Distributed Build Systems

Neil Mitchell
@ndm_haskell
https://ndmitchell.com
A simple build system

```
main.exe : main.o
  gcc -o main.exe main.o

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

Make, 1976
(42 years ago, 12BG)
A build system performs necessary actions, respecting dependencies.

We focus on general-purpose build systems.
Build systems

- Excel
- Buck
- Make
- Buck
- Ninja
- Pants
- Shake
- Bazel
- Nix
- Hippo
Build Systems à la Carte

Engineering +

necessary actions

dirty

verifying trace

constructive trace

deterministic constructive trace

topological

Restart

suspend

respecting dependencies

Make

Ninja

Pants

Buck

Excel

Bazel

Shake

Nix
The order in which to execute tasks

- Topological
- Restart
- Suspend
“Monadic” dependencies

- When do I tell you my dependencies?
  - Applicative: Before doing anything, in advance
  - Monad: Before I use them

```
main.o :
  need main.c
  need $(includes_of main.c)
  gcc -c main.c
```

main.c : ...
• Only works for Applicative dependencies
• Build a graph, traverse graph
• Build a rule
• If it depends on a rule not yet built
  – Restart: Cancel this rule, schedule it last, build dep
  – Suspend: Pause this rule, build dep, resume

• Can you cancel or pause your rules?
• Pause requires more memory, but less work
Tricks for restarting

• Bazel
  – Use the applicative dependencies to part order
  – Doesn’t really allow user written monadic deps
• Excel
  – Keep a list of the order that worked last time
  – Consequence: Your sheet calcs faster over time!
Respecting dependencies

- Topological – Applicative only, easy
- Restart – May duplicate work
- Suspend – May be hard to orchestrate

Shake

- Shake’s raison d'être is monadic deps
- Uses continuations to efficiently suspend
  - First version used green threads
I rebuilt this rule last time, should I do so again?

• Dirty
• Verifying trace
• Constructive trace
• Deterministic constructive trace
Dirty bit

A rule is dirty if anything it depends on is dirty

• Excel records it directly
• Make encodes dirty bit with relative modtimes
  – modtime(in) > modtime(out) = dirty
  – Cute trick: outputting a new result clears the bit, and propagates dirty bits upstream
• You need to know your deps, ~Applicative only
A *trace* records the relevant bit of the state

- What did I depend on last time?
- What were the values of those things?

`main.o` depends on `main.c`, which had hash `0x12`

- If the trace matches, don’t rerun
Early cut-off

- What if I build but don’t change?
- Possible with Dirty? Possible with Verifying?
Aka “Cloud build” or “Distributed build systems”

- Record the output with the trace
- Shove all the traces on the server
- Now you can download already built stuff

Lots of engineering involved...
Imagine the output of a rule depends only on its inputs (deterministic)

• Given the inputs, I can predict the value of any output, download the final answer
• Less round-trips to the server
• Doesn’t support cut-off
Necessary actions

- Dirty – ~Applicative only
- Verifying trace – local only
- Constructive trace
- Deterministic constructive trace – no cut-off

Shake

- Uses optimised verifying trace (two versions)
Build Systems à la Carte

Engineering +

necessary actions

dirty

verifying trace

constructive trace

deterministic constructive trace

respecting dependencies

topological

Make

Ninja

Pants

Buck

restart

Excel

Bazel

suspend

Shake HEAD

Nix

Accepted to ICFP 2018 with Andrey Mokhov, Simon Peyton Jones
Engineering: Shake

Neil Mitchell

@ndm_haskell

https://shakebuild.com
Rewind the clock

PhD build system → Haskell EDSL

Standard Chartered → Replace Make with Shake

Monadic dependencies → Academic paper

Open source

Commercial users ← Engineering

GHC build system

Comparative academic paper → Distributed

Academic paper

Papers with Andrey Mokhov, Simon Peyton Jones, Simon Marlow
Simple Shake

out : in
cp in out

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

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

:: Action ()
Monad Action

:: Rule ()
Monad Rule
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
Generated files

MyGen.hs  MySource.xml

MySource.c

MySource.o

What does MySource.o depend on?
Generated approaches

• Hardcode it?
  – Very fragile.

• Hack an approximation of MyGen?
  – Slow, somewhat fragile, a lot of effort.

• Build in stages?
  – Non-compositional

• Run MyGen.hs and look at MySource.c
  – Easy, fast, precise. Use monadic dependencies.
Monadic is necessary

• If *any* rule needs monadic, you need it
  – Even if “rare” in your system
• Workarounds are not compositional
• Generated files cry out for monadic
  – Generated code is common in large projects

• Advice: Don’t use a non-monadic system
Parallelism
Robustness
Efficient

Build system
Monadic + suspend
Modern engineering
+
Haskell

Shake

Profiling
Lint
Analysis

Syntax
Types
Abstraction
Libraries
Monads
• In use for three nine years:
  – 1M+ build runs, 30K+ build objects,
    1M+ lines source, 1M+ lines generated

• Replaced 10,000 lines of Makefile with 1,000 lines of Shake scripts
  – Twice as fast to compile from scratch
  – Massively more robust

Disclaimer: I used to be employed by Standard Chartered Bank. These slides do not represent the views of Standard Chartered.
Ready for primetime!

- **Standard Chartered** have been using Shake since 2009, 1000’s of compiles per day.
- **factis research GmbH** use Shake to compile their Checkpad MED application.
- **Samplecount** have been using Shake since 2012, producing several open-source projects for working with Shake.
- **CovenantEyes** use Shake to build their Windows client.
- **Keystone Tower Systems** has a robotic welder with a Shake build system.
- **FP Complete** use Shake to build Docker images.

Don’t write a build system unless you have to!
Stealing from Haskell

- Syntax, reasonable DSLs
- Some use of the type system (not heavy)
- Abstraction, functions/modules/packages
- Profiling the Haskell functions
Extra features

- HTML profile reports
- Very multithreaded
- Progress reporting
- Reports of live files
- Lint reports
- ...

[Image: Status bar indicating 3m12s (82%) progress]
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 nothing changes

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

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

- Avoid lots of locking/parallelism
  - Take a lock, check storedValue a lot
- Binary serialisation is a bottleneck
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)
State changes

- Ready
- Error
- Running
- Loaded
- Missing
Inside “Running”

• Build all my dependencies from last time
  – If any changed, then dirty
• Look at my result from last time
  – If it has changed, then dirty
• If dirty, see if I’m in the constructive trace
  – If I am, copy the result into my trace
• If still dirty
  – Run the user supplied action
Efficient suspend

- Continuations are mind-blowing (still)

```
a
(a -> r) -> r
```

- $a = \text{I get given } \text{‘a’ now}$
- $(a -> r) -> r = \text{I get given } \text{‘a’ later}$
- Covariant/contravariant equivalence
- Efficiently pause a running computation
• Resumption is restarting suspended things

```
data Status
    = Running [Either Error Ready -> IO ()]
    | ...
```

• Resume everything when changing status
  – Resumtion is required to be “quick”
  – Therefore most resumption adds to the Pool...
Efficient parallelism

• A thread pool

\[
\text{addPool} :: \text{Pool} \rightarrow \text{PoolPriority} \rightarrow \text{IO ()} \rightarrow \text{IO ()}
\]

• Not to reduce thread overhead
  – Haskell threads are super cheap

• To limit parallelism, and cleanup/finish
Efficient journaling

• Shake needs to record the verifying traces
  – Recorded in .shake.database

• A linear record of traces
  – Append to the end
  – Size prefixed to detect corruption
  – Compact if $< \frac{1}{2}$ the values still useful
  – Flush every 5s
Conclusions

• **Build systems make three choices:**
  – Respecting dependencies
  – Necessary actions
  – Engineering choices

• **Shake occupies an interesting spot**
  – Plenty of engineering required to make it work