Build Systems à la Carte

(Under review, feedback is sought)

ANDREY MOKHOV, Newcastle University, United Kingdom
NEIL MITCHELL, Digital Asset, United Kingdom
SIMON PEYTON JONES, Microsoft Research, United Kingdom

Build systems are awesome, terrifying – and unloved. They power developers around the world, but are rarely the object of study. In this paper we offer a systematic, and executable, framework for developing and comparing build systems, viewing them as related points in landscape rather than as isolated phenomena. By teasing apart existing build systems, we can recombine their components, allowing us to prototype the first build system that combines dynamic dependencies and cloud builds.

1 INTRODUCTION

Build systems (such as Make) are big, complicated, and used by every software developer on the planet. But they are a sadly unloved part of the software ecosystem, very much a means to an end, and seldom the focus of attention. Rarely do people ask questions like "What does it mean for my build system to be correct?" or "What are the trade-offs between different approaches?". Moreover, complex build systems use subtle algorithms, but they are often hidden away, and not the object of study. Recently, the challenges of scale have driven large software firms like Microsoft, Facebook, and Google to develop their own build systems, exploring new points in the design space.

In this paper we offer a general framework in which to understand and compare build systems, in a way that is both abstract (omitting incidental detail) and yet precise (implemented as Haskell code). Specifically we make these contributions:

- Build systems vary on many axes, including: static vs dynamic dependencies; local vs cloud; deterministic vs non-deterministic build rules; support for early cutoff; self-tracking build systems; and the type of persistent build information. In §2 we identify some key properties, in the context of four carefully-chosen build systems.
- We describe some simple but novel abstractions that crisply encapsulate what a build system is (§3), allowing us, for example, to speak about what it means for a build system to be correct.
- We identify two key design choices that are typically deeply wired into any build system: *the order in which dependencies are built* (§4.1) and *whether or not a dependency is (re-)built* (§4.2). These choices turn out to be orthogonal, which leads us to a new classification of the design space (§4.3).
- We show that we can instantiate our abstractions to describe the essence of a variety of different real-life build systems, including Make, Shake, Bazel, and Excel, each in a dozen lines of code or so (§5). Doing this modelling in a single setting allows the differences and similarities between these huge systems to be brought out clearly.
- Moreover, we can readily remix the ingredients to describe the first build system that supports both *dynamic dependencies* and *cloud build* (§5.5).

In short, instead of seeing build systems as unrelated points in space, we now see them as locations in a landscape, leading to a better understanding of what they do and how they compare, and

Authors' addresses: Andrey Mokhov, School of Engineering, Newcastle University, Newcastle upon Tyne, United Kingdom, andrey.mokhov@ncl.ac.uk; Neil Mitchell, Digital Asset, United Kingdom, ndmitchell@gmail.com; Simon Peyton Jones, Microsoft Research, Cambridge, United Kingdom, simonpj@microsoft.com.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

suggesting exploration of other (as yet unoccupied points) in the landscape. We discuss engineering aspects in §6, and related work in §7.

Papers about "frameworks" are often fuzzy. This one is not: all our abstractions are defined in Haskell, and we have (freely-available) executable models of all the build systems we describe.

2 BACKGROUND

Build systems automate the execution of simple repeatable tasks for individual users, as well as for large organisations. In this section we explore the design space of build systems, using four concrete examples: Make [Feldman 1979], Shake [Mitchell 2012], Bazel [Google 2016], and Excel [De Levie 2004][1]. We have carefully chosen these four to illustrate the various axes on which build systems differ; we discuss many other notable examples of build systems, and their relationships, in §7.

2.1 The venerable Make: static dependencies and file modification times

Make[2] was developed more than 40 years ago to automatically build software libraries and executable programs from source code. It uses *makefiles* to describe tasks (often referred to as *build rules*) and their dependencies in a simple textual form. For example:

```
util.o: util.h util.c
    gcc -c util.c

main.o: util.h main.c
    gcc -c main.c

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

The above makefile lists three tasks: (i) compile a utility library comprising files util.h and util.c into util.o by executing[3] the command gcc -c util.c, (ii) compile the main source file main.c into main.o, and (iii) link object files util.o and main.o into the executable main.exe. The makefile contains the complete information about the *task dependency graph*, which is shown in Fig. 1(a).

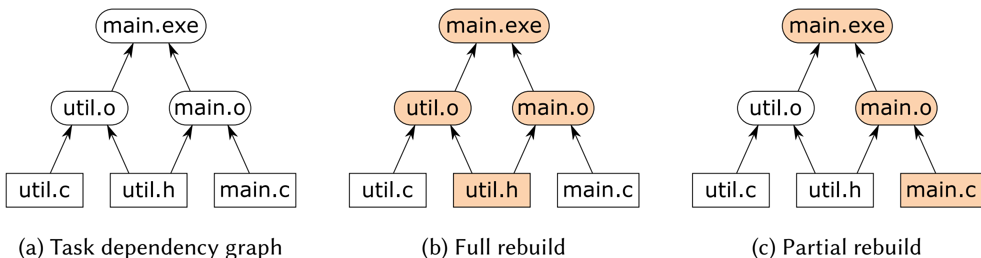


Fig. 1. A task dependency graph and two build scenarios. Input files are shown as rectangles, intermediate and output files are shown as rounded rectangles. Modified inputs and files that are rebuilt are highlighted.

If the user runs Make specifying main.exe as the desired output, Make will first build util.o and main.o, in any order since these tasks are independent, and then build main.exe. If the user modifies the sources of util.h and runs Make again, it will perform a *full rebuild*, because all three

[1] Excel appears very different to the others but, seen through the lens of this paper, it is very close indeed.
[2] There are numerous implementations of Make and none comes with a formal specification. In this paper we therefore use a simple and sensible approximation to a real Make that you might find on your machine.
[3] In this example we pretend gcc is a pure function for the sake of simplicity. In reality, there are multiple versions of gcc and the actual binary that is used to compile and link files is often also listed as a task dependency.

tasks transitively depend on util.h, as illustrated in Fig. 1(b). On the other hand, if the user modifies main.c then a *partial rebuild* is sufficient: the file util.o does not need to be rebuilt, since its inputs have not changed, see Fig. 1(c). Note that if the dependency graph is *acyclic* then each task needs to be executed at most once. Cyclic task dependencies are typically not allowed in build systems although there are rare exceptions, see §6.6.

The following property is essential for build systems, it is their raison d'être:

Definition 2.1 (Minimality). A build system is *minimal* if it executes tasks at most once per build and only if they transitively depend on inputs that changed since the previous build.

To achieve minimality MAKE relies on two main ideas: (i) it uses *file modification time* to detect which files changed[4], and (ii) it constructs a task dependency graph from the information contained in the makefile and executes tasks in a *topological order*. For a more concrete description see §5.1.

2.2 EXCEL: dynamic dependencies at the cost of minimality

EXCEL is a build system in disguise. Consider the following simple spreadsheet.

```
A1: 10      B1: A1 + A2
A2: 20
```

There are two input cells A1 and A2, and a single task that computes the sum of their values, writing the result into the cell B1. If either of the inputs change, EXCEL will recompute the result.

Unlike MAKE, EXCEL does not need to know all task dependencies upfront. Some dependencies may change *dynamically* according to computation results. For example:

```
A1: 10      B1: IF(C1=1,A1,A2)      C1: 1
A2: 20
```

Here the cell C1 controls which branch of the IF function is used to compute B1. When C1=1, the dependencies of B1 are {C1, A1}, otherwise they are {C1, A2}, which is not known statically[5].

EXCEL handles this example correctly: if C1=1 and the user changes A2, EXCEL will not require B1 to be computed in advance. Alas, other forms of dynamic dependencies can force EXCEL to perform unnecessary computation. Consider the following modification of the above example:

```
A1: 10      B1: INDIRECT("A" & C1)      C1: 1
A2: 20
```

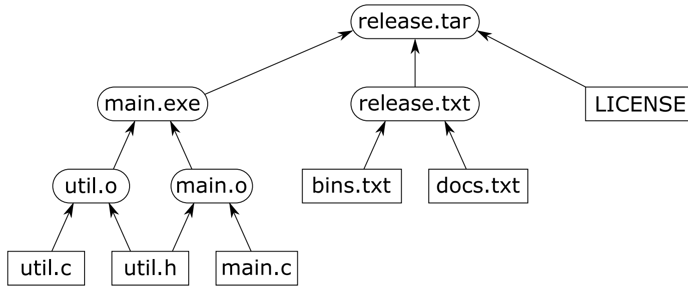
The new version uses the INDIRECT function, which allows us to reference a cell indirectly by a text string that is not necessarily known in advance. The current implementation of EXCEL recomputes indirect references in every build [Microsoft 2011]. This approach clearly violates the minimality property 2.1: if C1=1 and the user modifies A2, EXCEL will recompute B1, potentially triggering further unnecessary recalculation even though B1 does not transitively depend on A2.

EXCEL's build algorithm [Microsoft 2011] is significantly different from MAKE. EXCEL uses the *calculation chain* produced by the previous build as an approximation to the correct topological order. During recalculation, EXCEL processes cells in this order, but can *defer recalculation of a cell* by moving it down the chain if a newly discovered dependency has not yet been rebuilt. We refer to this algorithm as *reordering*, and will discuss it in more detail in §5.2.

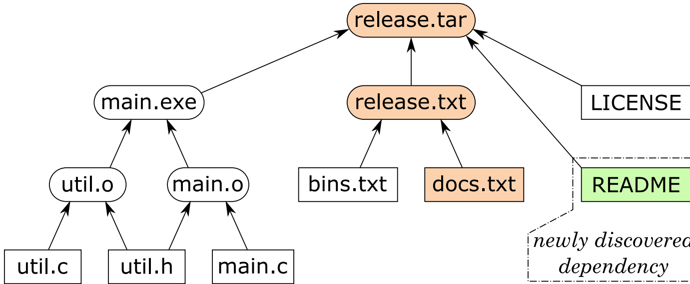
[4]Technically, you can fool MAKE by altering the modification time of a file without changing its content, e.g. by using the command touch. MAKE is therefore minimal only under the assumption that you do not do that.

[5]One might say that the value of C1 is statically known in this particular example, but imagine that it is the result of a long computation chain – its value will only become available during the build.

Another distinguishing feature of Excel is *self-tracking*. Most build systems only track changes of inputs and intermediate results, but Excel can also track changes in the tasks themselves: if a formula is modified, Excel will recompute it and propagate the changes. Self-tracking is uncommon in software build systems, where one often needs to manually initiate a full rebuild even if just a single build task has changed. We discuss self-tracking further in §6.5.



(a) Dependency graph produced after the previous build.



(b) Since docs.txt was modified, we rebuild release.txt and release.tar, discovering a new dependency.

Fig. 2. Dynamic dependencies example: create README and add it to the list of release documents docs.txt.

2.3 Shake: dynamic dependencies with no remorse

Shake was developed to solve the issue of dynamic dependencies [Mitchell 2012] without sacrificing the minimality requirement. Building on the Make example from §2.1, we add the following files whose dependencies are shown in Fig. 2(a):

- LICENSE is an input text file containing the project license.
- release.txt is a text file listing all files that should be in the release. This file is produced by concatenating input files bins.txt and docs.txt that list all binary and documentation files of the project.
- release.tar is the release archive built by executing the command tar on the release files.

The dependencies of release.tar are not known statically: they are determined by the content of release.txt, which might not even exist before the build. Makefiles cannot express such dependencies, requiring problematic workarounds such as *build phases* [Mokhov et al. 2016]. In Shake we can express the rule for release.tar as:

```
"release.tar" %> \_ -> do
    need ["release.txt"]
    files <- lines <$> readFile "release.txt"
    need files
    system "tar" $ ["-cf", "result.tar"] ++ files
```

We first declare the static dependency on release.txt, then read its content (a list of files) and depend on each listed file, dynamically. Finally, we specify the command to produce the resulting archive. Crucially, the archive will only be rebuilt if one of the dependencies (static or dynamic) has changed. For example, if we create another documentation file README and add it to docs.txt, SHAKE will appropriately rebuild release.txt and release.tar, discovering the new dependency, see Fig. 2(b).

SHAKE's implementation is different from both MAKE and EXCEL in two aspects. First, it uses the dependency graph from the previous build to decide which files need to be rebuilt. This idea has a long history, going back to *incremental* [Demers et al. 1981], *adaptive* [Acar et al. 2002], and *self-adjusting computations* (see [Acar et al. 2007] and §7). Second, instead of abandoning and deferring the execution of tasks whose newly discovered dependencies have not yet been built (as EXCEL does), SHAKE *pauses* their execution until the dependencies are brought up to date. We refer to this build algorithm as *recursive*.

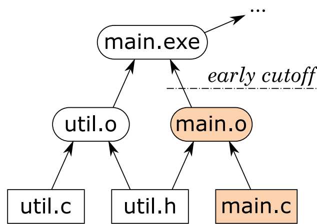


Fig. 3. An early cutoff example: if a comment is added to main.c, the rebuild is stopped after detecting that main.o is unchanged, since this indicates that main.exe and its dependents do not need to be rebuilt.

SHAKE also supports the *early cutoff optimisation*. When it executes a task and the result is unchanged from the previous build, it is unnecessary to execute the dependent tasks, and hence SHAKE can stop a build earlier, as illustrated in Fig. 3. Not all build systems support early cutoff: MAKE and EXCEL do not, whereas SHAKE and BAZEL (introduced below) do.

2.4 BAZEL: a cloud build system

When build systems are used by large teams, different team members often end up executing exactly the same tasks on their local machines. A *cloud build system* can speed up builds dramatically by sharing build results among team members. Furthermore, cloud build systems allow one to perform *shallow builds* that materialise only end build products locally, leaving all intermediates in the cloud. We illustrate shallow cloud builds by an example in Fig. 4.

The user starts by checking out the project sources, whose hashes are (for simplicity) 1, 2 and 3, and requests to build main.exe, see Fig. 4(a,b). By looking up the global history of all previous builds of the project[6], the build system finds that someone has already compiled these exact sources before and their resulting files util.o and main.o had hashes 4 and 5. Similarly, the build system finds that the hash of the resulting main.exe should be 6 and downloads the actual binary from the shared cloud storage, since it is the end build product and must therefore be materialised.

In the second scenario, shown in Fig. 4(c), the user modifies the source util.c, thereby changing its hash from 1 to 7. The build system finds that nobody has ever compiled the new {util.c, util.h} combination and must therefore build util.o, which results in changing its hash from 4 to 8. The combination of hashes of util.o and main.o has not been encountered before either, therefore the build system first downloads main.o from the cloud and then builds main.exe by linking the two object files. When the build is complete, the results can be uploaded to the cloud for future reuse by other team members.

[6] Here we ignore the issue of limited cloud storage resources for the sake of simplicity; in practice, old entries are regularly evicted from the storage, as further discussed in §6.4.

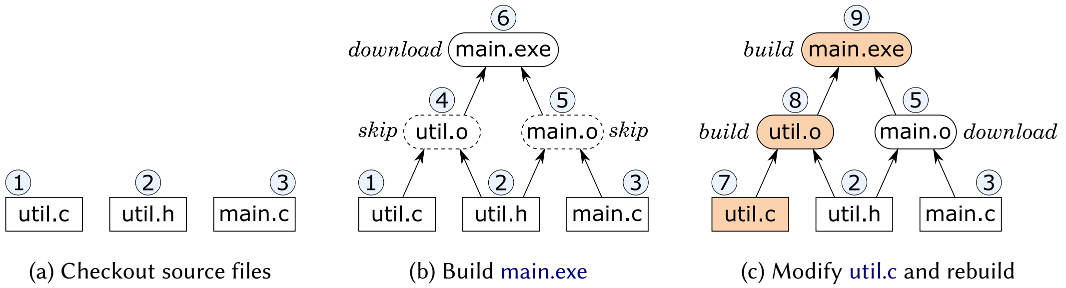


Fig. 4. A cloud build example: (a) checkout sources, (b) download main.exe from the cloud and skip intermediate files (only their hashes are needed), (c) modify util.c and rebuild main.exe, which requires building util.o (since nobody has compiled util.c before) and downloading main.o. File hashes are shown inside circles.

Bazel is one of the first examples of openly available cloud build systems. Like Make, it does not support dynamic dependencies and can therefore benefit from the simplicity of building tasks in a statically known topological order. It is minimal and supports the early cutoff optimisation. To support cloud builds, Bazel maintains a *content-addressable cache* that can be used to fetch a previously built file given the hash of its content, and *dependency graphs from all previous builds*, annotated with observed file hashes. The latter allows builds to bypass the execution of a task, by predicting the hash of the result from the hashes of its dependencies, and subsequently fetch the result from the cache. A concrete implementation is provided in §5.

2.5 Summary

We summarise differences between four discussed build systems in Table 1. The column *'persistent build information'* refers to the information that build systems persistently store between builds:

- Make stores file modification times, or rather, it relies on the file system to do that.
- Excel stores one dirty bit per cell and the calculation chain from the previous build.
- Shake stores the dependency graph discovered in the previous build, annotated with file content hashes for efficient checking of file changes.
- Bazel stores all dependency graphs discovered in previous builds annotated with file hashes, and the content-addressable cache.

Build system	Persistent build information	Algorithm	Dependencies	Minimal	Cutoff	Cloud
Make	File modification times	Topological	Static	Yes	No	No
Excel	Dirty cells, calculation chain	Reordering	Dynamic	No	No	No
Shake	Previous dependency graph	Recursive	Dynamic	Yes	Yes	No
Bazel	All dependency graphs, cache	Topological	Static	Yes	Yes	Yes

Table 1. Summary of build system differences.

In this paper we elucidate which build system properties are consequences of specific implementation choices (metadata and algorithm), and how one can obtain new build systems with desired properties by recombining parts of existing implementations. As a compelling example, we demonstrate how to combine the advantages of Shake and Bazel in a cloud build system with dynamic dependencies, see §5.5.

```
-- An abstract store
data Store i k v
getInfo    :: Store i k v -> i
putInfo    :: i -> Store i k v -> Store i k v
getValue   :: k -> Store i k v -> v
putValue   :: Eq k => k -> v -> Store i k v -> Store i k v
getHash    :: Hashable v => k -> Store i k v -> Hash v
initialise :: i -> (k -> v) -> Store i k v

-- Hashing
hash :: Hashable a => a -> Hash a

-- Applicative functors
pure  :: Applicative f => a -> f a
(<$>) :: Applicative f =>   (a -> b) -> f a -> f b -- Left-associative
(<*>) :: Applicative f => f (a -> b) -> f a -> f b -- Left-associative

-- Standard State monad from Control.Monad.State
data State s a
instance Applicative (State s)
instance Monad       (State s)
get       :: State s s
gets      :: (s -> a) -> State s a
modify    :: (s -> s) -> State s ()
execState :: State s a -> s -> s

-- Standard types from Data.Functor.Identity and Data.Functor.Const
newtype Identity a = Identity { runIdentity :: a }
newtype Const m a  = Const    { getConst    :: m }

instance Functor (Const m) where
    fmap _ (Const m) = Const m

instance Monoid m => Applicative (Const m) where
    pure _              = Const mempty
    Const x <*> Const y = Const (x <> y)
```

Fig. 5. Signatures of main data types and library functions.

3 BUILD SYSTEMS, ABSTRACTLY

This section presents purely functional abstractions that allow us to express all the intricacies of build systems discussed previously in §2, and design complex build systems from simple primitives. Specifically, we present the *task* and *build* abstractions in §3.2 and §3.3, respectively. Sections §4 and §5 scrutinise the abstractions further and provide concrete implementations for several build systems.

3.1 Common vocabulary for build systems

We begin by establishing a common vocabulary for build systems:

Keys, values, and the store. The goal of any build system is to bring up to date a *store* that implements a mapping from *keys* to *values*. In software build systems the store is the file system, the keys are filenames, and the values are file contents. In Excel, the store is the worksheets, the keys are cell names (such as A2) and the values are numbers, strings etc, displayed as the cell contents. Many build systems use *hashes* of values as compact summaries with a fast equality check.

Input, output, and intermediate values. Some values must be provided by the user as *input*. For example, main.c can be edited by the user who relies on the build system to compile it into main.o and subsequently main.exe. End build products, such as main.exe, are *output* values. All other values (in this case main.o) are *intermediate*; they are not interesting for the user but are produced in the process of turning inputs into outputs.

Persistent build information. As well as the key/value mapping, the store also contains information maintained by the build system itself, which persists from one invocation of the build system to the next – its "memory".

Task description. Any build system requires the user to specify how to compute the new value for one key, using the (up to date) values of its dependencies. We call this specification the *task description*. For example, in Excel, the formulae of the spreadsheet constitute the task description; in Make the rules in the makefile are the task description.

Build system. A *build system* takes a task description, a *target* key, and a store, and returns a new store in which the target key and all its dependencies have an up to date value.

Modelling in Haskell. We will model all our build systems concretely, as Haskell programs. To that end, Fig. 5 gives the type declarations and function signatures of the library functions. For example, `Store i k v` is the type of stores, with several associate functions (`getValue`, etc.). We use `k` as a type variable ranging over keys, `v` for values, and `i` for the persistent build information.

3.2 The Task abstraction

Our first main abstraction is for *task descriptions*:

```
type Task c k v = forall f. c f => (k -> f v) -> k -> Maybe (f v)
```

This highly-abstracted type[7] is best introduced by an example. Consider this Excel spreadsheet:

```
A1: 10     B1: A1 + A2
A2: 20     B2: B1 * 2
```

Here cell A1 contains the value 10, cell B1 contains the formula A1+A2, etc. We can represent the formulae of this spreadsheet with the following task description:

```
sprsh1 :: Task Applicative String Integer
sprsh1 fetch "B1" = Just ((+)   <$> fetch "A1" <*> fetch "A2")
sprsh1 fetch "B2" = Just ((* 2) <$> fetch "B1")
sprsh1 _     _    = Nothing
```

We instantiate keys `k` with `String`, and values `v` with `Integer`. (Real spreadsheet cells would contain a wider range of values, of course.) The task description `sprsh1` embodies all the *formulae* of the spreadsheet, but not the input values. Like every `Task`, `sprsh1` is given a *callback* `fetch` and a key. It pattern-matches on the key to see if it has a task description (in the Excel case, a formula) for it. If not, it returns `Nothing`, indicating the key is an input. If there is a formula in the cell, it computes the value of the formula, using `fetch` to find the value of any keys on which it depends.

The code to "compute the value of a formula" in `sprsh1` looks a bit mysterious because it takes place in an `Applicative` computation [McBride and Paterson 2008] – the relevant type signatures are given in Fig. 5. We will explain why in subsection §3.3.

For now, we content ourselves with observing that a task description, of type `Task c k v`, is completely isolated from the world of compilers, calculation chains, file systems, caches, and all other complexities of real build systems. It just computes a single output, in a side-effect-free way, using a callback (`fetch`) to find the values of its dependencies.

[7] Readers familiar with *lenses* or *profunctor optics* might recognise a familiar pattern. We discuss this in §7.4.

393
3.3 The Build abstraction

Next comes our second main abstraction – a build system:

```
type Build c i k v = Task c k v -> k -> Store i k v -> Store i k v
```

The signature is very straightforward. Given a task description, a target key, and a store, the build system returns a new store in which the value of the target key is up to date. What exactly does "up to date" mean? We answer that precisely in §3.6. Here is a simple build system:

```
busy :: Eq k => Build Monad () k v
busy task key store = execState (fetch key) store
  where
    fetch :: k -> State (Store () k v) v
    fetch k = case task fetch k of
        Nothing  -> gets (getValue k)
        Just act -> do v <- act; modify (putValue k v); return v
```

The busy build system defines the callback fetch so that, when given a key, it brings the key up to date in the store, and returns its value. The function fetch runs in the standard Haskell **State** monad – see Fig. 5 – initialised with the incoming store by execState. To bring a key up to date, fetch asks the task description task how to compute k. If task returns **Nothing** the key is an input, so fetch simply reads the result from the store. Otherwise fetch runs the action act returned by the task to produce a resulting value v, records the new key/value mapping in the store, and returns v. Notice that fetch passes itself to task as an argument, so that the latter can use fetch to recursively find the values of k's dependencies.

Given an acyclic task description, the busy build system terminates with a correct result, but it is not a *minimal* build system (Definition 2.1). Since busy has no memory (i = ()), it cannot keep track of keys it has already built, and will therefore busily recompute the same keys again and again if they have multiple dependents. We will develop much more efficient build systems in §5.

Nevertheless, busy can easily handle the example sprsh1 from the previous subsection §3.2. In the GHCi session below we initialise the store with A1 set to 10 and all other cells set to 20.

```
λ> store  = initialise () (\key -> if key == "A1" then 10 else 20)
λ> result = busy sprsh1 "B2" store
λ> getValue result "B1"
30
λ> getValue result "B2"
60
```

As we can see, busy built both B2 and its dependency B1 in the right order (if it had built B2 before building B1, the result would have been $20 * 2 = 40$ instead of $(10 + 20) * 2 = 60$). As an example showing that busy is not minimal, imagine that the formula in cell B2 was B1 + B1 instead of B1 * 2. This would lead to calling fetch "B1" twice – once per occurrence of B1 in the formula.

3.4 The need for polymorphism in Task

The previous example shows why the **Task** abstraction is polymorphic in f, recall the definition:

```
type Task c k v = forall f. c f => (k -> f v) -> k -> Maybe (f v)
```

The busy build system instantiates f to **State** (**Store** i k v), so that fetch :: k -> f v can side-effect the **Store**, thereby allowing successive calls to fetch to communicate with one another.

We really, really want **Task** to be *polymorphic* in f. Given *one* task description T, we want to explore *many* build systems that can build T – and we will do so in sections §4 and §5. As we shall see, each build system will use a different f, so the task description must not fix f.

But nor can the task description work for *any* f; most task descriptions (e.g. sprsh1 in §3.2) require that f satisfies certain properties, such as **Applicative** or **Monad**. That is why **Task** has the "c f =>" constraint in its type, expressing that f can only be instantiated by types that satisfy the constraint c.

So the type **Task** emerges naturally, almost inevitably. But now that it *has* emerged, we find the that constraints c classify task descriptions in a very interesting way:

- **Task Applicative**. In sprsh1 we needed only **Applicative** operations, expressing the fact that the dependencies between cells can be determined *statically*; that is, by looking at the formulae, without "computing" them (see §3.7).
- **Task Monad**. As we shall see in §3.5, a monadic task description allows *dynamic* dependencies, in which a formula may depend on the value of cell C, but *which* cell C depends on the value of another cell D.
- **Task Functor** is somewhat degenerate: the task description cannot even use the application operator <*>, which limits dependencies to a single linear chain. It is interesting to note that, when run on a **Task Functor**, the busy build system will build each key at most once, thus partially fulfilling the minimality requirement 2.1. Alas, it still has no mechanism to decide which input keys changed since the previous build.
- **Task Alternative**, **Task MonadPlus** and their variants can be used for describing tasks with a certain type of non-determinism, as discussed in §6.3.

Notice also that, even though busy takes a **Task Monad** as its argument, an application of busy to a **Task Functor** or a **Task Applicative** will typecheck and run just fine. It feels a bit like sub-typing, but is actually just ordinary higher-rank polymorphism at work [Peyton Jones et al. 2007].

3.5 Monadic tasks

As explained in §2.2, some task descriptions have dynamic dependencies, which are determined by values of intermediate computations. In our framework, such task descriptions correspond to the type **Task Monad** k v. Consider this spreadsheet example:

```
A1: 10      B1: IF(C1=1,B2,A2)      C1: 1
A2: 20      B2: IF(C1=1,A1,B1)
```

Note that B1 and B2 statically form a dependency cycle, but Excel (which uses dynamic dependencies) is perfectly happy. We can express this spreadsheet using our task abstraction as:

```haskell
sprsh2 :: Task Monad String Integer
sprsh2 fetch "B1" = Just $ do c1 <- fetch "C1"
                              if c1 == 1 then fetch "B2" else fetch "A2"
sprsh2 fetch "B2" = Just $ do c1 <- fetch "C1"
                              if c1 == 1 then fetch "A1" else fetch "B1"
sprsh2 _     _    = Nothing
```

The big difference compared to sprsh1 is that the computation now takes place in a **Monad**, which allows us to extract the value of c1 and fetch *different keys depending on whether or not* c1 == 1.

Since the busy build system introduced in §3.3 always rebuilds every dependency it encounters, it is easy for it to handle dynamic dependencies. For minimal build systems, however, dynamic dependencies, and hence monadic tasks, are much more challenging, as we shall see in §5.

3.6 Correctness of a build system

We can now say what it means for a build system to be *correct*, something that is seldom stated formally. Our intuition is this: *when the build system completes, the target key, and all its dependencies, should be up to date*. What does "up to date" mean? It means that if we recompute the value of the key (using the task description, and the final store), we should get exactly the same value as we see in the final store.

To express this formally we need an auxiliary function `compute`, that computes the value of a key in a given store *without attempting to update any dependencies*:

```
compute :: Task Monad k v -> Store i k v -> k -> Maybe v
compute task store = fmap runIdentity . task (\k -> Identity (getValue k store))
```

Here we do not need any effects in the `fetch` callback to `task`, so we can use the standard Haskell `Identity` monad (Fig. 5). The use of `Identity` just fixes the 'impedance mismatch' between the function `getValue`, which returns a pure value `v`, and the `fetch` argument of the `task`, which must return an `f v` for some `f`. To fix the mismatch, we wrap the result of `getValue` in the `Identity` monad: the function `\k -> Identity (getValue k store)` has the type `k -> Identity v`, and can now be passed to a `task`. The result comes as `Maybe (Identity v)`, hence we now need to get rid of the `Identity` wrapper by applying `runIdentity` to the contents of `Maybe`.

Definition 3.1 (Correctness). Suppose `build` is a build system, `task` is a build task description, `key` is a target key, `store` is an initial store, and `result` is the store produced by running the build system with parameters `task`, `key` and `store`. Or, using the precise language of our abstractions:

```
build         :: Build c i k v
task          :: Task c k v
key           :: k
store, result :: Store i k v
result = build task key store
```

The keys that are reachable from the target `key` via dependencies fall into two classes: input keys and non-input keys, which we will denote by I and O, respectively. Note that `key` may be in either of these sets, although the case when `key` is an input is degenerate: we have $I = \{$`key`$\}$ and $O = \emptyset$.

The build `result` is *correct* if the following two conditions hold:

- `store` and `result` *agree on inputs*, that is, for all input keys `k` $\in I$:

  `getValue k store == getValue k result`.

  In other words, no inputs were corrupted during the build.
- The `result` is *consistent* with the `task`, i.e. for all non-input keys `k` $\in O$, the result of recomputing the `task` matches the value stored in the `result`:

  `Just (getValue k result) == compute task result k`.

A build system is *correct* if it produces a correct `result` for any given `task`, `key` and `store`.

It is hard to satisfy the above definition of correctness given a task description with cycles. All build systems discussed in this paper are correct only under the assumption that the given task description is acyclic. This includes the `busy` build system introduced earlier: it will loop indefinitely given a cyclic `task`. Some build systems provide a limited support for cyclic tasks, see §6.6.

The presented definition of correctness needs to be adjusted to accommodate non-deterministic tasks and shallow cloud builds, as will be discussed in sections §6.3 and §6.4, respectively.

3.7 Computing dependencies

Earlier we remarked that a Task Applicative could only have static dependencies. Usually we would extract such static dependencies by (in the case of Excel) looking at the syntax tree of the formula. But a task description has no such syntax tree. Yet, remarkably, we can use the polymorphism of a Task Applicative to find its dependencies *without doing any of the actual work*. Here is the code:

```
dependencies :: Task Applicative k v -> k -> [k]
dependencies task key = case task (\k -> Const [k]) key of
                            Nothing         -> []
                            Just (Const ks) -> ks
```

Here Const is a standard Haskell type defined in Fig. 5. We instantiate f to Const [k]. So a value of type f v, or in this case Const [k] v, contains no value v, but does contain a list of keys of type [k] which we use to record dependencies. The fetch callback that we pass to task records a single dependency; and the standard definition of Applicative for Const (which we give in Fig. 5) combines the dependencies from different parts of the task. Running the task with f = Const [k] will thus accumulate a list of the task's dependencies – and that is just what dependencies does:

```
λ> dependencies sprsh1 "A1"
[]
λ> dependencies sprsh1 "B1"
["A1", "A2"]
```

Notice that these calls to dependencies do no actual computation (in this case, spreadsheet arithmetic). They cannot: we are not supplying a store or any input numbers. So, through the wonders of polymorphism, we are able to extract the dependencies of the spreadsheet formula, and to do so efficiently, simply by running its code in a different Applicative! This is not new, for example see [Capriotti and Kaposi 2014], but it is cool.

So much for applicative tasks. What about monadic tasks with dynamic dependencies? As we have seen in §2.3, dynamic dependencies need to be tracked too. This cannot be done statically; notice that the application of the function dependencies to a Task Monad will not typecheck. We need to run a monadic task on a store with concrete values, which will determine the discovered dependencies. Accordingly, we introduce the function track: a combination of compute and dependencies that computes both the resulting value and the list of its dependencies in an arbitrary monadic context m:

```
import Control.Monad.Writer

track :: Monad m => Task Monad k v -> (k -> m v) -> k -> Maybe (m (v, [k]))
track task fetch = fmap runWriterT . task trackingFetch
  where
    trackingFetch :: k -> WriterT [k] m v
    trackingFetch k = tell [k] >> lift (fetch k)
```

This implementation uses the standard Haskell WriterT *monad transformer* [Liang et al. 1995], which allows us to record additional information – a list of keys of type [k] – when computing a task in an arbitrary monad m. We substitute the given fetch with a trackingFetch that, in addition to fetching a value, tracks the corresponding key. The task returns the value of type Maybe (WriterT [k] m v), which we unwrap by applying runWriterT to the contents of Maybe. Below we give an example of tracking monadic tasks when m = IO.

```
λ> fetchIO k = do putStr (k ++ ": "); read <$> getLine
λ> fromJust $ track sprsh2 fetchIO "B1"
C1: 1
B2: 10
(10,["C1","B2"])

λ> fromJust $ track sprsh2 fetchIO "B1"
C1: 2
A2: 20
(20,["C1","A2"])
```

As expected, the dependencies of the cell `B1` from `sprsh2` (see the spreadsheet in §3.5) are determined by the value of `C1`, which in this case is obtained by reading from the standard input.

4 BUILD SYSTEMS À LA CARTE

The focus of this paper is on a variety of implementations of `Build c i k v`, given a *client-supplied* implementation of `Task c k v`. That is, we are going to take `Task` as given from now on, and explore variants of `Build`: first abstractly (in this section) and then concretely in §5.

As per the definition of minimality 2.1, a minimal build system must **rebuild only out-of-date keys** and at most once. The only way to achieve the "at most once" requirement while producing a correct build result (§3.6) is to **build all keys in an order that respects their dependencies**.

We have bolded two different phrases above, and tackle each aspect separately.

4.1 Respecting the dependency order

The build systems overview (§2.5) highlighted three distinct approaches to respecting the dependency order. This subsection explores their properties and possible implementations.

4.1.1 Topological. The topological approach pre-computes a linear order, which when followed, ensures the `build` is correct regardless of the initial `store`. Given a function from a key to its dependencies, and the output `key`, you can compute the linear order by first finding the reachable dependencies of `key`, and then computing a topological sort. However, as we have seen in §3.7, we can only extract dependencies from an applicative task, which requires the build system to choose `c = Applicative`, ruling out dynamic dependencies.

4.1.2 Reordering. The topological approach has two downsides: it is limited to `Applicative` build systems and requires a fresh topological sort each time. So, while the actions themselves may be incremental (i.e. unnecessary tasks will not be performed), the pre-processing is not. We can try to incrementalise the topological sort by storing the topological order between build runs and assume it to be correct, but if the build discovers it is wrong, fix it up.

This approach requires a way to abort tasks that have failed due to out-of-date dependencies. It is also not minimal in the sense that a task may start, do some meaningful work, then abort. However, in the case of an `Applicative` system, that work is zero.

4.1.3 Recursive. An alternative approach, utilised by the `busy` build system (§3.3), is to simply build dependencies when they are requested. By combining that with a transient set of which keys have already been built, you can obtain a minimal build system.

This approach requires that a task may be started, then during that execution another task will have to be run. Assuming an IO-driven task structure, that requires suspending a running task, which can be done with cheap green threads and blocking (the original approach of Shake) or using continuation-passing style (what Shake does currently). An alternative approach to suspending a task is to abort it and restart it again later, at the cost of doing additional work.

4.2 Determining out-of-date keys

The second aspect, determining what to rebuild, can be addressed in one of three fundamental ways, with a number of tweaks and variations within them.

4.2.1 A dirty bit. The idea of a dirty bit is to have one piece of persistent information per key, saying whether the key is *dirty* or *clean*. After a build, all bits are set to clean. When the next build starts, anything that changed between the two states is marked dirty. When reaching a key, if it and all its transitive dependencies are clean, the key does not need recomputing.

Excel models the dirty bit approach most directly, having an actual dirty bit associated with each cell, marking the cell dirty if the user modifies it. When rebuilding, if a cell only depends on clean cells it is skipped, otherwise it is rebuilt and marked dirty so that the cells that depend on it are subsequently rebuilt too.

Make uses file modification times, and compares files to their dependencies, which can be thought of as a dirty bit which is set when a file is newer than its dependencies. The interesting property of this dirty bit is that it is not under the control of Make; rather it is existing file-system information that has been repurposed. In particular, modifying a file automatically clears its dirty bit, and automatically sets the dirty bit of the nodes depending on it. One thing Make does require is that file timestamps only go forward in time – something that can be violated by backup software.

When using a dirty bit, it is necessary to check all the dependencies of a key. For applicative build systems that list is easy to obtain, but for monadic build systems there is no general way to get all dependencies. Instead Excel computes a *static approximation* of the dependencies. For applicative tasks that approximation is correct. For functions such as `IF` it marks the cell dirty if *any* potential dependency has changed, even on the untaken *if* branch. For functions such as `INDIRECT` whose dependencies cannot be guessed, it conservatively assumes the dependencies have always changed.

With a dirty bit it is simple to achieve minimality. However, to achieve early cutoff (§2.3) it would be important to not set the dirty bit after a computation that did not change the value. Excel could use this approach, but does not. In contrast, Make cannot implement early cutoff nicely – to do so it would have to mark the node clean (so it would not rebuild in the next run) and at the same time not mark the things it depends on dirty – an impossible task with only the ability to update to the latest modification time. Make can approximate early cutoff by not modifying the result file, and not marking it clean, but then it will rerun in every subsequent build.

4.2.2 Verifying traces. An alternative way to determine if a key is dirty is to record what state the values/hashes of dependencies were used at last time, and if something has changed, the key is dirty and must be recomputed – in essence a *trace* which we can use to *verify* existing values. We can describe a trace as:

```
data Trace k v = Trace
    { key          :: k
    , dependencies :: [(k, Hash v)]
    , result       :: Hash v }
```

We assume that `Hash v` is a small constant size, constructed from hashing the underlying `v` rather than storing it directly. Checking a trace requires ensuring all the dependencies are up to date (using whatever ordering strategy as per §4.1), then comparing if the dependencies are same as the current value and that the result is the same.

A build system that uses verifying traces needs to persistently maintain a set of traces. After computing a fresh value we add its `Trace k v` to the set. Therefore the information stored by a build system that verifies traces can be modelled as a list or set – we chose `[Trace k v]` for simplicity. In practice, different build systems can optimise the data structures used by traces for their specific use cases, which we discuss in §5.6.

4.2.3 *Constructive traces.*

A verifying trace allows us to mark a key dirty and rebuild it. Extending that information we can store a *constructive* trace which is the trace plus the actual result. Once we are storing the complete result it makes sense to record many constructive traces per key, and to share them with other users, providing cloud-build functionality. We can represent that as:

```
data Traces k v = Traces
    { traces    :: [Trace k v]
    , contents  :: Map (Hash v) v }
```

We have a list of traces, plus a `Map` from the hash to the actual contents. Checking a trace is the same as before, but if the result is the only thing that is different we can simply retrieve a fresh result from `contents` *without* recomputing it. We split the traces and contents because in cloud interactions to a remote server the checking system may have to examine many traces/hashes, but only retrieve at most one complete file per key.

4.3 Build Systems à la Carte

Property		Topological §4.1.1	Reordering §4.1.2	Recursive §4.1.3
Dirty bit	§4.2.1	MAKE	EXCEL	Approximate SHAKE *
Verifying trace	§4.2.2	NINJA	Traced EXCEL *	SHAKE
Constructive trace	§4.2.3	BAZEL	Cloud EXCEL *	Cloud SHAKE *

Table 2. Build systems à la carte. Systems marked * are hypothetical systems that do not currently exist.

With the information in this section we can build a table comparing the dependency order strategy with the out-of-date keys strategy, providing 9 possible build systems, 5 of which are actually inhabited by existing build systems (we discuss NINJA [Martin 2017] in §7.1). Of the remaining 4 spots, we believe neither Traced or Cloud EXCEL make sense – the EXCEL approach of reordering combined with static approximations reduces the memory usage significantly. However, as soon as you are paying the cost of storing traces, that benefit is gone. The advantage of Approximate SHAKE over SHAKE would be that it could avoid storing traces and having a separate information database, but that advantage is minor compared to the technical restrictions and approximations it would provide, so we consider it unlikely to be built. The Cloud SHAKE system is an interesting and important point in the design space, which we explore further in §5.5.

5 BUILD SYSTEMS, CONCRETELY

In the previous sections we discussed the types of build systems, and how they can be broken down. But these divisions were not obvious to us, and only by concretely implementing and refactoring each build system did we determine the underlying commonalities. In this section we share some of the code that got us there.

5.1 MAKE

We provide an implementation of MAKE using our framework in Fig. 6. MAKE processes keys in a linear order based on a topological sort (see §6.2 for parallel MAKE). For each key, it builds it if it is older than any of its dependencies. We capture the persistent build information that MAKE stores by a pair (`modTime`, `now`) comprising the *file modification time* function `modTime :: k -> Time` and the *current time* `now`. Setting aside the explicit manipulation of file modification times, which in reality is taken care of by the file system, the function `make` captures the essence of MAKE in a clear and precise manner.

```
-- Persistent build information
type Time       = Integer
type MakeInfo k = (k -> Time, Time)

-- Make build system
make :: Eq k => Build Applicative (MakeInfo k) k v
make = topological process
  where
    process key deps act = do
        (modTime, now) <- gets getInfo
        let dirty = or [ modTime dep > modTime key | dep <- deps ]
        when dirty $ do
            v <- act
            let newModTime k = if k == key then now else modTime k
            modify $ putInfo (newModTime, now + 1) . putValue key v

-- Standard graph algorithms (implementation omitted)
reachable :: Eq a => (a -> [a]) ->  a  -> [a]
topSort   :: Eq a => (a -> [a]) -> [a] -> [a]

-- Topological dependency strategy
topological :: Eq k => (k -> [k] -> State (Store i k v) v -> State (Store i k v) ())
            -> Build Applicative i k v
topological process task key = execState $ forM_ chain $ \k -> do
    let fetch k = gets (getValue k)
    case task fetch k of
        Nothing  -> return ()
        Just act -> process k (deps k) act
  where
    deps  = dependencies task -- dependencies is defined in §3.7
    chain = topSort deps (reachable deps key)
```

Fig. 6. An implementation of Make using our framework.

The helper function topological calls process on every key in a topological order, providing the list of its dependencies deps and the action act to compute the resulting value if it needs to be rebuilt. To determine if the key is dirty, process compares its modification time with those of its dependencies. If the key needs to be rebuilt, the action act is executed and the obtained result is stored, along with an updated file modification timestamp.

The implementation of topological encodes the dependency strategy that Make has chosen to use. The **where** clause corresponds to the pre-processing stage, which uses the function dependencies, defined in §3.7, to extract static dependencies from the provided applicative task. We compute the linear processing chain by taking the keys reachable from key via dependencies, and performing the topological sort of the result. We omit implementation of textbook graph algorithms reachable and topSort, e.g. see [Cormen et al. 2001].

The chain is processed in the **State** monad, with each non-input key k in the chain passed to the provided process function, along with k's dependencies and the action act, which when executed recomputes the k's value by fetching its dependencies from the store.

Note that dependencies is only defined for applicative tasks, which is what restricts Make to static dependencies, as reflected in the type **Build Applicative**. Moreover, any other build system following the same topological approach will also inherit the same restriction.

```haskell
-- Approximation of task dependencies
data DependencyApproximation k = SubsetOf [k] | Unknown

-- Persistent build information
type CalcChain k = [k]
type ExcelInfo k = ((k -> Bool, k -> DependencyApproximation k), CalcChain k)

-- Result of speculative task execution
data Result k v = MissingDependency k | Result v [k]

-- Reordering dependency strategy (implementation omitted, 21 lines)
reordering :: Ord k
    => (k -> State (Store i k v) (Result k v) -> State (Store i k v) (Maybe (Result k v)))
    -> Build Monad (i, CalcChain k) k v

-- Excel build system
excel :: Ord k => Build Monad (ExcelInfo k) k v
excel = reordering process
  where
    process key act = do
        (dirty, deps) <- gets getInfo
        let rebuild = dirty key || case deps key of SubsetOf ks -> any dirty ks
                                                    Unknown     -> True
        if not rebuild
            then return Nothing
            else do
                result <- act
                case result of
                    MissingDependency _ -> return ()
                    Result v dynamicDependencies -> do
                        let newDirty k = if k == key then True else dirty k
                        modify $ putInfo (newDirty, deps) . putValue key v
                return (Just result)
```

Fig. 7. An implementation of Excel using our framework.

5.2 Excel

We define Excel, with it's reordering dependency strategy, in Fig. 7. Excel's persistently stored information is a triple: (i) the dirty bit function `k -> Bool`, (ii) an approximation of key dependencies `k -> DependencyApproximation k` that Excel uses to handle dynamic dependencies, and (iii) the calculation chain `[k]` recorded in the previous build (§2.2).

The helper function `reordering`, whose implementation we omit since it is technical and not particularly enlightening, calls the function `process` to *try to build* a `key` by executing the action `act`, in the order determined by the calculation chain. To decide whether to `rebuild` the `key`, `process` checks if the `key` itself is marked dirty or the approximation of its dependencies contains a dirty key. Notice that if the dependencies of the `key` are `Unknown`, e.g. when it uses the INDIRECT function, the `key` is always rebuilt. If a `rebuild` is not needed, `process` returns `Nothing` to indicate that. Otherwise, it executes `act` leading to one of two possible `result`s:

- `MissingDependency k` indicates that the execution of `act` has failed, because one of its dependencies `k` was out-of-date, i.e. the calculation chain from the previous build was incorrect and therefore needs to be reordered, deferring the `key` to be rebuilt later.

834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882

- `Result v dynamicDependencies` indicates that the execution has succeeded producing the value `v` and the list of the `key`'s dynamic dependencies. We store the value, and mark it dirty to trigger the rebuilding of keys that depend on it.

In both of the above cases, we notify the parent `reordering` function of the outcome by returning `Just result`. The astute reader may notice that `process` ignores `dynamicDependencies` available in the `result`. While not required for Excel, we have implemented build systems using `reordering` which use verifying and constructive traces, effectively turning Excel into a cloud build system and ensuring `reordering` is not overly fitted to Excel alone.

In reality Excel works slightly differently (as far as we are able to ascertain) – on a change it propagates the dirty bits forward using the `dynamicDependencies`, then only checks if the current `key` is dirty. While both methods are equivalent, merely changing the interleaving, our approach allows us to model more of the behaviour of Excel.

5.3 Shake

The Shake approach for dependency tracking involves recording traces and verifying them, for which we use the `Trace` type defined in 4.2.2. Complete code is given in §8, split into three functions:

`traceMatch` takes a list of all recorded `Trace` values, the `key` you are currently building, and a function `check` which checks a specified dependency. It returns the `result` field of all traces that match. Since `check` is in an arbitrary monad, the function has to use `allM`/`&&^` instead of the more usual `all`/`&&` functions[8].

`recursive` defines the dependency ordering pattern. It requires a `process` function that builds a key given a way to recursively build a dependency, and a way to run task to produce result. The main purpose of `recursive` is to ensure that in a single run no key is built twice – for which it extends the `State` monad with a list of `done` keys.

`shake` ties everything together. First it tests if the traces allow the current state of the target key. If not, it builds the key and updates the store. The only subtlety is that Shake calls `fetch` on the keys while checking them – building the last-recorded dependencies before checking them. One minor annoyance is that the `State` has been extended and thus needs to be projected using `fst` before it is used.

5.4 Bazel

Now we have seen all three dependency schemes, we can directly reuse `topological` to define Bazel. Furthermore, as Bazel is a tracing build system, we can reuse `Trace` and `traceMatch`, along with the `Traces` type from §4.2.3. With these pieces in place, the implementation of Bazel is given in Fig. 9. We first figure out the possible results given the current state. If there are no recorded traces for the current dependencies we run it and record the results, otherwise grab a suitable result from the `contents` cache.

The program presented above captures certain aspects of Bazel, but the real implementation makes one important additional assumption on `Task` – namely that it is *deterministic*. With that assumption Bazel is able to create the result hash in a trace by hashing together the result hashes of the dependencies and the key – as the mapping between dependencies and results is deterministic. As a consequence Bazel can compute the results of traces locally, without looking at `Traces` (potentially saving a roundtrip to the server). To model this change in the code would require storing an additional transient piece of information `done` of type `Map k (Hash v)`, storing the computed hashes so far this run. With that available, `getHash key s` would become:

```
hash (key, [ done Map.! d | d <- ds ])
```

And that new hash would have to be stored in `done`.

[8]These functions are all available in the extra library on Hackage.

```
-- Determine whether a trace is relevant to the current state
traceMatch :: (Monad m, Eq k)
    => (k -> Hash v -> m Bool) -> k -> [Trace k v] -> m [Hash v]
traceMatch check key ts = mapMaybeM f ts
    where f (Trace k dkv v) = do
                b <- return (key == k) &&^ allM (uncurry check) dkv
                return $ if b then Just v else Nothing

-- Recursive dependency strategy
recursive :: Eq k => (k -> (k -> State (Store i k v, [k]) v)
                  -> State (Store i k v, [k]) (v, [k])
                  -> State (Store i k v, [k]) ())
                  -> Build Monad i k v
recursive process task key store = fst $ execState (ensure key) (store, [])
    where
        ensure key = do
            let fetch k = do ensure k; gets (getValue k . fst)
            done <- gets snd
            when (key `notElem` done) $ do
                modify $ \(s, done) -> (s, key:done)
                case track task fetch key of -- track is defined in §3.7
                    Nothing -> return ()
                    Just act -> process key fetch act

-- Shake build system
shake :: (Eq k, Hashable v) => Build Monad [Trace k v] k v
shake = recursive $ \key fetch act -> do
    traces <- gets (getInfo . fst)
    poss <- traceMatch (\k v -> (==) v . hash <$> fetch k) key traces
    current <- gets (getHash key . fst)
    when (current `notElem` poss) $ do
        (v, ds) <- act
        modify $ \(s, done) ->
            let t = Trace key [(d, getHash d s) | d <- ds] (getHash key s)
            in (putInfo (t : getInfo s) (putValue key v s), done)
```

Fig. 8. An implementation of Shake using our framework.

```
bazel :: (Eq k, Hashable v) => Build Applicative (Traces k v) k v
bazel = topological $ \key ds act -> do
    s <- get
    let Traces traces contents = getInfo s
    poss <- traceMatch (\k v -> return $ getHash k s == v) key traces
    if null poss then do
        v <- act
        modify $ \s ->
            let t = Trace key [(d, getHash d s) | d <- ds] (getHash key s)
                ts = Traces (t : traces) (Map.insert (hash v) v contents)
            in putInfo ts (putValue key v s)
    else do
        when (getHash key s `notElem` poss) $
            modify $ putValue key (contents Map.! head poss)
```

Fig. 9. An implementation of Bazel using our framework; topological is defined in Fig. 6.

```
cloudShake :: (Eq k, Hashable v) => Build Monad (Traces k v) k v
cloudShake = recursive $ \key fetch act -> do
    s <- gets fst
    let Traces traces contents = getInfo s
    poss <- traceMatch (\k v -> (==) v . hash <$> fetch k) key traces
    if null poss then do
        (v, ds) <- act
        modify $ \(s,done) ->
            let t = Trace key [(d, getHash d s) | d <- ds] (getHash key s)
                ts = Traces (t : traces) (Map.insert (hash v) v contents)
            in (putInfo ts (putValue key v s), done)
    else do
        s <- gets fst
        when (getHash key s `notElem` poss) $
            modify $ \(s, done) -> (putValue key (contents Map.! head poss) s, done)
```

Fig. 10. An implementation of Cloud Shake using our framework.

5.5 Cloud Shake

Using the abstractions and approaches built thus far, we have shown how to combine dependency scheme and change approach to reproduce existing build systems. In the attached materials we have implemented 9 build systems corresponding to all three dependency schemes, matched with all three change approaches. To us, the most interesting build system as yet unavailable would matching recursive ordering with constructive traces – providing a cloud-capable build system with minimality, cutoff and monadic dependencies. Using our framework it is possible to define and test such a system as per Fig. 10.

The differences from `bazel` are minor – the dependency scheme has changed from `topological` to `recursive`, and thus the dependency keys `dk` are captured from the action rather than in advance, the transient state has gained a list of keys, and the checking calls `fetch` to get the result instead of accessing the store directly.

5.6 Smarter `[Trace]` data structures

In the examples above, we have used `[Trace k v]` to capture a list of traces – however, using a list necessarily means that finding the right trace takes $O(n)$. For each of the `Trace` based systems it is possible to devise a smarter representation, which we sketch below. Note that these implementations do not avoid calls to `compute`, merely overheads in the build system itself.

1. Any system using verifying traces, e.g. Shake, is unlikely to see significant benefit from storing more than one `Trace` per key[9]. Therefore, such systems can store `Map k (Trace k v)`, where the initial `k` is the `key` field of `Trace`.
2. Any system using `Applicative` dependencies can omit the dependency keys from the `Trace` as they can be recovered from the `key` field.
3. Any `Applicative` build system storing constructive traces, e.g. Bazel, can index directly from the key and results to the output result – i.e. `Map (k, [Hash v]) (Hash v)`. Importantly, assuming the traces are stored on a central server, the client can compute the key and the hashes of its dependencies, then make a single call to the server to retrieve the result hash. In this formulation we have removed the possibility for a single key/dependency state to map to multiple different hashes, e.g. on a non-deterministic build – something Bazel already prohibits which is discussed more in §6.3.

[9] There is a small chance of a benefit if the dependencies change but the result does not, and then the dependencies change back to what they were before.

(4) Finally, a `Monad` build system with constructive traces can be stored as `Map k (Choice k v)`, assuming a definition of `Choice` as:

```
data Choice k v = Choice k (Map (Hash v) Choice)
                | Result (Hash v)
```

Here the `Choice` encodes a tree, asking successive questions about keys, and taking different branches based on the answers, until it reaches a final result. Implementing this structure over client-server communication requires either a chatty interface with lots of round-trips per `Choice` step, or sending over a part of the tree that is not subsequently explored.

6 ENGINEERING ASPECTS

In the previous sections we have modelled the most critical subset of various build systems. However, like all real-world systems, there are many corners that obscure the essence. In this section we discuss some of those details, what would need to be done to capture them in our model, and what the impact would be.

6.1 Partial stores and exceptions

Our model assumes a world where the store is fully-defined, every `k` is associated with a `v`, and every compute successfully completes returning a valid value. In the real world, build systems frequently deal with errors, e.g. "file not found", or "compilation failed". We can model such failure conditions by instantiating `v` to either `Maybe v` (for missing values) or `Either e v` (for exceptions of type `e`). That can model the values inside the store and the `Task`, but because `v` is polymorphic for builds, it does not let the build system apply special behaviour for errors, e.g. early aborting.

6.2 Parallelism

While we have given simple implementations assuming a single thread of execution, all the build systems we address can actually build independent keys in parallel. While it complicates the model, the complications can be restricted exclusively to the dependency strategy:

(1) The `topological` function can build the full dependency graph, and whenever all dependencies of a task are complete, the task itself can be started.
(2) The `reordering` function can be made parallel in a few ways, but the most direct is to have n threads reading entries from the list of keys. As before, if a key requires something not yet built, it is added to the end – the difference is that sometimes things will be moved to the back of the queue not because they are out of order but because of races with earlier nodes that had not yet finished. As a consequence, over successive runs, potentially racey dependencies will be separated, giving better parallelism over time.
(3) The `recursive` function can be made parallel by starting static dependencies of a `Task` in parallel, while dynamic dependencies are being resolved, using the strategy described by Marlow et al. [2014].

The actual implementation of the parallel strategies is not overly onerous, but neither is it beautiful or informative.

6.3 Impure computations

In our model we define `Task` as a function – when given the same inputs it will always produce the same outputs. Alas, the real-world is not so obliging. Some examples of impure tasks include:

Untracked state Some tasks depend on untracked state – for example C compilation will explicitly list the source.c file as a dependency, but it may not record that the version of gcc is also a dependency.

Non-determinism Some tasks are *non-deterministic*, producing a result from a possible set. As an example, GHC when compiled using parallelism can change the order in which unique variables are obtained from the supply, producing different but semantically identical results.

Volatility Some tasks are defined to change in every execution, for example Excel provides a "function" RANDBETWEEN which produces a random number in a specified range – on each recalculation it is defined to change. Similarly, build systems like Make and Shake provide *phony rules* which are also volatile.

Interestingly, there is significant interplay between all three sources of impurity. Systems like Bazel use various sandboxing techniques to guard against missing dependencies, but none are likely to capture all dependencies right down to CPU model and microcode version. Rules that do have untracked state can be marked as volatile, a technique Excel takes with the INDIRECT function, removing the untracked state at the cost of minimality.

Most of the implementations in §5 can deal with non-determinism, apart from Bazel, which requires deterministic execution, and in turn can optimise the number of roundtrips required to the server.

One tempting way of modelling non-determinism is to enrich `Task` from `Applicative` or `Monad` to `Alternative` or `MonadPlus`, respectively. More concretely, the following task description corresponds to a spreadsheet with the formula B1 = A1 + RANDBETWEEN(1,2):

```
sprsh3 :: Task MonadPlus String Integer
sprsh3 fetch "B1" = Just $ (+) <$> fetch "A1" <*> pure 1 `mplus` pure 2
sprsh3 _     _    = Nothing
```

Handling such tasks is possible in our framework, but requires an adjustment of the correctness definition (§3.6): instead of requiring that the result of recomputing the `task` matches the value stored in the `result`:

```
Just (getValue k result) == compute task result k
```

we now require that `result` contains *one possible result of recomputing* the `task`:

```
Just (getValue k result) `elem` computeND task result k
```

where `computeND :: Task MonadPlus k v -> Store i k v -> k -> Maybe [v]` returns the list of all possible results of the `task` instead of just one value ('ND' stands for 'non-deterministic').

Note that `Task MonadPlus` is powerful enough to model dependency-level non-determinism, for example, INDIRECT("A" & RANDBETWEEN(1,2)), whereas most build tasks in real-life build systems only experience a value-level non-determinism. Excel handles this example simply by marking the cell volatile – an approach that can be readily adopted by any of our implementations by introducing a special key RealWorld whose value is changed between every run.

6.4 Cloud implementations

Our model of cloud builds provides a basic framework to discuss and reason about them, but lacks a number of important engineering corners:

Eviction The store of traces as shown grows indefinitely, but often resource constraints require evicting old items from the store. One option is to evict the contents and any trace that mentions the now-defunct `Hash v`. However, if the build system can defer materialisation, it may be possible to only evict the contents, allowing builds to pass-through hashes of values where the underlying value is not known. If so, the build must be able to recreate the value if required, potentially dealing with a different result in a future run.

Frankenbuilds A build is considered a *frankenbuild* [Esfahani et al. 2016] if a value is calculated locally, but something that depends on that key is pulled from the cache, and the value calculated locally does not match what was previously calculated and stored in the cloud. Our implementations avoid this issue by storing complete traces, but if a cloud build system was to only reference input nodes this situation can arise.

Communication When traces or contents are stored on a central server communication can become a bottleneck, so it is important to send only the minimum amount of information, optimising with respect to build-system specific invariants – see §5.6 for some possible optimisations.

Offloading Once the cloud is storing build products and traces, it is possible for the cloud to also contain dedicated workers that can execute tasks remotely – offloading some of the computation and potentially running vastly more commands in parallel.

Shallow builds Sometimes input files will involve many intermediate tasks before producing the end result, e.g. an installer package. These intermediate steps may be large, so some cloud build systems are designed to build end products *without* downloading or *materialising* the results of intermediate tasks – only the final result – a so-called *shallow build*. Some build systems can go even further, integrating with the file system to only materialise the file when the user accesses it [Microsoft 2017].

To legitimise shallow builds, we need to relax the correctness Definition 3.1 as follows. Let the `shallow` store correspond to the result of a shallow build. Then `shallow` is correct, if *there exists* `result` which satisfies all requirements of Definition 3.1, *such that* `shallow` agrees with `result` on the input keys $k \in I$:

$$\texttt{getValue k shallow} == \texttt{getValue k result}$$

and on the target `key`:

$$\texttt{getValue key shallow} == \texttt{getValue key result}.$$

This relaxes the requirements on shallow builds by dropping the constraints on the `shallow` store for all intermediate keys $k \in O \setminus \{key\}$.

6.5 Tracking and self-tracking

Some build systems, for example Excel and Ninja, are capable of recomputing a task if either its dependencies change, *or* the rule itself changes. For example:

```
A1 = 20      B1 = A1 + A2
A2 = 10
```

In Excel the user can alter the value produced by B1 by either editing the inputs of A1 or A2, *or* editing the formula in B1 – e.g. to A1 - A2. This pattern can be captured by describing the rule producing B1 as also depending on the value B1-formula. The implementation can be given very directly in a **`Task Monad`** as:

```
task fetch "B1" = do
    formula <- fetch "B1-formula"
    evalFormula fetch formula
```

Namely, first look up the formula, then interpret it. It is not possible to change dependencies based on the formula in a **`Task Applicative`**, as would be required, so instead the formula can be captured as a dependency (but its value not used) and *also* baked directly into the `task` function.

The build systems that have precise self-tracking are all ones which use a *non-embedded domain specific language* to describe computations. Those which make use of a full programming language,

e.g. Shake, are faced with the challenge of implementing equality on arbitrary task functions. For such build systems the incredibly pessimistic assumption of saying that any change to the build system potentially changes any build rule can often be used – the classic example being a makefile depending on itself.

6.6 Iterative computations

Some computations are best described not by a chain of acyclic dependencies, but by a loop – for example LaTeX requires repeated rebuilding until it reaches a fixed point – something that can be directly expressed in build systems, such as Pluto [Erdweg et al. 2015]. Another example of cyclic computations is Excel, where a cell can depend on itself, for example:

```
A1 = A1 + 1
```

In such cases Excel will normally not execute anything, but if the "Iterative Calculations" feature is enabled will execute the formula for a specified maximum number N of times per calculation (where N is a setting that defaults to 100).

For examples like LaTeX we consider the proper encoding to not be with circular tasks, but with a series of iterative steps, as described by Mitchell [2013]. It is important that the number of executions is bounded, otherwise the build system may not terminate (a legitimate concern with LaTeX, which can be put into a situation where it is bistable or diverging over multiple executions).

The examples in Excel tend to encode either mutable state, or recurrence relations. The former is only required because Excel inherently lacks the ability to write mutable state, and the latter is probably better solved using explicit recurrence formulae.

Overall we choose not to deal with cyclic rules, a choice that most build systems also follow.

6.7 Polymorphism

Our build system abstraction assumes a k/v store, along with a build system that works directly on k and v values. However, certain build systems provide greater flexibility, e.g. Shake permits polymorphic keys and values, allowing types that are only stored in the Shake information, and never persisted to the store.

As one example of richer key/value types, consider the version of gcc – for many builds it should be a dependency. In Shake it is possible to define an *oracle* rule as per [Mitchell 2012] whose value is the result of running gcc –version and which is volatile, making the gcc version something that can be depended upon. Of course, provided the build can express volatile dependencies and supports cutoff, the version number could equally be written to a file and used in a similar way.

A more compelling example is build tasks that produce multiple output keys – for example, ghc Foo.hs produces both Foo.hi and Foo.o. That can be represented by having a key whose value is a pair of file names, and whose result is a pair of file contents. From that, the rule for Foo.hi can be the first component of the result of the pair. Again, such an operation can be encoded without polymorphic keys provided the pair of files (or a dummy file representing the pair) is marked as changed if either of the contained files change. Once again, polymorphic dependencies provide convenience rather than power.

Shake users have remarked that polymorphism provides a much easier expression of concepts, e.g. [Mokhov et al. 2016], but it is not essential and thus not necessary to model.

7 RELATED WORK

While there is research on individual build systems, there has been little research to date comparing different build systems. In §2 we covered several important build systems – in this section we relate a few other build systems to our abstractions, and discuss other work where similar abstractions arise.

7.1 Other Build Systems

Most build systems, when viewed at the level we talk, can be captured with minor variations on the code presented in §5. As some examples:

- NINJA [Martin 2017] combines the dependency strategy of MAKE with the validating traces of SHAKE – our associated implementation provides such a combination. Ninja is also capable of modelling rules that produce multiple results, a limited form of polymorphism §6.7.
- TUP [Shal 2009] functions much like MAKE, but with a refined dirty-bit implementation that watches the file system for changes and can thus avoid rechecking the entire graph, and with automatic deleting of stale results.
- REDO [Pennarun 2012] almost exactly matches SHAKE at the level of detail given here, differing only on aspects like polymorphic dependencies §6.7.
- BUCK [Facebook 2013] is very similar to BAZEL at the level of abstraction presented here.
- CLOUDBUILD [Esfahani et al. 2016] differs from BAZEL by allowing non-determinism §6.3, thus more closely modelling our original definition of BAZEL from §5.4.
- NIX [Dolstra et al. 2004] has coarse-grained dependencies, with precise hashing of dependencies and downloading of precomputed build products. When combined with import-from-derivation, NIX can also be considered monadic, making it similar to Cloud SHAKE from §5.5. However, NIX is not intended as a build system, and the coarse grained nature (packages, not individual files) makes it targeted to a different purpose.
- PLUTO [Erdweg et al. 2015] is based on a similar model to SHAKE, but additionally allows cyclic build rules combined with a user-specific resolution strategy. Often such a strategy can be unfolded into the user rules without loss of precision, but a fully general resolution handler extends the **Task** abstraction with additional features.

The one build system we are aware of that cannot be modelled in our framework is FABRICATE [Hoyt et al. 2009]. In FABRICATE a build system is a script which is run in-order, in the spirit of[10]:

```
gcc -c util.c
gcc -c main.c
gcc util.o main.o -o main.exe
```

To achieve minimality, each separate command is traced at the OS-level, allowing FABRICATE to record a trace entry stating that gcc -c util.c reads from util.c. In future runs FABRICATE runs the script from start to finish, skipping any commands where no inputs have changed.

Taking our abstraction, it is possible to encode FABRICATE assuming that commands like gcc -c util.c are keys, there is a linear dependency between each successive node, and that the OS-level tracing can be lifted back as a monadic **Task** function[11]. However, in our pure model the mapping is not perfect as gcc writes to arbitrary files whose locations are not known in advance.

7.2 Self-adjusting computation

While not typically considered build systems, self-adjusting computation is a well studied area, and in particular the contrast between different formulations has been thoroughly investigated [Acar et al. 2007].

Self-adjusting computations can automatically adjust to an external change to their inputs. A classic example is a self-adjusting sorting algorithm, which can efficiently (in $O(\log n)$ time

[10] FABRICATE requires scripts to be written in Python, but those details are not fundamental to what makes FABRICATE special.

[11] In fact, SHAKE has an execution mode that can model FABRICATE-like build systems – see **Development.Shake.Forward** in the SHAKE library.

where n is the length of the input) recalculate the result given an incremental change of the input. While very close to build systems in spirit, self-adjusting computations are mostly used for in-memory computation and rely on the ability to dynamically allocate new keys in the store for sharing intermediate computations – an intriguing feature rarely seen in build systems (Shake's oracles §6.7 can be used to model this feature to a limited degree).

A lot of research has been dedicated to finding efficient data structures and algorithms for self-adjusting computations – we plan to investigate how these insights can be utilised by build systems as future work.

7.3 Memoization

Memoization is a classic optimisation technique for storing values of a function instead of recomputing them each time the function is called. Minimal build systems (see the Definition 2.1) certainly perform memoization: they *store values instead of recomputing them each time*. Memoization can therefore be reduced to a minimal build system (as we demonstrate below), but not vice versa, since minimal build systems solve a more complex optimisation problem.

As a simple example of using a build system for memoization, we solve a textbook dynamic programming problem – Levenshtein's *edit distance* [Levenshtein 1966]: given two input strings a and b, find the shortest series of edit operations that transforms a to b. The edit operations are typically *inserting*, *deleting* or *replacing* a symbol. The dynamic programming solution of this problem is so widely known (e.g., see [Cormen et al. 2001]) that we provide its encoding in our `Task` abstraction without further explanation. We address elements of strings a_i and b_i by keys `A` i and `B` i, respectively, while the cost of a subproblem d_{ij} is identified by `D` i j.

```
data Key = A Integer | B Integer | D Integer Integer deriving Ord

editDistance :: Task Monad Key Integer
editDistance _     (D i 0) = Just $ pure i
editDistance _     (D 0 j) = Just $ pure j
editDistance fetch (D i j) = Just $ do
    ai <- fetch (A i)
    bj <- fetch (B j)
    if ai == bj
        then fetch (D (i - 1) (j - 1))
        else do
            insert  <- fetch (D  i      (j - 1))
            delete  <- fetch (D (i - 1)  j     )
            replace <- fetch (D (i - 1) (j - 1))
            return (1 + minimum [insert, delete, replace])
editDistance _ _ = Nothing
```

When asked to build `D` n m, a minimal build system will calculate the result using memoization. Furthermore, when an input symbol a_i is changed, only necessary, incremental recomputation will be performed – an optimisation that cannot be achieved just with memoization.

Self-adjusting computation, memoization and build systems are inherently related topics, which poses the question of whether there is an underlying common abstraction waiting to be discovered.

7.4 Profunctor Optics

The definition of `Task` is:

```
type Task c k v = forall f. c f => (k -> f v) -> k -> Maybe (f v)
```

Which looks tantalisingly close to the profunctor optics definition by [Pickering et al. 2017]:

```
type Optic p a b s t = p a b -> p s t
```

Provided we instantiate `p` to something like `k -> f v` – which many of the actual instances in that paper do. The properties of such optics are well studied, and the functions like `dependencies` are very much based on observations from that field of work. Alas, we have been unable to remove the `Maybe` used to encode whether a file is an input, without complicating other aspects of our definition. Furthermore, the `Build` abstraction lacks any further such symmetry.

8 CONCLUSIONS

We have investigated multiple build systems, showing how their properties are consequences of two implementation choices: what order you build in and whether you decide to rebuild. By first decomposing the pieces, we show how to recompose the pieces to find new points in the design space. In particular, a simple recombination leads to a design for a monadic cloud build system. Armed with that blueprint we hope to actually implement such a system as future work.

ACKNOWLEDGMENTS

Thank you for reading and feedback!

Andrey Mokhov is funded by a Royal Society Industry Fellowship on the topic "Towards Cloud Build Systems with Dynamic Dependency Graphs".

REFERENCES

Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2002. Adaptive Functional Programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 247–259.

Umut A Acar, Matthias Blume, and Jacob Donham. 2007. A consistent semantics of self-adjusting computation. In *European Symposium on Programming*. Springer, 458–474.

Paolo Capriotti and Ambrus Kaposi. 2014. Free applicative functors. *Proceedings 5th Workshop on Mathematically Structured Functional Programming* 153, 2–30.

T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. 2001. *Introduction To Algorithms*. MIT Press.

R. De Levie. 2004. *Advanced Excel for Scientific Data Analysis*. Oxford University Press.

Alan Demers, Thomas Reps, and Tim Teitelbaum. 1981. Incremental Evaluation for Attribute Grammars with Application to Syntax-directed Editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 105–116.

Eelco Dolstra, Merijn De Jonge, Eelco Visser, et al. 2004. Nix: A Safe and Policy-Free System for Software Deployment. In *LISA*, Vol. 4. 79–92.

Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A sound and optimal incremental build system with dynamic dependencies. *ACM SIGPLAN Notices* 50, 10 (2015), 89–106.

Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 11–20.

Facebook. 2013. Buck: A high-performance build tool. (2013). https://buckbuild.com/.

Stuart I Feldman. 1979. Make—A program for maintaining computer programs. *Software: Practice and experience* 9, 4 (1979), 255–265.

Google. 2016. Bazel. (2016). http://bazel.io/.

Berwyn Hoyt, Bryan Hoyt, and Ben Hoyt. 2009. Fabricate: The better build tool. (2009). https://github.com/SimonAlfie/fabricate.

Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 333–343.

Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 325–337.

Evan Martin. 2017. Ninja build system homepage. (2017). https://ninja-build.org/.

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of functional programming* 18, 1 (2008), 1–13.

Microsoft. 2011. Excel Recalculation (MSDN documentation). (2011). https://msdn.microsoft.com/en-us/library/office/bb687891.aspx. Also available in Internet Archive https://web.archive.org/web/20180308150857/https://msdn.microsoft.com/en-us/library/office/bb687891.aspx.

Microsoft. 2017. Git Virtual File System. (2017). https://www.gvfs.io/.

Neil Mitchell. 2012. Shake before building: Replacing Make with Haskell. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 55–66.

Neil Mitchell. 2013. How to write fixed point build rules in Shake. (2013). https://stackoverflow.com/questions/14622169/how-to-write-fixed-point-build-rules-in-shake-e-g-latex.

Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. 2016. Non-recursive Make Considered Harmful: Build Systems at Scale. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, 170–181.

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

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of functional programming* 17, 1 (2007), 1–82.

Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. 2017. Profunctor Optics: Modular Data Accessors. *The Art, Science, and Engineering of Programming* 1 (2017). Issue 2.

Mike Shal. 2009. Build System Rules and Algorithms. (2009). http://gittup.org/tup/build_system_rules_and_algorithms.pdf/.