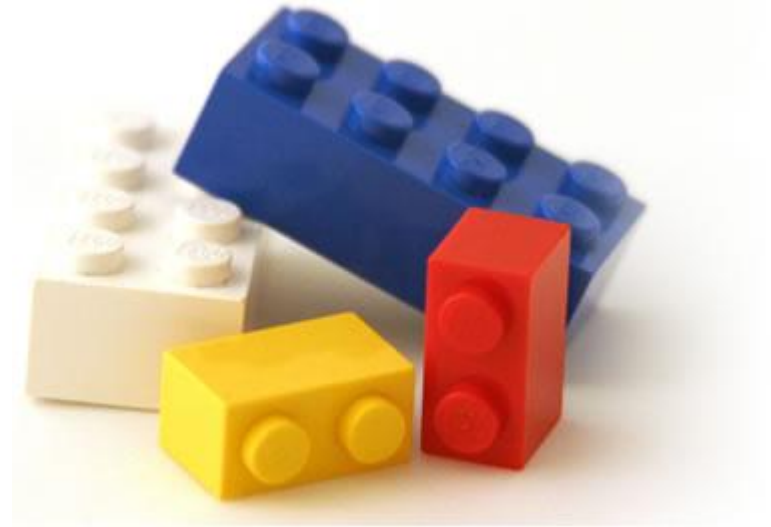


Shake Before Building

Replacing Make with Haskell

Neil Mitchell



community.haskell.org/~ndm/shake

General Purpose Build Systems

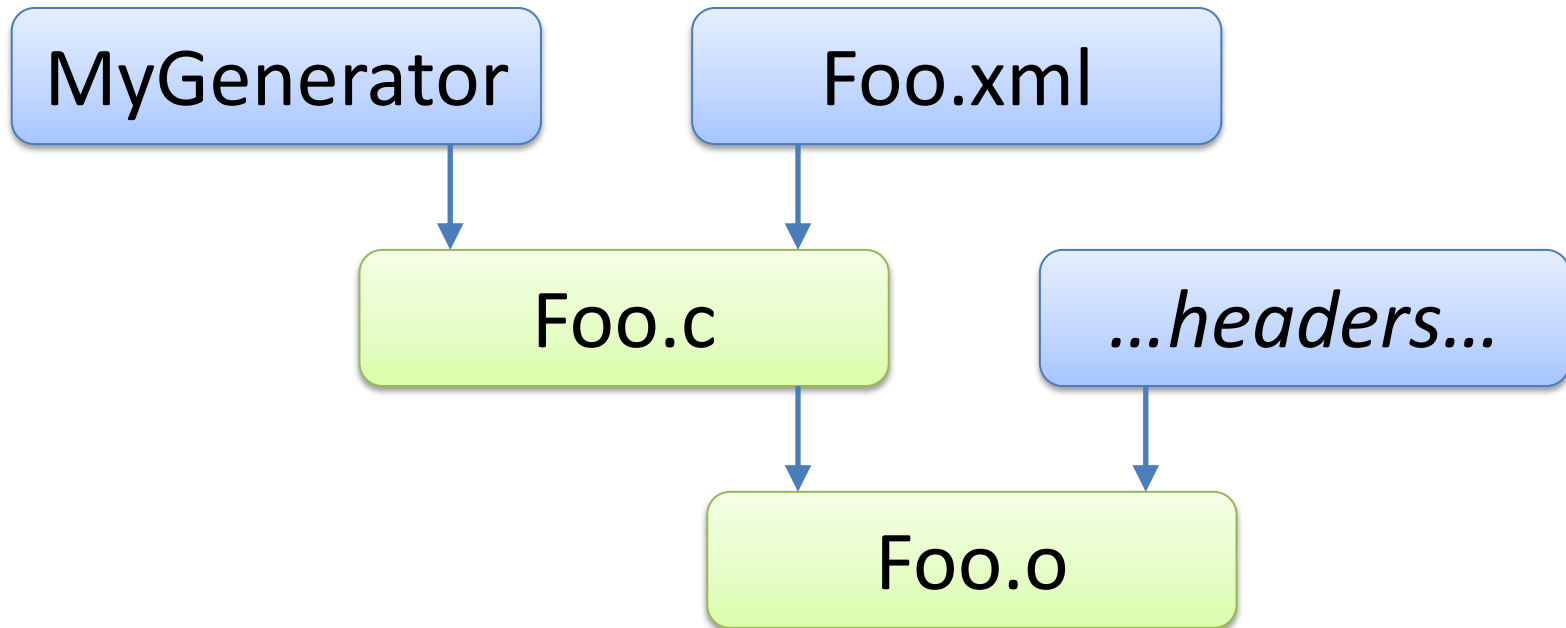


- Visual Studio
- ghc -make
- Cabal

- make
- Cmake
- Scons
- Waf
- Ant

Shake


Generated files



- 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

$$\mathbf{D}_{\text{shake}} > \mathbf{D}_{\text{make}}$$

**A build system with a
static dependency graph
is insufficient**



Parallelism
Robustness
Efficient

Profiling
Lint
Analysis

Build system
Better dependencies
Modern engineering
+ Haskell

Syntax
Types
Abstraction
Libraries
Monads

Shake

Shake at Standard Chartered



- In use for 3 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 am employed by Standard Chartered Bank. This paper has been created in a personal capacity and Standard Chartered Bank does not accept liability for its content. Views expressed in this paper do not necessarily represent the views of Standard Chartered Bank.

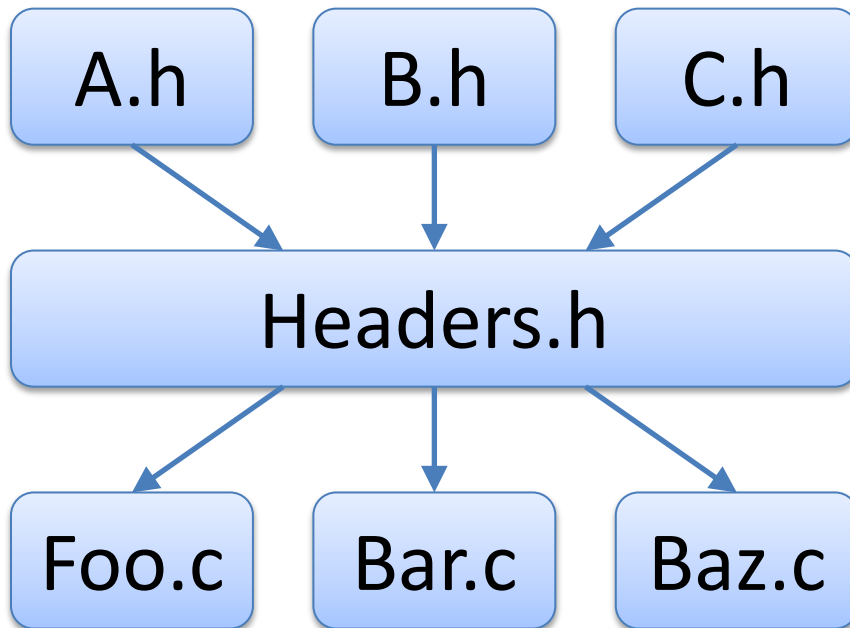
**Shake vs make:
10x shorter, 2x faster***



*for one real example

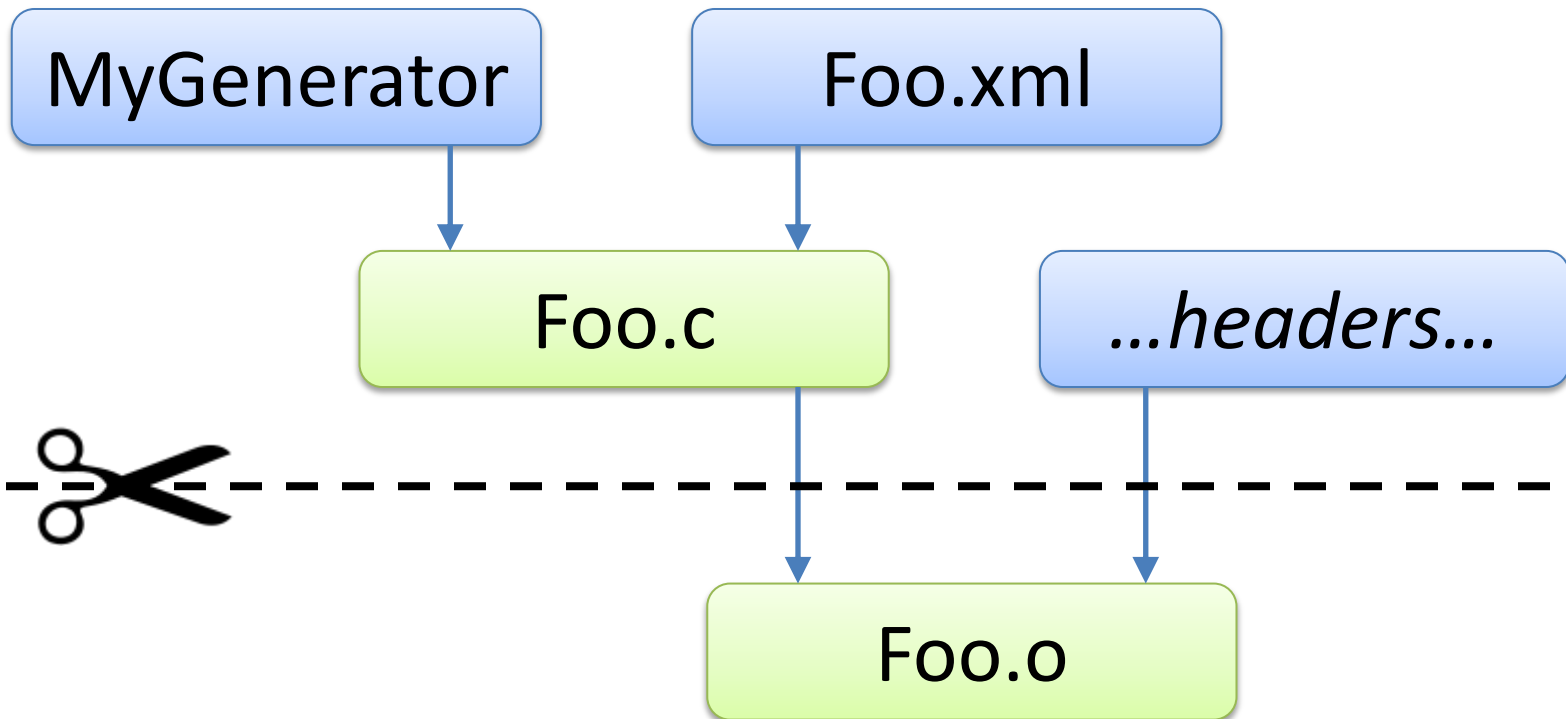
Faster 1 of 4: Less work

- gcc -MM finding headers has bad complexity
 - At large enough scale, it really matters



Scan each header once, instead of once per inclusion

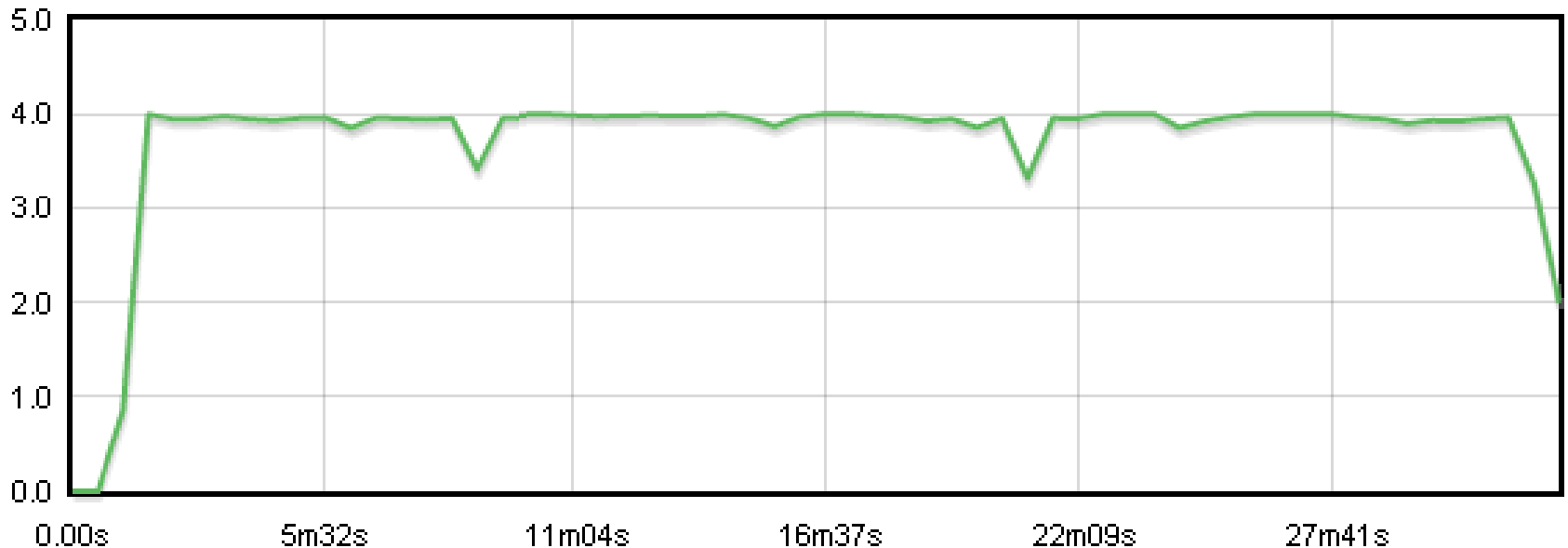
Faster 2 of 4: Less rebuilds



```
commit decea285a863ff147f53d3748aac8b13
Author: Neil Mitchell <neil@bigproject.com>
Comment: MyGenerator, whitespace only
```

Faster 3 of 4: More parallelism

- Big project \approx perfect parallelism
 - No unnecessary dependencies
 - Depend on only part of a file
 - No phases (overly coarse dependencies)



Faster 4 of 4: Better parallelism

- Random thread pool = 20% faster
 - Avoid all compiles then all links

Compiling



Linking



Shake outside a bank

- At least 10 Haskell build libraries
 - 3 are Shake inspired implementations
- 2 Shake addon libraries

There's a bit of scaffolding to get going, but the flexibility is really worth it to be able to handle auto-generated files easily.

More information

ICFP paper

Hackage (shake)

Shake Before Building
Replacing Make with Haskell

Ned Mitchell
n.mitchell@gmail.com

Abstract

Most complex software projects are compiled using a build tool to generate executables. Unfortunately, these build tools require a full dependency graph to be provided for the build system. This makes it difficult to change the build system, especially those involving files generated at build time. We show how to address this limitation, allowing additional dependencies to be specified while building. We show implementation in Haskell in the Haskell library Shake, and how to use Shake to write a complete build system to compile a collection of files of code.

Categories and Subject Descriptors D.3 [Software]: Programming Languages

General Terms Languages

Keywords build systems, compilation, Haskell

1. Introduction

A build tool, such as `make` (Richard 1976), takes a set of build files, `makefile` input files, and produces some set of files. Using `make`, a build system can be written as:

```
makefile file1.txt  
src/compile file1.txt
```

This file says that the file `result.txt` depends on the inputs `file1.txt` and `file2.txt` and produces a command to build `result.txt` from `file1.txt`. Whenever `file1.txt` or `file2.txt` change, the command will be run, and `result.txt` will be built.

One problem with using a build system from the list of files is that it is not clear how to generate the build system. Traditionally, the build system is generated from a `makefile` file, which is a text file containing the build system. Using the build tool to generate the build system is a more powerful approach.

`makefile` ← `A` → `file`
`cmd ["file1.txt"]`
`contents ← makefile, file1.txt → contents`
`system' "tar" ["-cf", "result.tar"] ← contents`

For a complete overview of the paper, please refer to Mitchell (2012).

2012.02.02

shake-0.2.5: Build system library, like Make, but more accurate dependencies. | Contents | Index | Frames

Development.Shake

This module is used for defining Shake build systems. As a simple example of a Shake build system, let us build the file `result.tar` from the files listed by `result.txt`:

```
import Development.Shake  
import Development.Shake.FilePath  
  
main = shake shakeOptions $ do  
  want ["result.tar"]  
  "*.tar" *> \out -> do  
    contents <- readFileLines $ replaceExtension out ".txt"  
    need contents  
    system' "tar" $ ["-cf", out] ++ contents
```

Contents

- Core of Shake
- Utility functions
- File rules
- Directory rules
- Additional rules
- Finite resources

We start by importing the modules defining both Shake and routines for manipulating `FilePath` values. We define `main` to call `shake` with the default `shakeOptions`. As the second argument to `shake`, we provide a set of rules. There are two common forms of rules, `want` to specify target files, and `*>` to define a rule which builds a `FilePattern`. We use `want` to require that after the build completes the file `result.tar` should be ready.

The `*.tar` rule describes how to build files with the extension `.tar`, including `result.tar`. We `readFileLines` on `result.txt`, after changing the `.tar` extension to `.txt`. We read each line into the variable `contents` -- being a list of the files that should go into `result.tar`. Next, we depend (`need`) all the files in `contents`. If any of these files change, the rule will be repeated. Finally we call the `tar` program. If either `result.txt` changes, or any of the files listed by `result.txt` change, then `result.tar` will be rebuilt.