Shake: Past, Present, Future

Neil Mitchell
shakembuild.com
Shake: a build system

- An alternative to Make, as a Haskell library
- About 9 years old
  - Built my PhD thesis
  - Proprietary SCB build system
  - Open-source reimplementation
  - Use in GHC
  - Research applications
PhD thesis builder

\[
(\leqslant) :: \text{FilePath} \to [\text{FilePath}] \to (\text{FilePath} \to \text{FilePath} \to \text{IO ()}) \to \text{IO ()}
\]

\[
(\leqslant) \text{ to froms @(from: _ ) action } = \text{ do }
\]

\[
b \leftarrow \text{ doesFileExist to}
\]

\[
\text{rebuild } \leftarrow \text{ if not } b \text{ then return } \text{True else do }
\]

\[
\text{from2 } \leftarrow \text{ liftM maximum } $ \text{ mapM getModificationTime froms}
\]

\[
\text{to2 } \leftarrow \text{ getModificationTime to}
\]

\[
\text{return } $ \text{ to2 } < \text{ from2}
\]

\[
\text{when rebuild } $ \text{ do }
\]

\[
\text{putStrLn } $ \text{ "Building: " ++ to }
\]

\[
\text{action from to}
\]
Shake: A Better Make

Neil Mitchell, Standard Chartered
Haskell Implementors Workshop 2010

OLD SLIDES: I’m no longer at Standard Chartered
import Development.Shake

main = shake $ do
    want ["Main.exe"]
    "Main.exe" *> \x -> do
        cs <- ls "*.c"
        let os = map (\`replaceExtension\` "obj") cs
        need os
        system $ ["gcc","-o",x] ++ os
    "*.obj" *> \x -> do
        let c = replaceExtension x "c"
        need [c]
        need =<< cIncludes c
        system ["gcc","-c",c,"-o",x]
Benefits of Shake

- A Haskell library for writing build systems
  - Can use modules/functions for abstraction/separation
  - Can use Haskell libraries (i.e. filepath)

- It’s got the useful bits from Make
  - Automatic parallelism
  - Minimal rebuilds

- But it’s better!
  - More accurate dependencies (i.e. the results of ls are tracked)
  - Can produce profiling reports (what took most time to build)
  - Can deal with generated files properly
  - Properly cross-platform
The Oracle

- The Oracle is used for non-file dependencies
  - What is the version of GHC? 6.12.3
  - What extra flags do we want? --Wall
  - ls is a sugar function for the Oracle

```haskell
type Question = (String, String)
type Answer = [String]
oracle :: (Question -> Answer) -> Shake a -> Shake a
query :: Question -> Act Answer
```
NO DEPENDENCY GRAPH!
Parallelisation

- need/want both take lists of files, which run in parallel

- Try and build N rules in parallel
  - Done using a pool of N threads and a work queue
  - need/want put their jobs in the queue

- Add a Building (MVar()) in DataBase

- Shake uses a random queue
  - Jobs are serviced at random, not in any fair order
  - link = disk bound, compile = CPU bound

- Shake is highly parallel (in theory and practice)
Profiling

- Can record every system command run, and produce:

```
Average was 3.42 (interquartiles were 4.00 and 4.00)

Hot Tools
7.7s 7 x gcc

Hot Commands
1.8s gcc -c file3.c -o file3.obj
1.8s gcc -c file2.c -o file2.obj
```
Practical Use

- Relied on by an international team of people every day
- Building more than a million lines of code in many languages

Before Shake
- Masses of really complex Makefiles, slow builds
- Answer to any build error was “make clean”

After Shake
- Robust and fast builds (at least x2 faster)
- Maintainable and extendable (at least x10 shorter)
Limitations/Disadvantages

- Creates a _database file to save the database
- Oracle is currently “untyped” (String’s only)
  - Although easy to add nicely typed wrappers over it
- Massive space leak (~ 12% productivity)
  - In practice doesn’t really matter, and should be easy to fix
- More dependency analysis tools would be nice
  - Changing which file will cause most rebuilding?
- What if the rules change?
  - Can depend on Makefile.hs, but too imprecise
- Not currently open source
Shake Before Building
Replacing Make with Haskell

Neil Mitchell

community.haskell.org/~ndm/shake
Generated files

- MyGenerator
- Foo.xml
- Foo.c
- ...headers...
- Foo.o

• What headers does Foo.c import?
  (Many bad answers, exactly one good answer)
Dependencies in Shake

"Foo.o" *> \_ -> do
  need ["Foo.c"]
  (stdout, _) <- systemOutput "gcc" ["-MM","Foo.c"]
  need $ drop 2 $ words stdout system' "gcc" ["-c","Foo.c"]

- Fairly direct
  - What about in make?
Make requires phases

Foo.o : Foo.c
  gcc -c Foo.o

Foo.o : ($(shell sed ... Foo.xml)

Foo.mk : Foo.c
  gcc -MM Foo.c > Foo.mk
#include Foo.mk

Disclaimer: make has hundreds of extensions, none of which form a consistent whole, but some can paper over a few cracks listed here
Dependency differences

• Make
  – Specify all dependencies *in advance*
  – Generate static dependency graph

• Shake
  – Specify additional dependencies *after* using the results of previous dependencies

\[ D_{\text{shake}} > D_{\text{make}} \]
A build system with a static dependency graph is insufficient
Build system + Haskell

Better dependencies
Modern engineering

Parallelism
Robustness
Efficient

Profiling
Lint
Analysis

Syntax
Types
Abstraction
Libraries
Monads

Shake
Profiling

Identical performance to make
Featureful, 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
```
• Assume you change whitespace in MyHeader.xml and MySource.c doesn’t change
  – What rebuilds?
  – What do you want to rebuild?
  – (Very common for generated code)
Unchanging consequences

• Assume you change whitespace in MyHeader.xml
  – Using file hashes: MyGen.hs runs and nothing
  – Using modtimes: Stops if MyGen.hs checks for Eq first

• Always build children before their parents

• What if a child fails, but the parent changed to no longer require that child?
  – Must rebuild the parent and fail on demand
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" UITableViewCell lessen "o" | c <- cs]
  need os
  cmd "gcc -o" [out] link os
```
Polymorphic dependencies

• About 7 built in Rule instances

```haskell
type ShakeValue a = (Show a, Typeable a, Eq a,
                     Hashable a, Binary a, NFData a)

class (ShakeValue k, ShakeValue v) => Rule k v where
  storedValue :: k -> IO (Maybe v)
```
Progress prediction

• Guesses how long the build will take
  – 3m12s more, is 82% complete
  – Based on historical measurements plus guesses
  – All scaled by a progress rate (guess at parallel setting)
  – An approximation...
Why is Shake fast?

• What does fast even mean?
  – Everything changed? Rebuild from scratch.

• In practice, a blend, but optimise both extremes and you win
Fast when everything changes

• If everything changes, rule dominate (you hope)
• One rule: Start things *as soon as you can*
  – Dependencies should be fine grained
  – Start spawning before checking everything
  – Make use of multiple cores
  – Randomise the order of dependencies (~15% faster)

• Expressive dependencies, Continuation monad, cheap threads, immutable values (easy in Haskell)
Fast when nothing changes

- Don’t run users rules if you can avoid it
- Shake records a journal, $[(k, v, ...)]$

```plaintext
unchanged journal = flip allM journal $ \{(k,v) \rightarrow$
  $(== \text{Just } v) <$ storedValue $\} \to storedValue k$
```

- Avoid lots of locking/parallelism
  - Take a lock, check storedValue a lot
- Binary serialisation is a bottleneck
Non-recursive Make Considered Harmful: Build Systems at Scale

Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, Simon Marlow

Haskell Symposium 2016
The GHC and the build system

Glasgow Haskell Compiler:
- 25 years old
- 100s of contributors
- 10K+ source files
- 1M+ lines of Haskell code
- 3 GHC stages
- 18 build ways
- 27 build programs: alex, ar, gcc, ghc, ghc-pkg, happy, ...

The current build system:
- Non-recursive Make
- Fourth major rewrite
- 200 makefiles
- 10K+ lines of code
- 3 build phases
- Highly user-customisable
- And it works! But...
The result of 25 years of development

$1/2/build/%.$3_osuf) : $1/4/%.hs $(LAX_DEPS_FOLLOW) \n $(1_2_HC_DEP) $(1_2_PKGDATA_DEP)
$(call cmd,1_2_HC) $(1_2_3_ALL_HC_OPTS) -c $< -o @@ \n$(if $(findstring YES,$1_2_DYNAMIC_TOO)), \n-dyno $(addsuffix .$(dyn_osuf),$(basename @@)))
$(call ohi-sanity-check,$1,$2,$3,$1/2/build/**)

Make uses a global namespace of mutable string variables
- Numbers, arrays, associative maps are encoded in strings
- No encapsulation and implementation hiding
- Variable references are spliced into Makefiles: avoid spaces/colons
- To expand a variable use $; to get $ use $$; to get $$ use $$$$...
There are other problems

1. A global namespace of mutable string variables
2. Dynamic dependencies
3. Build rules with multiple outputs
4. Concurrency reduction
5. Fine-grain dependencies
6. Computing command lines, essential complexity

Solution: use FP to design scalable abstractions
– To solve 1-5: we use Shake, a Haskell library for writing build systems
– To solve 6: we develop a small EDSL for building command lines

Accidental complexity
Build rules with multiple outputs

"*.o" \%> \obj  ->  do
  let src = obj -><.> "hs"
  need [src]  
  run "ghc" [src]

["*.o", ".hi"] \%> \[obj, hi\]  ->  do
  let src = obj -><.> "c"
  need [src]  
  run "ghc" [src]

How do we tell our build system that 

ghc produces both 

*.o and *.hi files?
Concurrency reduction

"//*.conf" %> \
\conf -> do
let src = confSrcFile conf
need [src]
run "ghc-pkg" ["update",
src]

But we can have at most one ghc-pkg running at a time as it mutates package db!

db <- newResource "package-db" 1

"//*.conf" %> \
\conf -> do
let src = confSrcFile conf
need [src]
withResource db 1 $ run "ghc-pkg" ["update",
src]
Dynamic dependencies

Build target \( t \):
- Lookup \( t \)'s dependencies \( \{d_1, \ldots, d_n\} \) in the database
- If the lookup fails or \( t \) doesn’t exist or \( t \) has changed or some \( d_k \) is not up to date then
  - Find the build rule matching \( t \)
  - Run the action, recording need’s
  - Update the database with newly recorded dependencies
More quick wins with Shake

Post-use dependencies
Order-only dependencies
Polymorphic/fine-grain dependencies
Tracking file contents
Avoiding external tools
...
Read the paper!
Each invocation of a builder is fully described by a \textit{target}

\begin{verbatim}
data Target = Target
  { context :: Context
    , builder :: Builder
    , inputs :: [FilePath]
    , outputs :: [FilePath] }

preludeTarget = Target
  { context = Context Stage1 base profiling
    , builder = Ghc Stage1
    , inputs = ["libraries/base/Prelude.hs"]
    , outputs = ["build/stage1/libraries/base/Prelude.p_o"] }
\end{verbatim}
Computing command line for a target

Given `preludeTarget` how to compute the build command for it?

```haskell
preludeTarget = Target
  { context = Context Stage1 base profiling,
    builder = Ghc Stage1,
    inputs = "libraries/base/Prelude.hs",
    outputs = "build/stage1/libraries/base/Prelude.p_o"
  }

commandLine :: Target -> Action [String]
```

```
inplace/bin/ghc-stage1 -O2 -prof -c libraries/base/Prelude.hs
    -o build/stage1/libraries/base/Prelude.p_o
```
An expression $\textbf{Expr} \ a$ is a computation that produces a value of type $\textbf{Action} \ a$ and can read the current build $\textbf{Target}$:

```haskell
type $\textbf{Expr} \ a$ = ReaderT $\textbf{Target}$ $\textbf{Action} \ a$

ghcArgs :: $\textbf{Expr} \ [\text{String}]$
ghcArgs = do
    target <- ask
    return $ [ "-O2" ]
    ++ [ "-prof" | way (context target) == profiling ]
    ++ [ "-c", head (inputs target) ]
    ++ [ "-o", head (outputs target) ]
```

An expression $\textbf{Expr} \ a$ is a computation that produces a value of type $\textbf{Action} \ a$ and can read the current build $\textbf{Target}$:
Current limitations

We can build stage 2 GHC, but still lack many features:

- We only build *vanilla* and *profiling* way
- Validation is not implemented
- Only HTML Haddock documentation is supported
- Not all build flavours are not supported
- Cross-compilation is not implemented
- No support for installation or binary/source distribution
- 46 open issues:
  https://github.com/snowleopard/hadrian/issues
Experiments

Qualitative analysis:

- We studied 11 common use-cases of GHC build system, such as “edit a source file and rebuild”, “add a new build command line argument and rebuild”, “git branch and rebuild”, etc.
- The old build system performs a lot of unnecessary rebuilds in many cases, whereas Hadrian correctly handles most cases.

Quantitative benchmarks: Hadrian is faster

- Zero build: 2.2s vs 2.0s (Linux), 12.3s vs 2.1s (Windows)
- Full build: 649s vs 578s (Linux), 1266s vs 737s (Windows)
Build GHC
Future directions – better API

- After 9 years, I’m still improving the API
  - Currently working on a rewrite for defining rule types
  - Makes rules faster and more powerful
  - Use type families to assert rule relationships
Future directions – tracing

- What if we could track every file accessed?
  - Lint checks
  - Automatic dependencies

- Requires cross-OS tracing primitives
Future directions – forward

import Development.Shake
import Development.Shake.Forward
import Development.Shake.FilePath

main = shakeArgsForward shakeOptions $ do
    contents <- readFileLines "result.txt"
    cache $ cmd "tar -cf result.tar" contents
Future directions – cloud

- “Towards Cloud Build Systems with Dynamic Dependency Graphs”

- Aka, Google scale, better dependencies
  - Compete with Bazel/Buck