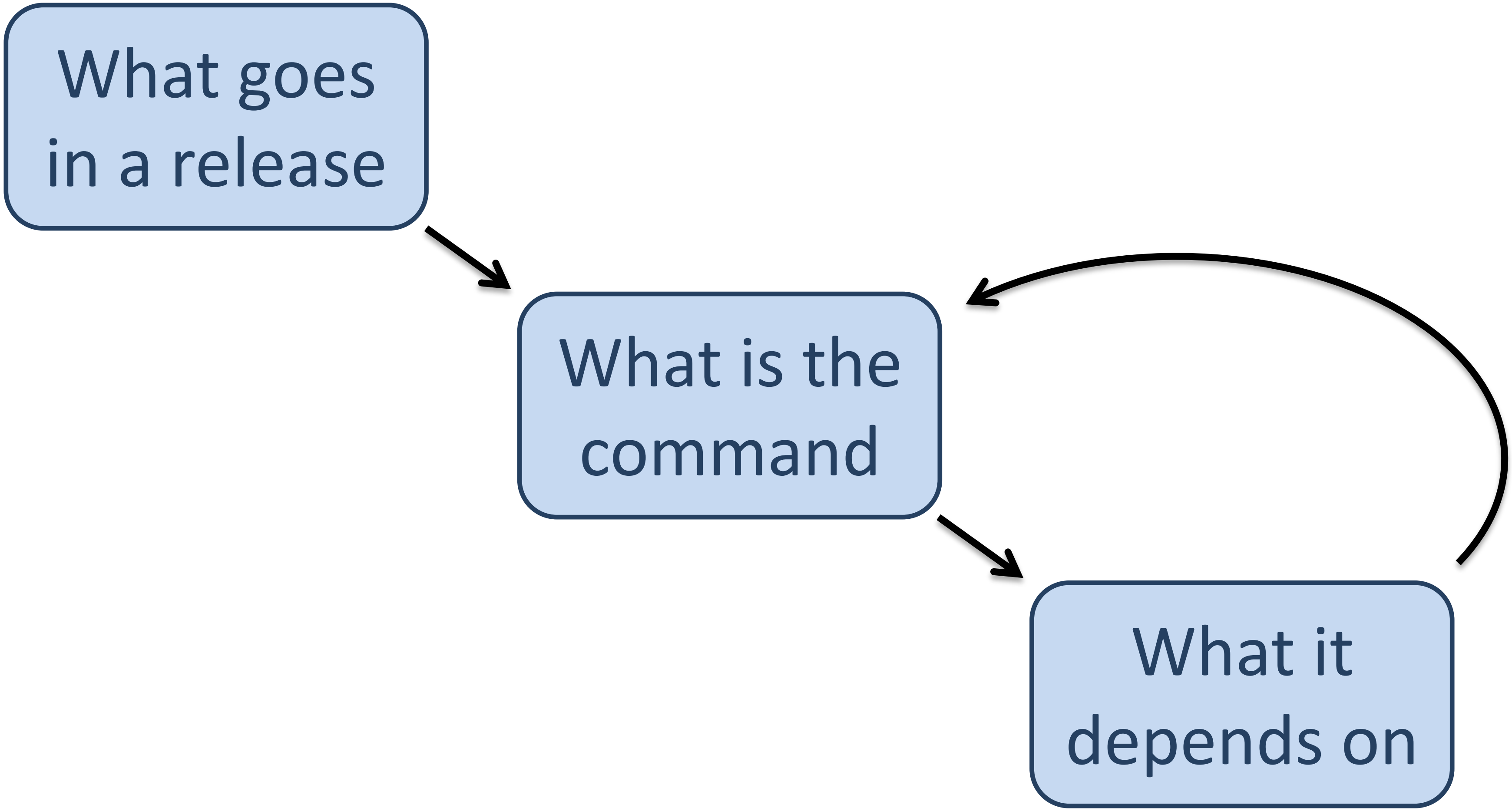
The result

```
main = shakeArgs shakeOptions $ do
    want ["main" <.> exe]

    "main" <.> exe %> \out -> do
        need ["main.o"]
        cmd "gcc main.o -o main"

    "main.o" %> \out -> do
        Stdout s <- cmd "gcc -c -MM main.c"
        need $ concatMap snd $ parseMakefile s
```

The “perfect” build system

- A bunch of wants
  - Each thing that goes in the release
- A bunch of rules
  - Simple pattern
  - A bunch of need, a bit of Haskell
  - A single command line (occasionally two)

Your thoughts



What goes in a release

What is the command

What it depends on

File patterns

Any file

```
"*.o" %> \out -> do
    let src = out -<.> "c"
    Stdout s <- cmd "gcc -c -MM" [src]
    need $ concatMap snd $ parseMakefile s
```

Why do we use [src], not just src?

Source to object

```
"obj//*.o" %> \out -> do
    let src = "src" </> dropDirectory1 out -<.> "c"
    Stdout s <- cmd "gcc -c -MM" [src] "-o" [out]
    need $ concatMap snd $ parseMakefile s
```

What if we want to do lower-case files?

Pattern predicates

```
(\x -> all isLower (takeBaseName x) &&
        "*.o" ?== x) ?> \out -> do
    let src = out -<.> "c"
    Stdout s <- cmd "gcc -c -MM" [src]
    need $ concatMap snd $ parseMakefile s
```

What can’t we do?

Dependencies on $PATH

```
"main" <.> exe %> \out -> do
    need ["main.o"]
    cmd "gcc main.o -o main"
```

- We depend on the version of gcc on $PATH
    - But we don’t track it

What else don’t we track?

Version deps

Store gcc version

```
"gcc.version" %> \out -> do
    alwaysRerun
    Stdout s <- cmd "gcc --version"
    writeFileChanged out s
```

What if we didn’t use writeFileChanged?

Version deps

Depending on gcc version

```
"main" <.> exe %> \out -> do
    need ["main.o", "gcc.version"]
    cmd "gcc main.o -o main"
```

Are two need’s after each other equivalent?

Compile all files in a dir

```
"main" <.> exe %> \out -> do
    need ["main.o"]
    cmd "gcc main.o -o main"
```

- Compile in all .c files in a directory

Do we already have enough to do that?

Dir contents

getDirectoryFiles

```
"main" <.> exe %> \out -> do
   xs <- getDirectoryFiles "" ["*.c"]
   let os = map (-<.> "o") xs
   need os
   cmd "gcc" os "-o main"
```

What if we want to find all files recursively?

The four features

1. Monadic (dynamic?) dependencies
2. Unchanging dependencies
3. Polymorphic dependencies
4. Abstraction

Where have we used each so far?

#1: Monadic dependencies

- Ask for further dependencies at any point
  - The need doesn’t have to be on the first line
- Absolutely essential
- Found in Shake (+clones), Redo, a bit in Scons

- Every non-monadic build system has hacks to get some monadic power
  - None are direct and powerful

#2: Unchanging dependencies

- A dependency may rebuild, but not change
- Very important to reduce rebuilds
  - Allows writeFileChanged, depending on gcc
- More common, but not in make, not a default
  - Ninja = restat, Tup = ^o^
  - Redo = redo-ifchange
  - Requires a database of metadata

#3: Polymorphic dependencies

- Dependencies don’t have to be files
- If you have monadic + unchanging, polymorphic is no new power
  - Just more convenient, avoid on-disk files

- Quite rare, only Shake that I know of
  - (Redo has redo-ifcreate)

#4: Abstraction

- Mostly a DSL vs EDSL question
  - Custom languages usually lack abstraction
  - Almost always lack package managers
- Monadic also makes abstraction easier
  - Shake has about 7 released packages of rules
  - Other build systems don’t seem to share as much
- Available in Scons, Shake, a few others

Generate .c

Generate the .c file

```
"main.c" %> \out -> do
    need ["main.txt"]
    cmd Shell "generate main.txt > main.c"
```

Where is the bug?

Generate .c

Generate the .c file

```
"*.o" %> \out -> do
    let src = out -<.> "c“
    need [src]
    Stdout s <- cmd "gcc -c -MM" [src]
    needed $ concatMap snd $ parseMakefile s
```

Is there a way to fix gcc -MM directly?

Avoid gcc -M

Manual header scan

```
usedHeaders :: String -> [FilePath]
usedHeaders src =
    [ init x
    | x <- lines src
    , Just x <- [stripPrefix "#include \"" x]]
```

What’s the disadvantage of a manual scan?

Avoid gcc -M

Manual header scan

```
"main.o" %> \out -> do
    src <- readFile' "main.c"
    need $ usedHeaders src
    cmd "gcc -c main.c"
```

What’s the advantage of a manual scan?

Generate the .h file

```
"*.h" %> \out -> do
    let src = out -<.> "txt"
    need [src]
    cmd Shell "generate" [src] ">" [out]
```

What made this change self-contained?

Transitive

One-step includes

```
["*.c.dep","*.h.dep"] |%> \out -> do
    src <- readFile' $ dropExtension out
    writeFileLines out $ usedHeaders src
```

What are we reusing?

Transitive includes

```
"*.deps" %> \out -> do
    dep <- readFileLines $ out -<.> "dep"
    deps <- mapM (readFileLines . (<.> "deps")) dep
    writeFileLines out $ nub $
        dropExtension out : concat deps
```

deps a = a : concatMap deps (dep a)

Transitive includes

```
"main.o" %> \out -> do
    src <- readFileLines "main.c.deps"
    need src
    cmd "gcc -c main.c"
```

How could we test this rule?

Config

Define config

- Keep regularly changing details out of .hs

```
# build.cfg
main.exe = main foo
config.exe = config foo
```

Is this easy enough for Haskell-phobes?

Config

Interpret config

```
import Development.Shake.Config

usingConfigFile "build.cfg"
action $ need =<< getConfigKeys

"*.exe" %> \out -> do
    Just src <- getConfig out
    let os = map (<.> "o") $ words src
    need os
    cmd "gcc" os "-o" [out]
```

What else might we put in the config?

Resources

What is a resource?

- Build systems allocate CPU resources
- What about *other* resources?

- Only have 12 licenses for the FPGA tester
- Can only run one copy of Excel at a time

What are some other resources?

Using resources

```
disk <- newResource "Disk" 4
"*.exe" %> \out ->
    withResource disk 1 $
        cmd "gcc -o" [out] ...
```

What is the performance impact?

Command line flags

$ runhaskell Main.hs --help

```
Usage: shake [options] [target] ...
Options:
  -B, --always-make            Unconditionally make all targets.
  --no-build                   Don't build anything.
  --color, --colour            Colorize the output.
  -d[=FILE], --debug[=FILE]    Print lots of debugging information.
  -j[=N], --jobs[=N]           Allow N jobs/threads at once [default CPUs].
  -k, --keep-going             Keep going when some targets can't be made.
  -l, --lint                   Perform limited validation after the run.
  --live[=FILE]                List the files that are live [to live.txt].
  --assume-skip                Don't remake any files this run.
  -p[=N], --progress[=N]       Show progress messages [every N secs, default 5].

... 57 lines in total ...
```

Flags vs options

```
opts = shakeOptions{shakeThreads=8}
main = shakeArgs opts …

$ runhaskell Main.hs -j5
```

Who wins? Developer or user?

Flags

Named arguments

```
phony "clean" $ do
    removeFilesAfter ".shake" ["//*"]
```

Why removeFilesAfter?

Extra flags

```
data Flags = DistCC
flags = Option "" ["distcc"]
    (NoArg $ Right DistCC)
    "Run distributed."

main = shakeArgsWith shakeOptions [flag] …
```

What do non-flags args do by default?

Also files

Many-out

```
["*.o","*.hi"] &%> \[o,hi] -> do
    let hs = o -<.> "hs"
    need ... -- all files the .hs import
    cmd "ghc -c" [hs]
```

Could we avoid &%> ?

Lint rules

- Enable by passing --lint
  - Don’t change current directory
  - Files written only once
  - Files not used before need
- Enabled by passing --lint-tracker
  - Dependencies are not used without need

What others?

Lint rules

```
"main.o" %> \out -> do
    Stdout s <- cmd "gcc -c -MM main.c"
    needed $ concatMap snd $ parseMakefile s
```

When is needed safe?

Error:
Out of slides