Non-recursive Make Considered Harmful

Build Systems at Scale

Andrey Mokhov *
Newcastle University, UK
andrey.mokhov@ncl.ac.uk

Neil Mitchell †
Standard Chartered Bank, UK
ndmitchell@gmail.com

Simon Peyton Jones
Microsoft Research, UK
simonpj@microsoft.com

Simon Marlow
Facebook, UK
smarlow@fb.com

Abstract

Most build systems start small and simple, but over time grow into hairy monsters that few dare to touch. As we demonstrate in this paper, there are a few issues that cause build systems major scalability challenges, and many pervasively used build systems (e.g. Make) do not scale well.

This paper presents a solution to the challenges we identify. We use functional programming to design abstractions for build systems, and implement them on top of the Shake library, which allows us to describe build rules and dependencies. To substantiate our claims, we engineer a new build system for the Glasgow Haskell Compiler. The result is more scalable, faster, and spectacularly more maintainable than its Make-based predecessor.

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

Keywords build system, compilation, Haskell

1. Introduction

In 1998 Peter Miller published his famously influential paper *"Recursive Make Considered Harmful"* (Miller 1998). He made a compelling case that, when designing the build system for a large project, it is far better to ensure that Make can see the entire dependency graph rather than a series of fragments.

Miller was right about that. But he then went on to say "'*But, but, but*' I hear you cry. '*A single makefile is too big, it's unmaintainable, it's too hard to write... it's just not practical*"', after which he addresses each concern in turn[1]. Here, however, he is wrong. Using Make for large projects really *is* unmaintainable, and the rules really *are* too hard to write.

In this paper we substantiate this claim, and offer a solution, making the following contributions:

- Using the Glasgow Haskell Compiler (GHC) as a substantial exemplar, we give concrete evidence of the fundamental lack of scalability of Make and similar build systems (§2 and §4). GHC's build system is certainly large: it consists of over 10,000 lines of (often incomprehensible) code spread over 200 Makefiles. Motivated by its shortcomings, GHC developers have implemented no fewer than four major versions of the build system over the last 25 years; it improved each time, but the result is still manifestly inadequate.
- We describe Shake, an embedded domain specific language (or library) in Haskell that directly addresses these challenges in §3. Although Shake has been introduced before (Mitchell 2012), here we describe several key features that were mentioned only in passing if at all, notably: post-use and order-only dependencies; how to use polymorphic dependencies; resources; and content hashes.
- We show in some detail how Shake's built-in abstractions address many of the scalability challenges that have caused the GHC developers such pain over two decades (§4).
- A huge benefit of using an embedded DSL as a build system is that we can use the facilities of the host language (Haskell) to build abstractions on top of Shake, to fit our particular use case. This sort of claim is easier to make than to substantiate; so in §5 we present an overview of the new build system we have developed for GHC, and the new abstractions (not part of Shake) that we built to support it.
- To validate our claims, we have completely re-implemented GHC's build system, for the fifth and final time. The new version is only a little shorter than the old – Make is already extremely terse. Much more importantly, while the original was hard to comprehend and almost impossible to modify safely, the replacement is beautifully modular, statically typed, and extensible. Not only that, but the resulting system has much better behaviour and performance, as we discuss in §6.

None of this is fundamentally new; we review related work in §7. The distinctive feature of this paper is that it is grounded in the reality of a very large, long-lived software project. Peter Miller would be happy.

2. Challenges of Large-scale Build Systems

Many existing build systems work well for small projects, or projects that follow a common pattern. For example, building a single executable or library from single-language source files is well-supported in virtually all build systems. A lot of projects fit into this category, and so never run into the limits of existing build systems. Things start to get hairy in big projects, when complexities such as these show up:

- The project has a large number of components (libraries or executables), using multiple languages, with complex interdepen-

* Andrey Mokhov conducted this work during a 6-month research visit to Microsoft Research Cambridge that was funded by Newcastle University, EPSRC (grant reference EP/K503885/1), and Microsoft Research.

† Neil Mitchell is 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.

[1] As a historical aside, Miller points out that memory size is no longer a problem, because "the physical memory of modern computers exceeds 10MB". Indeed!

Haskell'16, September 22-23, 2016, Nara, Japan
ACM. 978-1-4503-4434-0/16/09...$15.00
http://dx.doi.org/10.1145/2976002.2976011

dencies. For example, executables depending on libraries, or tools generating inputs for other parts of the build system.

- Many components follow similar patterns, so there is a need for abstractions that allow common functionality to be shared between different parts of the build system.
- Parts of the build system are not static but are generated, perhaps by running tools that are built by the same build system.
- The build system has complex configuration, with aspects of its behaviour being controlled by a variety of different sources: automatic platform-specific configuration, command-line options, configuration files, and so on.

The GHC build system includes all of the above. To illustrate the consequences, here is one rule from GHC's current build system, which uses Make:

```
$1/$2/build/%.$$($3_osuf) : \
    $1/$4/%.hs $$(LAX_DEPS_FOLLOW) \
    $$$$($1_$2_HC_DEP) $$($1_$2_PKGDATA_DEP)
  $$(call cmd,$1_$2_HC) $$($1_$2_$3_ALL_HC_OPTS) \
    -c $$< -o $$@ \
    $$(if $$(findstring YES,$$($1_$2_DYNAMIC_TOO)), \
      -dyno $$(addsuffix .$$(dyn_osuf),$$(basename $$@)) )
  $$(call ohi-sanity-check,$1,$2,$3,$1/$2/build/$$*)
```

Yes, there are four dollar signs in a row! If it wasn't so tragic, impenetrable code like this would be hilarious. This kind of thing goes on for *thousands* of lines. The result is a nighmare to understand, maintain and modify.

Perhaps GHC's implementors just aren't very clever? Maybe so, but they did at least try hard. The current build system is the fourth major iteration, as they struggled to find good solutions to the problems that arose:

- The first incarnation of the build system used jmake, a system inspired by the X Consortium's imake. This tool was essentially ordinary Make combined with the C preprocessor to allow the use of macros in Makefiles. The macro layer partly solves the problem of needing to share functionality between different parts of the build system.
- The C preprocessor was somewhat painful to use with Make. The syntax was difficult to get right, and the need to constantly make Makefiles when working on the build system was tiresome. GNU Make came along which had built-in support for include files and other features, meaning the GHC build system could do away with the C preprocessor. The build system was rewritten to use GNU Make, and simulated macros with include files (GNU Make didn't have macros at the time).
- Next there were several large changes to the GHC build system that didn't amount to complete rewrites. First, the build system started to build multiple bootstrap *stages* in a single build tree; that is, build the compiler (stage 1) and then build the compiler again using the stage 1 compiler (stage 2). Previously this process had required two separate builds.

  Around this time the library ecosystem of Haskell was exploding, and the Cabal build system for Haskell libraries emerged. The GHC build system was integrated with Cabal to avoid duplicating Cabal metadata or build logic for the libraries that were part of the GHC.
- The build system in its current form had grown unwieldy, and had many idiosyncrasies. The root of many of the problems was that the build system was constructed as a set of Makefiles that recursively invoked each other. Recursive Make is considered harmful for very good reasons (Miller 1998); it is not possible to accurately track dependencies when the build system is constructed of separate components that invoke each other.

  In the next rewrite the build system was made non-recursive. However, in doing so, Make was stretched to its absolute limits, as the example above illustrates. Every part of the rule is there for a good reason, but the cumulative effect of solving all the problems that arise in a complex build system is impenetrable.

By carefully employing a set of idioms[2] (such as the prefix idiom $1_$2_ above, which is explained in §4.2 and §4.3), it was possible to construct a non-recursive build system for GHC in GNU Make. It works surprisingly well, and by virtue of being non-recursive it tracks dependencies accurately; but, it is almost impossible to understand and maintain. So while it is *possible* to develop large-scale build systems using Make, it is clear that GHC developers have gone far beyond Make's scalability limits.

3. Background about Shake

Many of the issues raised in §2 stem from the fact that Make was never designed to be a programming language. A promising approach is, therefore, abandon Make in favour of an embedded domain-specific language (DSL), so that we have access to the full static typing and abstraction facilities of the host language. We have done exactly that with GHC's build system, replacing Make with Shake, a DSL embedded in Haskell (Mitchell 2012). In this section we recap the key ideas behind Shake (§3.1-§3.3), and also describe some of the additional features provided by Shake, but not covered in the original paper (§3.4-§3.8). These features are of general use, and all predate our efforts to replace the GHC build system.

3.1 Introduction

As an example of a complete Shake build system, let us compile a C file into an object file:

```
 1  module Main(main) where
 2  import Development.Shake
 3  import System.FilePath
 4
 5  main :: IO ()
 6  main = shake shakeOptions $ do
 7      want ["foo.o"]
 8
 9      "*.o" %> \out → do
10          let src = out -<.> "c"
11          need [src]
12          cmd "gcc -c" src "-o" out
```

Following the code from top to bottom:

Line 1 declares a Haskell module. Shake is a Haskell library, so all Shake build systems are written in Haskell and can make full use of other Haskell libraries and Haskell abstractions (functions, modules, packages, let expressions etc).

Line 2 imports the Development.Shake module, which provides most of the functions and types in Shake. Some of the Shake API is given in Figure 1.

Line 3 imports the System.FilePath module, which in this example provides the -<.> function to replace a file's extension.

Lines 6 declares the main function, which calls shake. The shake function takes some options (parallelism settings, etc.), along with a set of Rules, and executes the necessary rules.

Line 7 calls want, to declare that after the build system has finished we would like the file foo.o to be available and up-to-date.

Line 9 defines a rule to build *.o files, namely those files which end with the extension .o. The %> operator produces a rule of

[2] https://ghc.haskell.org/trac/ghc/wiki/Building/Architecture

```
newtype Rules a = ...                                    Generic API
  deriving (Monoid, Functor, Applicative, Monad)
newtype Action a = ...
  deriving (Functor, Applicative, Monad, MonadIO)

data ShakeOptions = ShakeOptions {shakeThreads :: Int, ...}
shakeOptions :: ShakeOptions

shake :: ShakeOptions → Rules () → IO ()
action :: Action a → Rules ()

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

class (ShakeValue key, ShakeValue value) ⇒
      Rule key value where
    storedValue :: ShakeOptions →
      key → IO (Maybe value)

rule :: Rule key value ⇒ (key → Maybe (Action value)) → Rules ()
apply :: Rule key value ⇒ [key] → Action [value]

data Resource
newResource :: String → Int → Rules Resource
withResource :: Resource → Int → Action a → Action a
```

```
type FilePattern = String                          File-specific API
want :: [FilePath] → Rules ()
need :: [FilePath] → Action ()
needed :: [FilePath] → Action ()
orderOnly :: [FilePath] → Action ()
(%>) :: FilePattern → (FilePath → Action ()) → Rules ()
(&%>) :: [FilePattern] → ([FilePath] → Action ()) → Rules ()
(?>) :: (FilePath → Bool) → (FilePath → Action ()) → Rules ()
```

Figure 1. Shake API

type Rules which takes a pattern on the left, and an Action on the right. The variable out will be bound to the actual file being produced, namely foo.o in this example.

Line 10 computes the name of the source file, i.e. src = "foo.c".

Line 11 uses the Shake function need to ensure foo.c has been built before continuing, and to introduce a dependency that if foo.c changes then this rule will require rerunning.

Line 12 uses the variable-arity function cmd to execute the system command gcc with appropriate arguments to produce out from src. Since the Action type has an instance of MonadIO we can do any IO operation at this point.

On the first execution, this example will start running the ∗.o rule to produce foo.o. When execution gets to need [src] this rule will stop and the rule for foo.c will be run. Shake provides a default rule for files that do not match any other rules, which simply checks the file already exists. After completing this simple rule, Shake will resume running the ∗.o rule, executing gcc to build foo.o.

3.2 Parsimonious Rebuilding

Both Make and Shake try to eliminate unnecessary rebuilding, but each uses a different approach:

- In Make, a rule is rerun if *the output does not exist, or if the modification time of the output is earlier than any of the inputs*. In an example similar to that above, gcc would be rerun if either foo.o did not exist, or if the modification time of foo.o was earlier than that of foo.c.
- In Shake, a rule is rerun if *any of the inputs or outputs have changed since the last run of the rule*. In our example, Shake will rerun gcc if either foo.c or foo.o does not exist or if either file changes modification time from when the rule was last run (or more generally, if either file changes contents, see §3.8).

Shake achieves this result by maintaining a per-project database, storing information about inputs and outputs after a rule completes, and rerunning a rule if anything has changed (and thus the result could be expected to change). Using a separate database has a number of advantages: (i) we can track things that aren't files as in §3.6; (ii) we can depend on a more refined measure than modification time as in §3.8; (iii) we are robust to system time changes or extracting old files from backups; (iv) the execution of a rule does not need to update its output to avoid future rebuilds. The final point is often worked around in Make by using *stamp files* to provide a modification time but having no actual content – Shake simplifies this common use-case.

3.3 Pre-use Dependencies: need

The call need [src] on line 11 tells Shake that foo.c is required to build foo.o. But note that this dependency is only announced after Shake has started to run the rule building foo.c. In particular, we can perform arbitrary computation including I/O, running system commands and examining files required by a previous need, before declaring additional requirements with need. In contrast, in most build systems (Make included), all dependencies of a rule must be declared before the rule starts executing. The power to add additional dependencies while running a rule turns out to be important in practice (§4.6).

3.4 Post-use Dependencies: needed

Looking at our first example, the object gets recompiled if the C source file changes (via the call to need [src], line 11). But a C file may #include any number of header files, and changes to these headers will also affect the resulting object. One option is to write our own code to search for transitively included headers (Mitchell 2012, §6.4), another approach is to reuse the logic already present in gcc. The gcc compilation command (-c) can also take a flag -MD to generate a Makefile listing all headers that were used by the compilation. We can integrate this flag into our example by replacing the final line with:

```
let makefile = out -<.> "m"
unit $ cmd "gcc -MD -MF" makefile "-c" src "-o" out
deps ← liftIO $ readFile makefile
need $ makefileDependencies deps
```

We write the Makefile to foo.m, then use makefileDependencies to parse the Makefile and return all dependencies[3]. After obtaining the list of dependencies we need them, making them inputs to this rule, and ensuring that if a header changes the object will be rebuilt.

There is something a bit fishy here: *after* compiling foo.o we announce that its *pre-requisites* are bar.h and wibble.h. There is no issue if bar.h is a source file, and now Shake will know to re-build foo.o if the user edits bar.h. But if bar.h is itself generated by the build system, we have told Shake "too late", and Shake may now regenerate the file *after* the compilation has used it.

We solve this problem by using needed instead of need, which combines need with an assertion that the file does not change as a result of building, and thus the result is consistent. In the common case of the header files being source files, the associated rule will be the default file rule, which does not modify the file, and the assertion will not trigger.

[3] Using the parseMakefile helper function provided by Shake we can define makefileDependencies = concatMap snd . parseMakefile.

3.5 Order-only Dependencies: orderOnly

We have seen how needed can be safely used for source files, but what about generated files? Imagine we have a rule to build config.h from a configuration data file. With the existing formulation, needed ["config.h"] will raise an error if the configuration data has changed and config.h is not up-to-date. One solution is to need ["config.h"] *before* executing gcc. This solution is safe – the file config.h will always be built before it is used and will not change when needed (since it has already been built). However, this solution introduces a dependency on config.h, which causes any object file that does *not* include config.h to rebuild unnecessarily if config.h changes.

A better solution is to use order-only dependencies, with the expression orderOnly ["config.h"]. This expression ensures that config.h has been built and is up-to-date before continuing, but does not introduce a dependency on config.h: if config.h changes the rule will not be re-run. Afterwards, if the file turns out to have been required, needed can be used to express the dependency.

So, for a given rule; orderOnly ensures that a (potential) input file has been built before the rule is run; and needed ensures that the rule is re-run if an (actual) input file changes. Combining the two we obtain the equivalence:

```
need xs ≡ (orderOnly xs >> needed xs)
```

Although in practice both orderOnly and needed can be implemented on top of the need primitive.

3.6 Polymorphic Dependencies

While build-systems are typically focused around files, the core of Shake is fully polymorphic, operating on objects that are uniquely labelled by a key and have rules that produce a value. In the case of files, the key is the filename and the value is the modification time, or content hash (§3.8). Using the polymorphic API we can define new types of rules to track additional dependencies. This polymorphic API, and the file-specific API built on top of it, are presented in corresponding sections of Figure 1.

As an example of a new type of rule, let us imagine that the actual compiler name (gcc) is obtained from a configure script that produces a configuration file containing many key/value pairs. The ∗.o rule could need the configuration file, and parse it to extract the compiler name, but that would be wildly over-conservative: *any* change to the configuration file would cause *all* rules using the C compiler to rebuild, *even if the compiler name did not change*. Alternatively, we could have rules to split the single configuration file into one file per key, where each file is only updated if that key changes, but that might result in thousands of files. Instead, we can define a rule to map from configuration keys to values.

In Figure 2 we define two new types of rule. First, we define a rule from the configuration file to the key/value map it contains, using the ConfigMap type. Next we define a rule from a single key in the configuration file to the associated value, using the ConfigKey type. We define Rule instances to declare these new rule types to Shake, allowing us to use rule and apply from Figure 1.

- We *declare* rules using rule, which takes a function that for a given key, says how to compute its value. We use Just to indicate both rules apply to all keys of the appropriate type. In the first rule we read the file and parse it, producing an associative map.
- We *use* rules with apply, which takes a list of rule keys and computes the corresponding values. Here we use apply with singletons, but in general apply operates on list of keys which can be computed in parallel. When defining the rule for ConfigKey we use the value of ConfigMap. When defining ∗.o, we use the value of ConfigKey "cc".

```
newtype ConfigMap = ConfigMap FilePath deriving ...
instance Rule ConfigMap (Map String String)

newtype ConfigKey = ConfigKey String deriving ...
instance Rule ConfigKey String

main = shake shakeOptions $ do
    rule $ \(ConfigMap file) → Just $ do
        cfg ← readFile' file
        return $ parseConfig cfg

    rule $ \(ConfigKey key) → Just $ do
        [dict] ← apply [ConfigMap "config.file"]
        return $ dict Map.! key

    "*.o" %> \out → do
        [cc] ← apply [ConfigKey "cc"]
        ...
        cmd cc "-c" src "-o" out

parseConfig :: String → Map String String
parseConfig = ...
```

Figure 2. Using polymorphic dependencies in Shake

Crucially, Shake treats these computations incrementally: it parses each configuration file only once, regardless of the number of uses of apply; and it re-runs rules that look up ConfigKey "cc" only if the name of the C compiler actually changes. So if file config.file changes, Shake will re-run the ConfigMap rule, but if cc does not change, the ∗.o rule will not re-run.

The use of non-file rule types is common in large-scale build systems; for example, we use ten different types in our implementation of GHC's build system in §6. Polymorphic dependencies do not give any fundamental additional expressive power over file rules, but they do provide:

- An easy way to get distinct keys by using a freshly defined type.
- Additional structure for keys and values, instead of hierarchical filenames and binary content files. In particular they can contain full algebraic data types or associative maps.
- Greater granularity, by not forcing each rule to be backed by a separate file. GHC's build system uses 6,677 non-file rule values.

3.7 Concurrency Reduction

Like most build systems, Shake can run independent rules concurrently. However, running too many rules in parallel can cause a computer to become overloaded, spending much of its time switching between tasks instead of executing them. Most build systems take an argument to set the maximum number of rules that can be run in parallel (Shake has shakeThreads), and typically that number is based on the number of available CPUs. However, some rules may not be independent:

- Some APIs are global in nature. If you run two programs that access the Excel API simultaneously things start to fail.
- Many people have large numbers of CPUs, but only one slow rotating hard-drive. Running many disk-heavy linker processes simultaneously can overload the hard-drive.
- Some proprietary software has licenses which limit the number of concurrent processes, for example ModelSim.

Rules using such limited *resources* should not be run in parallel even though they do not have explicit dependencies between them. Build systems typically obey such constraints by setting an artificially low CPU limit, pausing rules competing for the resource, or

adding fake dependencies to serialise the rules. All these solutions fail to make best use of the available CPUs, and the fake dependencies are also fragile and tricky to implement.

Shake provides a Resource abstraction that solves this problem directly. A Resource value represents a finite resource that multiple build rules can use; such values are created with newResource and used by withResource. As an example, only one set of calls to the Excel API can occur at once, therefore Excel is a finite resource of quantity 1. We can write:

```
want ["a.xls","b.xls"]
excel ← newResource "Excel" 1
"*.xls" %> \out →
  withResource excel 1 $ cmd "excel" out ...
```

We create a new resource excel of quantity 1, and then we call withResource excel 1 to use it. Now we will never run two copies of Excel simultaneously, regardless of the shakeThreads setting. Moreover, the build system will never block waiting for the resource if there are other rules that could be run. We use this approach to deal with the GHC package database, which only permits one writer at a time, see §4.5.

3.8 Tracking File Contents

Build systems run actions on files, skipping the actions if the files have not changed. An important part of that process is determining if a file has changed. As we saw in §3.1, Make uses modification time to check outputs are newer than inputs, while Shake uses modification time as a proxy for file contents and rebuilds on any change. However, there are two common cases where a file can change modification time without changing its contents:

- When generating source code, e.g. C code from a Yacc/Bison parser generator, most trivial whitespace changes to the parser input will probably result in identical output.
- When working on two git branches, both of which are based on a common master branch, a typical pattern is to switch from one branch to another. If the first branch was recently synced with master, but the second has not been for a while, the typical workflow is to switch to the second branch and then merge with master. Assuming the differences between the two branches are small, the number of changed files is also likely to be small. However, if master changes regularly, any files that changed in master since the second branch was last synced will have a new modification time, despite having the same contents.

Shake can solve these problems by simply making the value of a file rule reflect the *contents* of that file, instead of the *modification time*. In the remainder of this section we discuss how to efficiently implement something approximating that simple scheme. Since Shake rules are polymorphic it is easy to provide multiple types of coexisting file rules, although for simplicity we instead provide the option of which file value type to use as a field of ShakeOptions.

The obvious problem with simply storing the entire contents of all files is that it will result in a huge Shake database. Instead, we store the result of applying a hash function to the file contents, where the hash changing indicates the file contents have changed. There is a remote risk that the file will change without its hash changing, but unless the build system users are actively hostile, that is unlikely. The disadvantage of content hashes over modification times is that hashes are expensive to compute, requiring a full scan of the file. In particular, after a rule finishes Shake must scan the file it just built, and on startup Shake must scan all files. Scanning all files can cause rebuild checks to take minutes instead of less than a second.

As an optimisation, Shake stores the modification time, file size and hash of the contents. After a rule completes all the information is computed and stored. When checking if a file has changed, first the modification time is checked, and if that matches, the contents are assumed to have not changed. If the modification time has changed, and the file size has also changed, then the file has definitely changed. Only in the case where the modification time has changed but the size has not do we compute the actual hash. If that hash is equal to the previously recorded hash we store a new modification time, so that future checks will avoid computing the hash. These optimisations give most of the benefits of storing the file contents, but with significantly reduced costs.

4. Quick Wins: From Make to Shake

The GHC build system stretches Make beyond its limits. In this section we illustrate, from our experience with GHC's build system, some of these challenges, and how they can be solved. These problems can be divided into two groups: those caused by the Make language (solved using Haskell); and those caused by a lack of expressive power when describing dependencies (solved using Shake). Building on these "quick wins", we show how to structure a large build system in §5.

4.1 Variables

Make's program state involves a global namespace of mutable string variables that are spliced into the program. This model naturally causes challenges:

- Since variables live in a single global namespace, there is limited encapsulation and implementation hiding.
- Since variables are strings, arrays and associative maps are typically encoded using computed variable names, which is error-prone.
- Since variable references are spliced into the Makefile contents and interpreted, certain special characters can cause problems – notably space (which splits lexemes) and colon (which declares rules, but is also found in Windows drive letters).

By using a build system embedded in Haskell, or indeed any other modern programming language, most of these issues are eliminated. An alternative solution is to generate the Makefile, either using a tool such as Automake or a full programming language. However, a generator cannot interact with the build system after generation, resulting in problems with dynamic dependencies (§4.6).

4.2 Macros

Consider the following Make rule:

```
%.o : %.hs
  ghc $HC_OPTS $<
```

It tells Make that object files *.o are produced from Haskell source files *.hs by compiling them using ghc invoked with HC_OPTS arguments. The notation is terse and works well for this example. Unfortunately, this simple formulation does not scale:

- What if we want the rule to match foo.o and bar.o, but not baz.o? It is impossible to do any non-trivial computation – we are forced to rely on patterns whose expressive power is limited.
- What if HC_OPTS should depend on the file being compiled?

The standard Make approach to solve these problems is to use *macros*. As an example, here is a highly simplified version of the macro from §2:

```
$1/$2/build/%.o : $1/$4/%.hs
  ghc $$($1_$2_$3_HC_OPTS) -c $$< -o $$@
```

As before, the rule is responsible for compiling a Haskell source file into an object file. The arguments to the macro are available as $1 to $4. We have solved the two problems above, but inelegantly:

- To make up for weak pattern matching, we use macros to generate a separate rule for each $1/$2/$4 combination.
- The dependence of the command line arguments on the file is solved using a computed variable name, which is dereferenced using $$(...) to ensure the value is expanded when the macro is *used*, not when it is *defined*. However, certain variables are present in positions where the value is unavailable when first called, requiring $$$$(..) to further delay expansion. Alas, it is far from obvious when extra delaying is required.

Using Shake, the simplest variant looks slightly more complex, because it must be expressed in Haskell syntax:

```
"*.o" %> \out → do
    let hs = out -<.> "hs"
    need [hs]
    cmd "ghc" hc_opts hs
```

However, as the complexity grows, the build system scales properly. Making hc_opts depend on the Haskell file requires the small and obvious change:

```
cmd "ghc" (hc_opts hs) hs
```

Namely, we turn hc_opts from a constant to a function. To use richer pattern matching we can drop down to a lower-level Shake operation. In Shake %> is itself defined in terms of ?>:

```
pattern %> act = (pattern ?==) ?> act
```

Wildcard pattern matching is just a special case, and we can use an arbitrary predicate to exert more precise control over what matches.

An alert reader may have noticed that $1_$2_$3_HC_OPTS refers to $3, which is not related to the file being built. Indeed, the macro argument $3 specifies *how* the file must be built. The next subsection discusses this pattern and associated challenges.

4.3 Computing Command Lines

In any large-scale build system, there are rules that say how to build targets, and then there is a long list of special cases, listing targets that need to be treated specially. For example, the GHC build system generates different rules for each combination of package, compiler version, and build way (in §5.1 we refer to such combinations as *build contexts*). The build rules differ in several aspects, and in particular they use different command line arguments. The following snippet shows how command line arguments $1_$2_$3_HC_OPTS are computed in the GHC build system:

```
WAY_p_HC_OPTS = -static -prof
...
base_stage1_HC_OPTS = -this-unit-id base
...
$1_$2_$3_HC_OPTS = $$(WAY_$3_HC_OPTS) \
                   $$($1_$2_HC_OPTS) \
                   ... plus 30 more configuration patterns
```

This says that when building targets with GHC in the *profiling way* p (i.e. with the profiling information enabled), we need to add -static -prof to the command line. Furthermore, when we compile targets from the base library using the Stage1 GHC, we need to add -this-unit-id base to the command line.

WAY_p_HC_OPTS = -static -prof is concise, but also inflexible. For example, if we do not want to pass the -prof flag when building a particular file, we have to either add a new component to the variable name, e.g. WAY_p_file_HC_OPTS, or conditionally filter out the -prof flag from HC_OPTS after it is constructed. Both approaches are not scalable; consequently, much of the complexity of the GHC build system is caused by computing command lines. This complexity is unnecessary and can be tackled using high-level abstractions readily available in Haskell. We elaborate on this solution in §5.3, where we design a DSL for succinct and type-safe computation of build command lines.

4.4 Rules with Multiple Outputs

Given a Haskell source file Foo.hs, GHC compilation produces both an interface file Foo.hi and an object file Foo.o. The typical way of expressing multiple outputs in Make is to use two rules, one depending on the other. For example:

```
%.o : %.hs
    ghc $HC_OPTS $<

%.hi : %.o ;
```

The second rule is a no-op: it tells Make that an interface file can be produced simply by depending on the corresponding object file. Alas, this approach is fragile: if we delete the interface file, but leave the object file intact, Make will not rerun the *.o rule, because the object file is up-to-date. Consequently, the build system will fail to restore the deleted interface file.

In Shake we can express this rule directly using the operator &%>, which defines a build rule with multiple outputs:

```
["*.o", "*.hi"] &%> \[o, hi] → do
    let hs = o -<.> "hs"
    need [hs]
    cmd "ghc" hc_opts hs
```

4.5 Reducing Concurrency

As discussed in §3.7, it is sometimes necessary to reduce concurrency in a build system. As an example, GHC packages need to be registered by invoking the ghc-pkg utility. This utility mutates the global state (package database) and hence at most one package can be registered at a time, or the database is corrupted.

In Make, the solution is to introduce fake *concurrency reduction* dependencies. In GHC's build system there are 25 packages that require registration, and in the old build system they all depended on each other in a chain, to ensure no simultaneous registrations occur. This solution works, but is fragile (easy to get wrong) and inefficient (reduces available parallelism).

In Shake, the solution is to use the *resources* feature (§3.7):

```
packageDb ← newResource "package-db" 1
...
action $ withResource packageDb 1 $ cmd "ghc-pkg" ...
```

This snippet declares a global resource named packageDb with quantity 1, then later calls withResource asking for a single quantity of the resource to be held while running the ghc-pkg utility. Provided all ghc-pkg calls are suitably wrapped, we will never run two instances simultaneously. Furthermore, thanks to the availability of functions, we can abstract a function that both executes ghc-pkg and takes the resource.

4.6 Dynamic Dependencies

Make, in common with many other build systems, works by constructing a dependency graph and then executing it. This approach makes it possible to analyse the graph ahead of time, but it is limiting in some fundamental ways. Specifically, it is common that parts of the dependency graph can be known only after executing some other parts of the dependency graph. Here are some examples of this pattern from the GHC build system:

- GHC contains Haskell packages which have metadata specified in .cabal files. We need to extract the metadata and generate package-data.mk files to be included into the build system; this is done by a Haskell program called ghc-cabal, which is built by the build system. Hence we do not know how to build the packages until we have built and run the ghc-cabal tool.

  One "solution" to this problem is to generate the .mk files and check them into the repository whenever they change. But checking in generated files is ugly and introduces more failure cases when the generated files get out of sync.

- The dependencies of a Haskell source file can be extracted by running the compiler, much like using gcc -M on a C source file. But the Haskell compiler is one of the things that we are building, so we cannot know the dependencies of many of the Haskell source files until we have built the stage 1 compiler.

- Source files processed by the C preprocessor use #include directives to include other files. We can only know what the dependencies are by running the C preprocessor to gather the set of filenames that were included.

There are several other cases of such *dynamic dependencies* in the build system. Indeed, they arise naturally even when building a simple C program, because the #include files for a C source file are not known until the source file is compiled or analysed by the C compiler, using something like gcc -M. Build systems that have a static dependency graph typically have special-case support for changing dependencies; for example, Make will re-read Makefiles that change during building. In the case of Make, this works for simple cases, but fails when there are dependencies between the generated Makefiles, which arises in more complex cases.

The GHC build system works around this limitation of Make by dividing the build system into *phases*, where each phase generates the parts of the build system that are required by subsequent phases. The GHC developers managed to reduce the number of phases to three, by carefully making use of Make's automatic restarting feature where possible. However, explicit phases are a terrible solution for several reasons:

- Phases reduce concurrency: we cannot start the next phase until the previous phase is complete.
- Knowing what targets to put in each phase requires a deep understanding of generated parts of the build system and their dependencies, and this area of our build system is notoriously difficult to work on. Diagnosing problems is particularly hard.
- We have to run all the phases even when doing an incremental build, because explicit phases preclude dependency tracking across the phase boundary. The result is so slow that we were forced to allow the user to short-circuit the earlier phases, by promising that nothing has changed that would require rebuilding the early phases (make fast). This solution is less than ideal.

Shake supports dynamic dependencies natively, so these problems just go away. The need function can introduce new dependencies *after observing the results of previous dependencies*[4] (see §3.3). For example, when building a Haskell package, we first need the ghc-cabal tool, then we run it on the package's .cabal file to extract the package metadata, and then we consult the result to determine what needs to be done to build the package.

[4] It has been suggested that the dependency mechanism in Shake should rightly be called *Monadic dependencies* to contrast with the *Applicative dependencies* in Make. We agree. In an applicative computation the structure cannot depend on the values flowing through the container. In a monadic computation the structure can depend on the values.

The existence of need means that Shake cannot construct the dependency graph ahead of time, because it doesn't know the full dependency graph until the build has completed. But this is not a limitation in practice, whereas the lack of dynamic dependencies really *is* a limitation requiring painful workarounds.

4.7 Avoiding External Tools

Build systems typically spend most of their time calling out to external tools, e.g. compilers. But sometimes it is useful to replace external tools with direct functions. As an example, the Make system uses xargs to split command line arguments that exceed a certain size; but xargs does not work consistently across OS versions, requiring lots of conditional logic. Fortunately, with Haskell at our disposal, we can write:

```
-- | @chunksOfSize size strings@ splits a given list of strings
-- into chunks not exceeding @size@ characters. If that is
-- impossible, it uses singleton chunks.
chunksOfSize :: Int → [String] → [[String]]
chunksOfSize n = repeatedly $ \xs →
    let ys = takeWhile (≤ n) $ scanl1 (+) $ map length xs
    in splitAt (max 1 $ length ys) xs
```

Writing a small function is easy in Haskell, as is testing it (we test this function using QuickCheck). Writing it in bash would be infeasible. Reducing the specific behaviours required from external tools leads to significantly fewer cross-platform concerns.

4.8 Summary

The unnecessary complexities described in this section have a big impact on the overall complexity of the build system. The main lessons we have learnt are:

- Abstraction is a powerful and necessary tool. Lack of good abstraction mechanisms is a significant cause of the complexity of previous attempts in Make. Functional programming provides excellent abstractions – marrying build systems and functional programming works well.
- Expressive dependencies are important; we use multiple outputs §4.4, resources §4.5 and dynamic dependencies §4.6. While some of these features are only used in a few places (e.g. resources), their absence requires pervasive workarounds, and a significant increase in the overall complexity.

5. Abstractions

In the previous section we covered how Shake helps us sidestep the unnecessary complexities inherent in a large-scale build system. In this section we focus on the complexities that remain. In particular, we develop abstractions for describing build configurations on top of Shake. Common configuration settings include turning on/off documentation, choosing different sets of optimisation flags, etc. As shown in §4.3, many GHC users work with GHC in different modes and in different environments, leading to a combinatorial explosion of possible configurations.

We first focus on the specifics of the GHC build system (§5.1). The remaining abstractions (§5.2) are independent of GHC and will, we believe, be useful to others. We then describe the configuration language we developed (§5.3), which is tracked (if a configuration setting changes, the affected build rules are rerun), can have provenance (§8), and permits easy configuration. As an example:

```
builderGhc ? way profiling ? arg "-prof"
```

This expression adds the -prof argument to the command line when building a file with GHC in the profiling way.

```
data PackageType = Library | Program                    Build types

data Package = Package
    { pkgName :: PackageName
    , pkgPath  :: FilePath
    , pkgType  :: PackageType }

newtype Way = ... deriving Eq

data Stage = Stage0 | Stage1 | Stage2 | Stage3 deriving Enum

data Context = Context
    { stage   :: Stage
    , package :: Package
    , way     :: Way }

data Builder = Alex
             | Ar
             | GenPrimopCode
             | Ghc Stage
             | Haddock
             ... plus 22 more builders

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

```
type Expr a = ReaderT Target Action a                   Expressions

newtype Diff a = Diff { fromDiff :: a → a }

type Args = Expr (Diff [String])

append :: [String] → Args
append as = return $ Diff (<> as)

remove :: [String] → Args
remove as = return . Diff $ filter (`notElem` as)

arg :: String → Args
arg = append . return

interpret :: Target → Args → Action [String]
interpret target args = do
    diff ← runReaderT args target
    return $ fromDiff diff mempty
```

```
way :: Way → Expr Bool                                  Predicates
way w = do
    target ← ask
    return $ Context.way (context target) == w

stage          :: Stage       → Expr Bool
package        :: Package     → Expr Bool
builder        :: Builder     → Expr Bool
builderGhc     ::               Expr Bool
input, output  :: FilePattern → Expr Bool

(?) :: Monoid a ⇒ Expr Bool → Expr a → Expr a
predicate ? expr = do
    bool ← predicate
    if bool then expr else return mempty
```

Figure 3. GHC build system abstractions

5.1 Context

We start by describing GHC-specific build types, which form our *build context*. By abstracting over the context in the subsequent sections we derive a set of generally useful build abstractions that are applicable to many build systems.

GHC source code is split into logical units, or *packages*. We model packages with the Package type, see Figure 3 (Build types). A package is identified by a unique PackageName and a FilePath pointing to its location in the source tree. A GHC package can be a library (e.g. array) or a program (e.g. haddock), which is captured by PackageType. There are 32 libraries and 18 programs (the latter includes GHC itself and various utilities).

A package can be built multiple *ways*, for example, to produce a library with or without profiling information. The way is captured by an opaque type Way inhabited by values such as vanilla (the simplest possible way), profiling (with profiling information), debug (with debug information), and many others (there are 18 ways in total). Some ways can be combined, e.g. debugProfiling; however, not all combinations make sense. By making Way opaque we make it easier to add new ways or change their internal representation, something that would be impossible to achieve in Make, where no information hiding is possible.

In addition to different build ways, each package can be built by several versions of GHC, which leads to the notion of *stages*. In Stage0 we use the *bootstrap* GHC, i.e. the one that is installed on the system. During this stage we build Stage1 GHC, an intermediate compiler that still lacks many features. It is used during the following Stage1 for building a fully-featured Stage2 GHC, the primary goal of the build system. We sometimes also build Stage3 GHC as a self-test: the object code of Stage2 and Stage3 compilers should be the same.

Stage, Package and Way form a GHC-specific *build context* represented by the type Context, see Figure 3. A typical GHC build rule, such as compilePackage, depends on the context as follows: it uses an appropriate compiler version (e.g. the bootstrap compiler in Stage0), produces object files with different extensions (e.g. vanilla $*$.o or profiled $*$.p_o object files), puts build artifacts into an appropriate directory (e.g. stage1/libraries/base), etc.

5.2 Builder and Target

A typical build system invokes several build tools, or *builders*, such as compilers, linkers, etc., some of which may be built by the build system itself. The builders are captured by the Builder type. It is useful to distinguish *internal* and *external* builders, i.e. those that are built by the build system and those which are installed on the system, respectively. The function builderProvenance returns the stage during which an internal builder is built, the way it is built, and the package containing the sources (all captured by a Context); Nothing is known about the provenance of external builders.

```
builderProvenance :: Builder → Maybe Context
builderProvenance x = case x of
  Ghc Stage0 → Nothing
  Ghc stage  → Just $ Context (pred stage) ghc vanilla
  Haddock    → Just $ Context Stage2 haddock vanilla
  ...
  _          → Nothing
```

In particular, we can see that Ghc Stage0 is an external builder, Ghc Stage1 is internal, built from package ghc during Stage0, Haddock is built in Stage2, etc. There are 27 builders, 16 of which are internal. Furthermore, some builders are *optional*, e.g. HsColour, which (if installed) is used to colourise Haskell code when building documentation.

Each invocation of a builder is fully described by a Target, which comprises a build Context, a Builder, a list of input files and

a list of output files. 3748 targets are built when building Stage2 GHC with documentation (with vanilla and profiled libraries). Consider the following Target as an example:

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

By examining preludeTarget it is possible to compute the full command line for building build/stage1/libraries/base/Prelude.p_o:

- The builder is Ghc Stage1. We lookup the right command inplace/bin/ghc-stage1 with help of builderProvenance, and use it in the following command line template:

  -O2 -c <input> -o <output>

- The way is profiling, so we know that we need to add -prof.
- We know how to substitute <input> and <output> in the above template.

The resulting full command line is:

```
inplace/bin/ghc-stage1 -O2 -prof -c libraries/base/Prelude.hs
    -o build/stage1/libraries/base/Prelude.p_o
```

A build system typically contains many such computations (at least one for each builder) and it is important to provide a terse and readable notation to describe the transformation from a Target to a command line. After experimenting with several abstractions, we converged on *expressions*, as defined in the next subsection.

5.3 Expressions

An expression Expr a is a computation that produces a value of type Action a and can read the current build Target, as shown in Figure 3 (Expressions). For example, the following expression computes command line arguments for invoking GHC:

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

5.3.1 Predicates

The use of conditional Expr values is pervasive. In the ghcArgs expression above -prof is only applied when profiling. But in fact, the entire expression is only applicable when using the Ghc builder. To make conditionals more concise we use *predicates* of type Expr Bool, see Figure 3 (Predicates). In particular, we use way :: Way → Expr Bool to check which way is currently being built. For example, predicate way profiling returns True when the current target is built using the profiling way.

Operator (?) :: Monoid a ⇒ Expr Bool → Expr a → Expr a is a convenient shortcut for applying a predicate to an expression that computes a monoidal value, such as [String]. For example, the following expression returns ["-prof"] when the current target is built the profiling way, and an empty list of arguments otherwise:

```
prof :: Expr [String]
prof = way profiling ? return ["-prof"]
```

Expressions computing monoids themselves form a monoid:

```
instance Monoid a ⇒ Monoid (Expr a) where
    mempty  = return mempty
    mappend = liftM2 mappend
```

Predicates and monoidal expressions are a powerful combination with many useful laws that allow us to reason about them:

1. Absorption: p ? mempty ≡ mempty
2. Distributivity: p ? (e <> f) ≡ p ? e <> p ? f
3. Conjunction: p ? q ? e ≡ (p ∧ q) ? e ≡ q ? p ? e
4. Disjunction: if p ∧ q ≡ False then
   p ? e <> q ? e ≡ (p ∨ q) ? e ≡ q ? e <> p ? e
5. Complement: p ? e <> ¬p ? e ≡ e

These laws appear in other applications and come with efficient proof methods, as studied by Mokhov and Khomenko (2014).

5.3.2 Extensibility

All expressions need to be modifiable by users of the build system. We therefore need to provide a way not only to add new arguments, but also to modify and remove them (e.g. remove -O2 under some condition). A simple solution is to switch to difference list expressions, represented by the type Expr (Diff a), which is used to construct values of type Diff a with the following monoid instance:

```
instance Monoid (Diff a) where
    mempty                  = Diff id
    mappend (Diff x) (Diff y) = Diff $ y . x
```

The reverse order of function composition y . x ensures that when two Expr (Diff a) computations are combined c1 <> c2, then c1 is applied first and c2 is applied second.

The following functions can be used to append and remove items to/from a difference list:

```
append :: Monoid a ⇒ a → Expr (Diff a)
append x = return $ Diff (<> x)

remove :: Eq a ⇒ [a] → Expr (Diff [a])
remove xs = return . Diff $ filter (`notElem` xs)
```

We are now ready to introduce Args, a type of expression for constructing command line arguments in the build system. In addition to the above generic functions (whose specialised versions are shown in Figure 3), it is equipped with the function arg :: String → Args for injecting simple String arguments into an expression, e.g. arg "-prof" :: Args.

With these abstractions in place, we can construct command line arguments for GHC as follows:

```
ghcArgs :: Args
ghcArgs = builderGhc ? mconcat
    [ arg "-O2"
    , way profiling ? arg "-prof"
    , arg "-c", arg =<< getInput
    , arg "-o", arg =<< getOutput ]
```

Here getInput :: Expr FilePath and getOutput :: Expr FilePath are expressions that check that Target inputs and outputs contain exactly one element and return it.

The resulting ghcArgs expression is terse and readable. All distracting plumbing details have been abstracted away so that the designers and users of the build system can focus on what matters.

We compose all command line arguments into a single args expression, applying custom user modifications userArgs to the end, allowing the user to override any default setting:

```
args :: Args
args = mconcat [ ghcArgs, ..., userArgs ]
```

The resulting expression is used in the build function that is responsible for building a given Target:

```
build :: Target → Action ()
build target@Target {..} = do
    path ← builderPath builder
    need [path]
    checkArgsHash target
    argList ← interpret target args
    cmd [path] argList
```

The build function proceeds as follows:

- First, builderPath :: Builder → Action FilePath determines the path to the builder depending on its provenance and the contents of the system.config file.
- We need the builder to make sure it is up-to-date. Some builders are built by the build system, e.g. genprimopcode, ghc-cabal, Stage1 GHC, so it is important to rebuild them if needed.
- The function checkArgsHash :: Target → Action () checks whether the command line computed from args expression has changed since the previous build. If it has, the target is rebuilt even if it is otherwise up-to-date. This step tracks changes both in the environment and in the build system itself. We track command lines using polymorphic dependencies, see §3.6.
- We interpret the args expression w.r.t. to the target, and obtain the list argList :: [String] of arguments to be passed to the builder. See Figure 3 for the implementation of interpret.
- Finally, we invoke the builder with appropriate arguments using Shake's cmd function.

5.4 A Simple Build System Example

Figure 4 shows a simple build system that uses the abstractions introduced in this section. The build system comprises two packages (Array and Base) that can be built in two ways (Vanilla and Profiling), using two versions of GHC (Ghc Head and Ghc Release) and an archiver tool Ar. Note, the Context in Figure 4 is different from the GHC Context presented in Figure 3, but this does not prevent us from using the same abstractions.

Build rules are generated by the buildPackage function, which given a build Context, describes how to compile a Haskell source file and build a library by archiving the obtained object files. buildPackage relies on several helper functions, which define the path to build artifacts, set way-specific extensions for object and archive files, lookup dependencies of a source file, and compute source files of a library. Command line arguments are specified as a single expression args, which makes use of package-, way-, builder-, and file-specific arguments; alternatively, parts of args can be defined separately and combined in a more modular way.

The main function is straightforward: for each possible Context we request the corresponding library to be built using want, as well as generate necessary rules by calling buildPackage. If we run the build system it will build 8 libraries and all associated object files.

6. Shaking up GHC

In this section we report on our experience of applying the techniques presented so far to building a large-scale software project: the Glasgow Haskell Compiler. We implemented[5] a new build system for GHC from scratch using Shake and our build abstractions from §5. The new build system does not yet implement the full functionality of the old build system, but we are currently addressing remaining limitations; nothing presents any new challenges or requires changes to the build infrastructure.

[5] https://github.com/snowleopard/hadrian

Build types

```
data Version  = Head   | Release
data Package  = Array  | Base
data Way      = Vanilla | Profiling

data Context  = Context Version Package Way

data Builder  = Ghc Version | Ar
```

Command line arguments

```
args :: Args
args = mconcat
   [ builderGhc ? mconcat
      [ arg "-O2"
      , way Profiling ? arg "-prof"
      , arg "-c", arg =<< getInput
      , arg "-o", arg =<< getOutput ]

   , builder Ar ? mconcat
      [ arg "q"
      , arg =<< getOutput
      , append =<< getInputs ]

   , builderGhc ? output "//GHC/IO.*" ? arg "-funbox-strict-fields"

   , builderGhc ? package Array ? arg "-Wall" ]
```

Build rules

```
buildPackage :: Context → Rules ()
buildPackage context@Context {..} = do
   path context </> "*" ++ osuf way %> \obj → do
      let src = obj -<.> "hs"
      deps ← lookupDependencies context obj
      need $ src : deps
      build $ Target context (Ghc version) [src] [obj]

   path context </> "*" ++ asuf way %> \a → do
      srcs ← lookupSources context
      let objs = [ src -<.> osuf way | src ← srcs ]
      need objs
      build $ Target context Ar objs [a]

path :: Context → FilePath
path Context {..} = show version </> show package

osuf :: Way → String
osuf Vanilla  = ".o"
osuf Profiling = ".p_o"

asuf :: Way → String
asuf Vanilla  = ".a"
asuf Profiling = "_p.a"

lookupDependencies :: Context → FilePath → Action [FilePath]
lookupDependencies context src = do ...

lookupSources :: Context → Action [FilePath]
lookupSources context = do ...
```

Main

```
main :: IO ()
main = shake shakeOptions $ do
   for_ [Head, Release] $ \version →
      for_ [Array, Base] $ \package →
         for_ [Vanilla, Profiling] $ \way → do
            let context = Context version package way
            want [path context </> "HSlib" ++ asuf way]
            buildPackage context
```

Figure 4. Example of a build system

Use case	Old build system based on Make		New build system based on Shake	
U1: Fully-featured GHC build	Everything is built	☑	Not all features supported	☐
U2: Clean build	Everything is built	☑	Everything is built	☑
U3: Zero build	Nothing is rebuilt	☑	Nothing is rebuilt	☑
U4: Touch: libraries/base/Prelude.hs	Prelude.o, base library, and all dependent binaries are rebuilt	☐	Nothing is rebuilt	☑
U5: Add comment: libraries/base/Prelude.hs	Prelude.o, base library, and all dependent binaries are rebuilt	☐	Only Prelude.o is rebuilt	☑
U6: Modify code: libraries/base/Prelude.hs	Prelude.o and all its dependents are rebuilt	☑	Prelude.o and all its dependents are rebuilt	☑
U7: Add comment: utils/ghc-cabal/Main.hs	Almost everything is rebuilt	☐	All ghc-cabal rules are rerun	☐
U8: Modify code: utils/ghc-cabal/Main.hs	Almost everything is rebuilt	☐	Only the targets affected by the change are rebuilt	☑
U9: Modify the build system: pass -O2 when compiling Stage2 GHC	Nothing is rebuilt	☐	Stage2 GHC and its dependents are rebuilt	☑
U10: Modify the build system without changing command line arguments of build tools	Nothing is rebuilt	☐	Nothing is rebuilt	☐
U11: Change path to gcc	Everything is rebuilt	☐	Only gcc dependents are rebuilt	☑

Table 1. Comparison of GHC build systems on common use cases. Checkmarks ✓indicate desired behaviour.

6.1 Qualitative Analysis

In this section we discuss several use cases of the GHC build system, which are fairly typical for build systems in general. Table 1 lists use cases U1-U11 highlighting differences between the old and the new build systems. Below we go through some of the use cases in more detail. See §6.2 for performance comparison.

U1-U3 are simplest use cases. For the sake of fairness we start with U1, where the old build system reigns over our current implementation due to the aforementioned limitations. When unsupported features are not used (U2), the new build system successfully builds all expected targets. Running a build system twice in a row must be equivalent to only running it once; the second build must do nothing, hence the name *zero build* (U3). The new build system works as expected, and is faster than the old one (see §6.2).

In U4-U6 we modify libraries/base/Prelude.hs and rebuild GHC. If we touch the file (U4), i.e. change only its modification time, the new build system rebuilds nothing, as desired. The old build system rebuilds Prelude.o, the base library, and all dependent binaries, such as Stage2 GHC. This use case commonly occurs when switching git branches, as explained in §3.8, or whenever a user changes a file, but then decides to undo the changes. In U5 we add comments, forcing the new build system to recompile Prelude.hs. It then notices the object code is unchanged, and stops: there is nothing else to be done. The old build system continues to rebuild the base library and dependent binaries, which is unnecessary. In U6 the modification of Prelude.hs leads to changes in Prelude.o, which causes all dependencies to be rebuilt. Both build systems handle this case correctly.

U7-U8 are similar, but we now modify sources of the ghc-cabal build tool. The old build system rebuilds almost everything in both cases, which is unnecessary. Rebuilds are caused by rerunning the updated ghc-cabal binary, which changes modification time of generated package-data.mk files. The lack of polymorphic dependencies means we have to depend on the whole file when using Make, therefore even if only one field in a package-data.mk file is changed, e.g. CC_OPTS, we end up rebuilding everything, not only C compilation rules that depend on CC_OPTS. The new build system uses Shake's polymorphic dependencies §3.6 to avoid such unnecessary rebuilds. However, U7 behaviour is still suboptimal: ghc-cabal rules are rerun, because GHC currently produces nondeterministic output (ghc-cabal's binary is changed).

In U9 we modify the build system itself by changing command line arguments for one of the build files. The old build system rebuilds nothing, as Make does not track such changes. The new build system correctly reruns all affected rules. We currently only track command line arguments, therefore other, more subtle modifications of the build system (U10) go unnoticed, for example, adding a new need dependency does not cause a rebuild. In U11 we modify the build environment, by changing the path to gcc in the configuration file. As in U7-U8, the old build system rebuilds almost everything since depending on a single configuration setting is not supported. The new build system correctly reruns only affected rules.

In summary, the new build system correctly handles most use cases, whereas the old one performs a lot of unnecessary rebuilds in many cases.

6.2 Quantitative Analysis: Benchmarks

When building from scratch, ignoring the initial boot/configure steps which are shared, the old build system takes 1266 seconds on Windows and 649 seconds on Linux; while the new build system takes 737 seconds on Windows and 578 seconds on Linux. These were tested in as similar configurations as we could manage (by disabling all the features the new system does not support), but due to the complexity of the build systems, there are almost certainly minor differences. For the zero build, the old build system takes 12.3 seconds on Windows and 2.2 seconds on Linux; while the new one takes 2.1 seconds on Windows and 2.0 seconds on Linux. All measurements were collected using -j4.

Since Shake and Haskell both provide profiling and analysis tools, we have already used these features to optimise the new build system, resulting in modest gains so far and several opportunities we have yet to exploit. Looking at the current full Windows build time of 737 seconds, the longest single task is building the GMP library (315 seconds), and the total time of single-threaded computation is 2206 seconds, of which 2099 is calling out to external processes. The critical dependency chain has 378 steps in it, and requires 463 seconds – a lower bound on the clean build time even with an unlimited number of processors.

7. Related Work

This paper is about writing build systems at scale, the most complete modern advice on developing such systems comes from Smith (2011). When McIntosh et al. (2011) studied software maintenance they found that build systems can take up to 27% of the development effort, and that improvements to the build system rapidly paid off. Recently Martin et al. (2015) surveyed which Make features are used, and then Martin and Cordy (2016) classified them by complexity – unsurprisingly they found that as Makefiles grow, their complexity increases, and that hand written Makefiles require

most complex features. In the remainder of this section we focus on features found in other build systems which are useful at scale.

7.1 Embedded Language

A build system can either be specified using structured metadata, e.g. Bazel (Google 2016), or embedded into a standard programming language – for example SCons in Python (Knight 2005), Pluto in Java (Erdweg et al. 2015) and Jenga in OCaml (Jane Street Group 2016). For complex bespoke build systems, embedding into a language allows both complex operations (§4.7) and better abstractions (§5) – essentially allowing us to write most of our build system in a domain language tailored to our specific project.

Even sticking to Haskell as the embedded language, there are a surprisingly large number of libraries implementing a dependency aware build system – we know of eleven in addition to Shake (Abba, Blueprint, Buildsome, Coadjute, Cake $\times$ 2, Hake, Hmk, Nemesis, OpenShake and Zoom). Of these, the two Cake libraries and OpenShake are based on an early presentation of the principles behind Shake.

7.2 Advanced Dependencies

We have found that while powerful dependencies might only be used in a few places, if they are missing the workarounds can be pervasive (§4.6). A few build systems support resources, e.g. Ninja (Martin 2013), and several support monadic dependencies, e.g. Redo (Pennarun 2012), Jenga, Pluto, SCons. A few build systems directly support dependency features more powerful than Shake, e.g. Pluto supports rules that run until a fixed-point is reached and rules whose output filename is not known in advance. These features can be encoded in Shake, but are not present natively.

7.3 Automatic Dependency Management

In both Shake and Make, all dependencies must be declared explicitly. However, in build systems such as Tup (Shal 2009) and Buildsome (Lotem 2016), some dependencies are automatically captured by monitoring program execution. The Fabricate tool (Hoyt et al. 2009) takes a unique approach to defining build systems, providing a series of steps that run sequentially, but are skipped if their automatically-detected inputs have not changed. Unfortunately no cross-platform APIs are available to detect used dependencies, so such tools are all limited in which platforms they support.

7.4 Build Clusters

The build systems Bazel and Buck (Facebook 2016) are used at Google and Facebook respectively, both operating at sizes significantly beyond that of the GHC build system (reportedly billions of lines of code). Both systems take a metadata approach, with various rule types baked in. As an example, the `cxx_binary` rule builds a C/C++ binary given a list of source files and dependencies, taking care of suitable build flags and conventions, much like a very sophisticated version of `buildPackage` from §5.4.

The disadvantage of such an approach is that the available rules are fixed, making it difficult to encode something like a bootstrapping compiler. Generating build metadata is not really supported – a problem typically solved by committing the generated files to version control. Both tools also support build clusters, which build code once and share the resulting objects to everyone without recompiling them locally – an essential feature at such scales. Support for build clusters using Shake is made harder by the powerful dependencies, but we believe is still tractable, and hope to add support in a future version.

8. Conclusions and Future Work

We have demonstrated that Make really is unsuitable for large complex build systems, regardless of whether used recursively or non-recursively. Using Shake we have rewritten the GHC build system, producing the fifth and hopefully final version. While all previous versions have started simple and gained complexity as they progressed, this version is different. Developing the abstractions in §5 took many months of discussion and refinement. Once the fundamental concepts were in place, the rest was "just" coding and reverse engineering the existing system. The result is faster, more maintainable and more correct. There are three major tasks remaining for future work:

- While we have demonstrated that our approach works, we have not yet implemented all features of the build system, and hope to do so over the next few months. Once complete, we expect it to quickly become the only supported method of building GHC.
- Our abstractions from §5 were designed to allow tracking provenance of command line arguments – mapping each flag to the location of the expression that generated it. This feature will rely on the *implicit locations* feature of the latest GHC.
- While faster than the old system, the build is still slower than we would like. The zero build time could be reduced by switching to a faster serialisation library. The critical path of a full rebuild takes over seven minutes, limiting the gains available from additional processors. We hope to break this critical path by refactoring the build system, which is now a feasible task.

References

Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. A sound and optimal incremental build system with dynamic dependencies. In *Proc. of the ACM SIGPLAN Int'l Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 89–106. ACM, 2015.

Facebook. Buck. `https://buckbuild.com/`, 2016.

Google. Bazel. `http://bazel.io/`, 2016.

Berwyn Hoyt, Bryan Hoyt, and Ben Hoyt. fabricate – the better build tool. `https://github.com/SimonAlfie/fabricate`, 2009.

Jane Street Group. Jenga. `https://github.com/janestreet/jenga`, 2016.

Steven Knight. Building software with SCons. *Computing in Science and Engineering*, 7(1):79–88, 2005.

Eyal Lotem. Buildsome build system. `https://github.com/ElastiLotem/buildsome`, 2016.

Douglas H. Martin and James R. Cordy. On the maintenance complexity of Makefiles. In *Proceedings of the 2016 7th Interenational Workshop on Emerging Trends in Software Metrics*, WETSoM '16, 2016.

Douglas H. Martin, James R. Cordy, Bram Adams, and Giulio Antoniol. Make it simple: An empirical analysis of GNU Make feature use in open source projects. In *Proc. of the IEEE International Conference on Program Comprehension*, ICPC'15, pages 207–217. IEEE Press, 2015.

Evan Martin. Ninja. In Tavish Armstrong, editor, *The Performance of Open Source Applications*, chapter 3. 2013.

Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 141–150. ACM, 2011.

Peter Miller. Recursive make considered harmful. *Journal of AUUG Inc*, 19(1):14–25, 1998.

Neil Mitchell. Shake before building - replacing Make with Haskell. In *ICFP '12: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2012.

Andrey Mokhov and Victor Khomenko. Algebra of parameterised graphs. *ACM Transactions on Embedded Computing Systems*, 13(4s):143, 2014.

Avery Pennarun. redo: a top-down software build system. `https://github.com/apenwarr/redo`, 2012.

Mike Shal. Build system rules and algorithms. `http://gittup.org/tup/build_system_rules_and_algorithms.pdf`, 2009.

Peter Smith. *Software Build Systems*. Pearson Education, 2011.