Writing Shake Rules

Neil Mitchell

https://github.com/ndmitchell/shake

http://shakebuild.com

Shake build system

Expressive, Robust, Fast

Haskell EDSL
Monadic
Polymorphic
Unchanging

1000's of tests
100's of users
Heavily used

Faster than
Ninja to
build Ninja

Simple example

```
out : in
    cp in out
```

```
(%>) :: FilePattern -> (FilePath -> Action ()) -> Rule ()
```

```
"out" %> \out -> do
    need ["in"]
    cmd "cp in out"
```

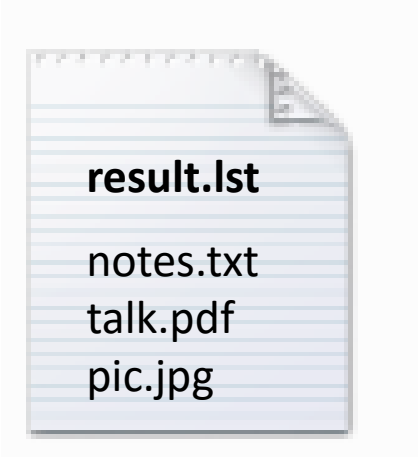
:: Action ()
Monad Action

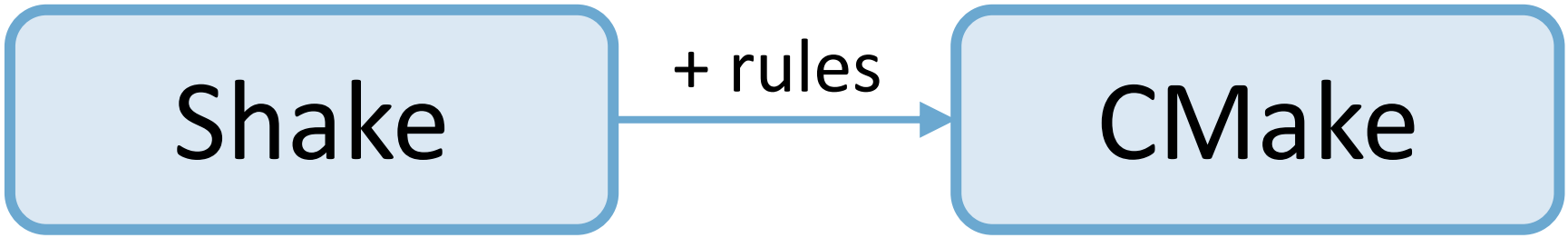
:: Rule ()
Monad Rule

Longer example

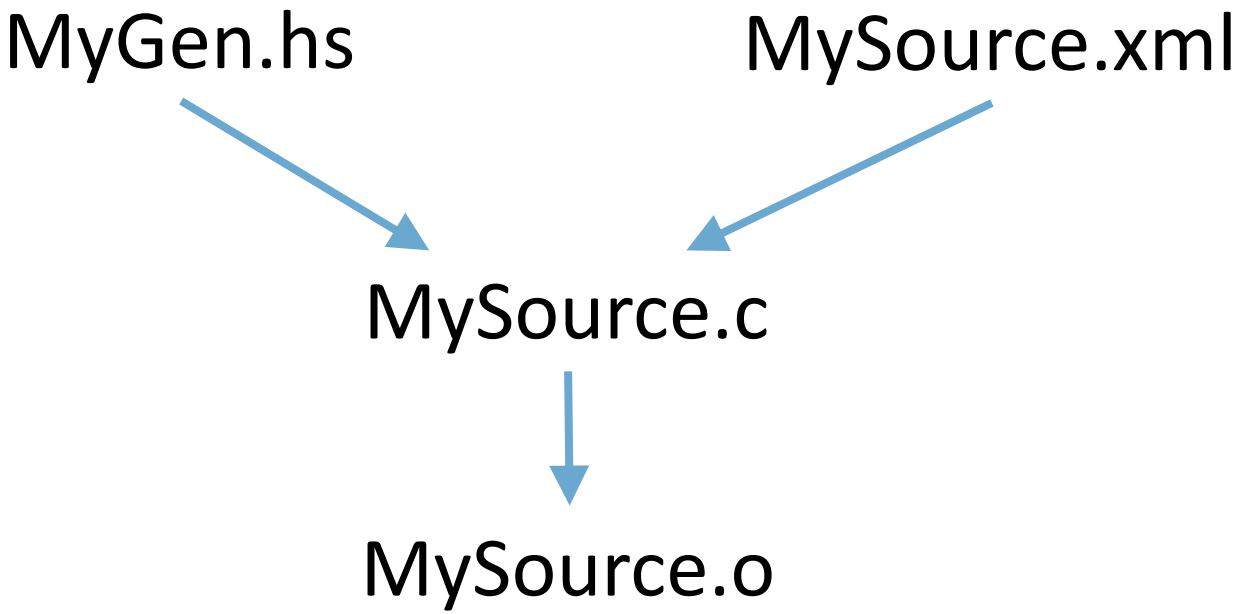
```
import Development.Shake
import Development.Shake.FilePath

main = shakeArgs shakeOptions $ do
  want ["result.tar"]
  "*.tar" %> \out -> do
    need [out -<.> "lst"]
    contents <- readFileLines $ out -<.> "lst"
    need contents
    cmd "tar -cf" [out] contents
```




Shake
+ rules
CMake

Generated files



What does MySource.o depend on?

Generated approaches

- Hardcode it?
  - Very fragile.
- Hack an approximation of MyGen?
  - Slow, somewhat fragile, a lot of effort.
- Run MyGen.hs and look at MySource.c
  - Easy, fast, precise.
  - Requires *monadic* dependencies

Monadic dependencies

Determine future dependencies
based on the results
of previous dependencies

Monadic dependencies in code

```
"MyHeader.h" %> \out -> do
  need ["MyGen.hs","MyHeader.xml"]
  cmd "runhaskell MyGen.hs"

"MySource.o" %> \out -> do
  need =<< readFile' "MySource.c.deps"
  cmd "gcc -c MySource.c"
```

See later for .deps rule

Polymorphic dependencies

- Can dependency track more than just files

```
"_build/run" <.> exe %> \out -> do
  link <- fromMaybe "" <$> getEnv "C_LINK_FLAGS"
  cs <- getDirectoryFiles "" ["//*.c"]
  let os = ["_build" </> c -<.> "o" | c <- cs]
  need os
  cmd "gcc -o" [out] link os
```

Using Shake for our build system has been a very good decision so far, we've been able to minimise the time spent with platform-dependent build systems and IDEs and get to write Haskell code instead ;)

Stefan Kersten, CTO Samplecount
Cross-platform music stuff in C/Haskell
Using Shake for > 2 years

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";
```

Compiling C

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

What files are involved at each step?

Compiling C with Shake

```
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 ()
```

Asking gcc for depends

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

Asking gcc with Shake

```
"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"
```

Manual header scan

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

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

Transitive header scan: depth 1

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

Transitive header scan: depth *

```
"*.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 header scan

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

What should a .c rule look like?

- Scan manually?
- Use gcc -M?
  - What if it can’t see a not-yet generated header?
  - Fixed point? GHC build system is doing that
- Make the user manually specify generated files?
- Configuration options? $CFLAGS? Output dir?
- Prior art: shake-language-c, shake-cpp and hadrian
- What about other rule types?