Build Systems à la Carte

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 are used by every developer 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 new build systems with desired properties.

CCS Concepts: • **Software and its engineering**; • **Mathematics of computing**;

Additional Key Words and Phrases: build systems, functional programming, algorithms

ACM Reference Format:
Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build Systems à la Carte. *Proc. ACM Program. Lang.* 2, ICFP, Article 79 (September 2018), 29 pages. https://doi.org/10.1145/3236774

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. For years Make dominated, but more 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. These complex build systems use subtle algorithms, but they are often hidden away, and not the object of study.

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 tasks; support for early cutoff; self-tracking build systems; and the type of persistent build information. In §2 we identify some key properties, illustrated by 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 tasks are built* (§4.1) and *whether or not a task is (re-)built* (§4.2). These choices turn out to be orthogonal, which leads us to a new classification of the design space (§4.4).

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.

2475-1421/2018/9-ART79
https://doi.org/10.1145/3236774

- 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, CLOUDBUILD, BUCK, NIX, and EXCEL[1], each by the composition of the two design choices (§5). Doing this modelling in a single setting allows the differences and similarities between these huge systems to be brought out clearly[2].
- Moreover, we can readily remix the ingredients to design new build systems with desired properties, for example, to combine the advantages of SHAKE and BAZEL (§5.4).

In short, instead of seeing build systems as unrelated points in space, we now see them as locations in a connected landscape, leading to a better understanding of what they do and how they compare, and suggesting exploration of other (as yet unoccupied points) in the landscape. We discuss engineering aspects in §6, and related work in §7.

2 BACKGROUND

Build systems automate the execution of repeatable tasks for individual users and 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]. 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 §5 and §7.

2.1 The Venerable MAKE: Static Dependencies and File Modification Times

MAKE[3] 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[4] 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).

If the user runs MAKE specifying main.exe as the desired output, MAKE will build util.o and main.o, in any order since these tasks are independent, and then main.exe. If the user modifies util.h and runs MAKE again, it will perform a *full rebuild*, because all three 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: 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 but there are rare exceptions, see §6.6.

[1]EXCEL appears very different to the others but, seen through the lens of this paper, it is very close indeed.
[2]All our models are executable and are available on Hackage: https://hackage.haskell.org/package/build-1.0.
[3]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.
[4]In this example we pretend gcc is a pure function for the sake of simplicity. In reality, there are multiple versions of gcc; the actual binary used to compile and link files, and standard libraries such that stdio.h, are often also listed as dependencies.

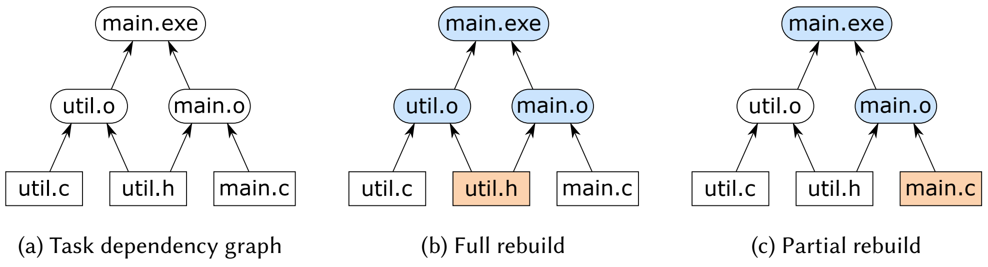


Fig. 1. A task dependency graph and two build scenarios. Input files are shown in rectangles, intermediate and output files are shown in rounded rectangles. Modified inputs and files that are rebuilt are highlighted.

The following property is essential for build systems; indeed, 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 times* to detect which files changed[5], 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, producing the result in cell B1. If either of the inputs change, EXCEL will recompute the result.

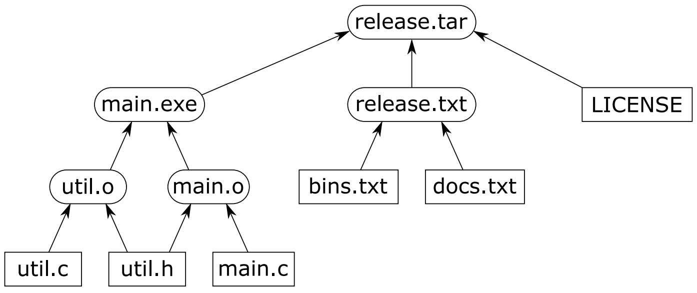
Unlike MAKE, EXCEL does not need to know all task dependencies upfront. Indeed, some dependencies may change *dynamically* according to computation results. For example:

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

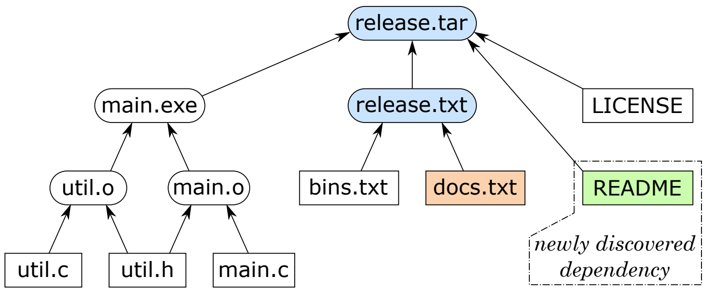
The formula in B1 uses the INDIRECT function, which takes a string and returns the value of the cell with that name. The string evaluates to "A1", so B1 evaluates to 10. However, the dependencies of the formula in B1 are determined by the value of C1, so it is impossible to compute the dependency graph before the build starts[6].

To support dynamic dependencies, EXCEL's calc engine [Microsoft 2011] is significantly different from MAKE. EXCEL arranges the cells into a linear sequence, called the *calc chain*. During the build, EXCEL processes cells in the calc-chain sequence, but if computing a cell C requires the value of a cell D that has not yet been computed, EXCEL *aborts* computation of C, moves D before C in the calc chain, and resumes the build starting with D. When a build is complete, the resulting calc chain respects all the dynamic dependencies of the spreadsheet. When an input value (or formula) is changed, EXCEL uses the final calc chain from the *previous* build as its starting point so that, in the common case where changing an input value does not change dependencies, there are no aborts. Notice that build always succeeds regardless of the initial calc chain (barring truly circular

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



(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.

dependencies); the calc chain is just an optimisation. We refer to this algorithm as *restarting*, and discuss it in more detail in §5.2.

Dynamic dependencies complicate minimality. In the above example, B1 should only be recomputed if A1 or C1 change, but not if (say) A2 changes; but these facts are not statically apparent. In practice Excel implements a conservative approximation to minimality: it recomputes a formula if (i) the formula statically mentions a changed cell, or (ii) the formula uses a function like INDIRECT whose dependencies are not statically visible, or (iii) the formula itself has changed.

Item (iii) in the above list highlights another distinguishing feature of Excel: it is *self-tracking*. Most build systems only track changes of inputs and intermediate results, but Excel also tracks 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 task has changed. We discuss self-tracking further in §6.5.

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, which 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,

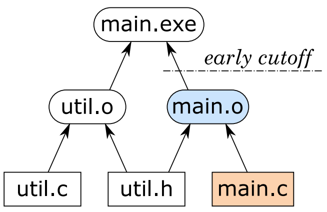


Fig. 3. An early cutoff example: if a comment is added to main.c, the rebuild is stopped after detecting that main.o is unchanged, since this indicates that main.exe and its dependents do not need to be rebuilt.

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", "release.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 aborting and deferring the execution of tasks whose newly discovered dependencies have not yet been built (as Excel does), Shake *suspends* their execution until the dependencies are brought up to date. We refer to this task scheduling algorithm as *suspending*, see a concrete implementation in §5.3.

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.

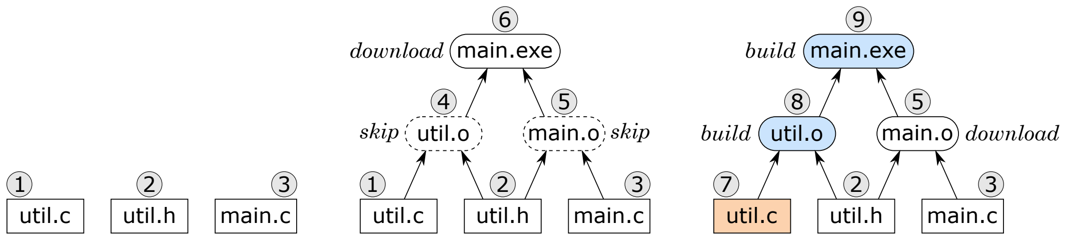
2.4 Bazel: A Cloud Build System

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 can support *shallow builds* that materialise only end build products locally, leaving all intermediates in the cloud.

Consider an example in Fig. 4. The user starts by downloading the sources, whose content 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[7], the build system finds that someone has already compiled these exact sources before and the resulting files util.o and main.o had hashes 4 and 5, respectively. Similarly, the build system finds that the hash of the resulting main.exe was 6 and downloads the actual binary from the cloud storage – it must be materialised, because it is the end build product.

In Fig. 4(c), the user modifies the source file util.c, thereby changing its hash from 1 to 7. The cloud lookup of the new combination {util.c, util.h} fails, which means that nobody has ever compiled it. The build system must therefore build util.o, materialising it with the new hash 8. The combination

[7]In practice, old entries are regularly evicted from the cloud storage, as further discussed in §6.4.



(a) Download source files (b) Build main.exe (c) Modify util.c and rebuild

Fig. 4. A cloud build example: (a) download sources, (b) build main.exe by downloading it from the cloud and skipping intermediate files (only their hashes are needed), (c) modify util.c and rebuild main.exe, which requires building util.o (nobody has compiled util.c before) and downloading main.o (it is needed for linking main.exe). File hashes are shown in circles, and non-materialised intermediates in dashed rounded rectangles.

of hashes of util.o and main.o has not been encountered before either, thus 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.

Bazel is one of the first openly-available cloud build systems. As of writing, it is not possible to express dynamic dependencies in user-defined build rules; however some of the pre-defined build rules require dynamic dependencies and the internal build engine can cope with them by using a *restarting* task scheduler, which is similar to that of Excel but does not use the calc chain. Bazel is not minimal in the sense that it may restart a task multiple times as new dependencies are discovered and rebuilt, but it supports the early cutoff optimisation.

To support cloud builds, Bazel maintains (i) a *content-addressable cache* that can be used to download a previously built file given the hash of its content, and (ii) the history of all executed build commands annotated with observed file hashes. The latter allows the build engine to bypass the execution of a task, by predicting the hash of the result from the hashes of its dependencies, and subsequently download the result from the cache. A concrete implementation is provided in §5.4.

Table 1. Build system differences.

Build system	Persistent build information	Scheduler	Dependencies	Minimal	Cutoff	Cloud
Make	File modification times	Topological	Static	Yes	No	No
Excel	Dirty cells, calc chain	Restarting	Dynamic	No	No	No
Shake	Previous dependency graph	Suspending	Dynamic	Yes	Yes	No
Bazel	Cloud cache, command history	Restarting	Dynamic$^{(*)}$	No	Yes	Yes

$^{(*)}$At present, user-defined build rules cannot have dynamic dependencies.

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 calc 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 the content-addressable cache and the history of all previous build commands annotated with file hashes. This information is shared among all users of the build system.

In this paper we elucidate which build system properties are consequences of specific implementation choices (stored metadata and task scheduling algorithm), and how one can obtain new build systems with desired properties by recombining parts of existing implementations. As a compelling example, in §5.4 we demonstrate how to combine the advantages of Shake and Bazel.

```haskell
-- An abstract store containing a key/value map and persistent build information
data Store i k v -- i = info, k = key, v = value
initialise :: i -> (k -> v) -> 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
data Hash v -- a compact summary of a value with a fast equality check
hash    :: Hashable v => v -> Hash v
getHash :: Hashable v => k -> Store i k v -> Hash v
-- Build tasks (see §3.2)
newtype Task  c k v = Task { run :: forall f. c f => (k -> f v) -> f v }
type    Tasks c k v = k -> Maybe (Task c k v)
-- Build system (see §3.3)
type Build c i k v = Tasks c k v -> k -> Store i k v -> Store i k v
-- Build system components: a scheduler and a rebuilder (see §5)
type Scheduler c i ir k v = Rebuilder c ir k v -> Build c i k v
type Rebuilder c   ir k v = k -> v -> Task c k v -> Task (MonadState ir) k v
```

Fig. 5. Type signatures of key build systems abstractions.

3 BUILD SYSTEMS, ABSTRACTLY

This section presents purely functional abstractions that allow us to express all the intricacies of build systems discussed 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

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 A1) 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.

We will model build systems concretely, as Haskell programs. To that end, Fig. 5 provides the type signatures for all key abstractions introduced in the paper. 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. Fig. 6 lists standard library definitions.

```haskell
-- Applicative functors
pure  :: Applicative f => a -> f a
(<$>) :: Functor     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 Monad (State s)
get       :: State s s
gets      :: (s -> a) -> State s a
put       :: s -> State s ()
modify    :: (s -> s) -> State s ()
runState  :: State s a -> s -> (a, 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   -- mempty is the identity of the monoid m
    Const x <*> Const y = Const (x <> y) -- <> is the binary operation of the monoid m
```

Fig. 6. Standard library definitions.

3.2 The Task Abstraction

Our first main abstraction is for *task descriptions*:

```haskell
newtype Task  c k v = Task { run :: forall f. c f => (k -> f v) -> f v }
type    Tasks c k v = k -> Maybe (Task c k v)
```

Here `c` stands for *constraint*, such as `Applicative` (§3.4 explains why we need it). A `Task` describes a single build task, while `Tasks` associates a `Task` to every non-input key; input keys are associated with `Nothing`. The highly-abstracted type `Task` describes how to build a value when given a way to build its dependencies, and is best explained 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:

```haskell
sprsh1 :: Tasks Applicative String Integer
sprsh1 "B1" = Just $ Task $ \fetch -> ((+)  <$> fetch "A1" <*> fetch "A2")
sprsh1 "B2" = Just $ Task $ \fetch -> ((*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. 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 that the key is an input. If there is a formula in the cell, it returns the `Task` to compute the value of the formula. Every `Task` is given a *callback* `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. 6. We will explain why in §3.3.

For now, we content ourselves with observing that a task description, of type **Tasks** c k v, is completely isolated from the world of compilers, calc 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.

3.3 The Build Abstraction

Next comes our second main abstraction – a build system:

```
type Build c i k v = Tasks 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 Applicative () k v
busy tasks key store = execState (fetch key) store
  where
    fetch :: k -> State (Store () k v) v
    fetch k = case tasks k of
        Nothing   -> gets (getValue k)
        Just task -> do v <- run task fetch; 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. 6 – initialised with the incoming store by execState. To bring a key up to date, fetch asks the task description tasks how to compute the value of k. If tasks returns **Nothing** the key is an input, so fetch simply reads the result from the store. Otherwise fetch runs the obtained 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 "B1" result
30
λ> getValue "B2" result
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:

```
newtype Task c k v = Task { run :: forall f. c f => (k -> f v) -> 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 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 allows *dynamic* dependencies, in which a formula may depend on 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 linear chain, as e.g. in Docker containers [Hykes 2013]. It is interesting to note that, when run on such a task description, 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.
- `Task (MonadState i)` will be used in §5 to describe tasks that have read and write access to the persistently stored build information `i`.

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 `Tasks 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:

```
sprsh2 :: Tasks Monad String Integer
sprsh2 "B1" = Just $ Task $ \fetch -> do
    c1 <- fetch "C1"
    if c1 == 1 then fetch "B2" else fetch "A2"
sprsh2 "B2" = Just $ Task $ \fetch -> 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 -> v
compute task store = runIdentity $ run 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. 6). 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 has type **Identity** v, which we unwrap with runIdentity.

Definition 3.1 (Correctness). Suppose build is a build system, tasks 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 tasks, key and store. Or, using the precise language of our abstractions:

```
build          :: Build c i k v
tasks          :: Tasks c k v
key            :: k
store, result  :: Store i k v
result = build tasks 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 $I = \{\texttt{key}\}$ and $O = \emptyset$.

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

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

  getValue k result == getValue k store.

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

  getValue k result == compute task result.

A build system is *correct* if it produces a correct result for any given tasks, 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 tasks. 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]
dependencies task = getConst $ run task (\k -> Const [k])
```

Here `Const` is a standard Haskell type defined in Fig. 6. 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. 6) 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 what `dependencies` does:

```
λ> dependencies $ fromJust $ sprsh1 "B1"
["A1","A2"]
λ> dependencies $ fromJust $ sprsh1 "B2"
["B1"]
```

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 extremely cool. We will see a practical use for `dependencies` when implementing applicative build systems, see §5.1.

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 we cannot apply the function `dependencies` to a `Task Monad` because the `Const` functor has no `Monad` instance. 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 (key/value pairs) in an arbitrary monadic context `m`:

```
import Control.Monad.Writer

track :: Monad m => Task Monad k v -> (k -> m v) -> m (v, [(k, v)])
track task fetch = runWriterT $ run task trackingFetch
  where
    trackingFetch :: k -> WriterT [(k, v)] m v
    trackingFetch k = do v <- lift (fetch k); tell [(k, v)]; return v
```

This implementation uses the standard Haskell `WriterT` *monad transformer* [Liang et al. 1995], which allows us to record additional information – a list of key/value pairs `[(k, v)]` – 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/value pair. The `task` returns a value of type `WriterT [(k, v)] m v`, which we unwrap with `runWriterT`. We will use `track` when implementing monadic build systems with dynamic dependencies, see §5.3. Here we show an example of `track`ing monadic tasks when `m = IO`.

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

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

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 **`Tasks`** `c k v`. That is, we are going to take **`Tasks`** 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 emboldened two different aspects above: the part of the build system responsible for scheduling tasks in the dependency order (a 'scheduler') can be cleanly separated from the part responsible for deciding whether a key needs to be rebuilt (a 'rebuilder'). We tackle each aspect separately in subsections §4.1 and §4.2.

4.1 The Scheduler: Respecting the Dependency Order

Section §2 introduced three different *task schedulers* that decide which tasks to execute and in what order; see the "Scheduler" column of Table 1 in §2.5. This subsection explores the properties of the three schedulers, and possible implementations.

4.1.1 Topological. The topological scheduler pre-computes a linear order of tasks, which when followed, ensures the build result is correct regardless of the initial store. Given a task description and the output `key`, you can compute the linear order by first finding the (acyclic) graph of the `key`'s reachable dependencies, 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 Restarting. To handle dynamic dependencies we can use the following approach: build tasks in an arbitrary initial order, discovering their dependencies on the fly; whenever a task calls `fetch` on an out-of-date key `dep`, abort the task, and switch to building the dependency `dep`; eventually the previously aborted task is restarted and makes further progress thanks to `dep` now being up to date. 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, and then abort.

To reduce the number of aborts (often to zero) Excel records the discovered task order in its *calc chain*, and uses it as the starting point for the next build (§2.2). Bazel's restarting scheduler does not store the discovered order between build runs; instead, it stores the most recent task dependency information. Since this information may become outdated, Bazel may also need to abort a task if a newly discovered dependency is out of date.

4.1.3 Suspending. An alternative approach, utilised by the busy build system (§3.3) and Shake, is to simply build dependencies when they are requested, suspending the currently running task. By combining that with tracking the keys that have already been built, one can obtain a minimal build system with dynamic dependencies.

This approach requires that a task may be started and then suspended until another task is complete. Suspending can be done with cheap green threads and blocking (the original approach of Shake) or using continuation-passing style [Claessen 1999] (what Shake currently does).

4.2 The Rebuilder: Determining Out-of-date Keys

Suppose the scheduler decides that a key should be brought up to date. The next question is: does any work need to be done, or is the key already up to date? Or, in a cloud-build system, do we have a cached copy of the value we need? These questions can be addressed in one of four 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 builds is marked dirty. If a key and all its transitive dependencies are clean, the key does not need to be rebuilt.

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. It also marks dirty all cells that (transitively) depend on the modified cell. Excel does not record dynamic dependencies of each cell; instead it computes a *static over-approximation* – it is safe for it to make more cells dirty than necessary, but not vice versa. The over-approximation is as follows: a cell is marked dirty if its formula statically refers to a dirty cell, or if the formula calls a function like INDIRECT whose dependencies cannot be guessed from the formula alone. The over-approximation is clear for INDIRECT, but it is also present for IF, where both branches are followed even though dynamically only one is used.

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 older 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. Modifying a file automatically clears its dirty bit, and automatically sets the dirty bit of the files depending on it (but not recursively). Note that Make requires that file timestamps only go forward in time, which can be violated by backup software.

With a dirty bit it is possible to achieve minimality. However, to achieve early cutoff (§2.3) it would be important to reset the dirty bit after a computation that did not change the value and make sure that cells that depend on it are not rebuilt unnecessarily. For Excel, this is difficult because the dependent cells have already been recursively marked dirty. For Make, it is impossible to mark a file clean and at the same time not mark the keys that depend on it dirty. Make can approximate early cutoff by not modifying the result file, and not marking it clean, but then it will be rebuilt in every subsequent build.

4.2.2 Verifying Traces. An alternative way to determine if a key is dirty is to record the values/hashes of dependencies used last time, and if something has changed, the key is dirty and must be rebuilt – in essence a *trace* which we can use to *verify* existing values. For traces, there are two essential operations – adding a new value to the trace store, and using the traces to determine if a key needs rebuilding. Assuming a store of verifying traces VT k v, the operations are:

```
recordVT :: k -> Hash v -> [(k, Hash v)] -> VT k v -> VT k v
verifyVT :: (Monad m, Eq k, Eq v) => k -> Hash v -> (k -> m (Hash v)) -> VT k v -> m Bool
```

Rather than storing (large) values v, the verifying trace VT stores only hashes, of type Hash v, of those values. Since the verifying trace persists from one build to the next – it constitutes the build system's "memory" – it is helpful for it to be of modest size. After successfully building a key, we call `recordVT` to add a record to the current `VT`, passing the key, the hash of its value, and the list of hashes and dependencies.

More interestingly, to *verify* whether a key needs rebuilding we supply the key, the hash of its current value, a function for obtaining the post-build value of any key (using a scheduling strategy as per §4.1), and the existing `VT` information. The result will be a `Bool` where `True` indicates that the current value is already up to date, and `False` indicates that it should be rebuilt.

One potential implementation would be to record all arguments passed to `recordVT` in a list, and verify by simply checking if any list item matches the information passed by `verifyVT`. We discuss smarter implementations in §4.3.

A verifying trace, and other types of traces discussed in this section, support dynamic dependencies and minimality; furthermore, all traces except for deep traces (§4.2.4) support early cutoff.

4.2.3 Constructive Traces. A verifying trace deliberately records only small hashes, so that it can be small. A *constructive* trace additionally stores the resulting value. 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 this additional information by providing the operations:

```
recordCT    :: k -> v -> [(k, Hash v)] -> CT k v -> CT k v
constructCT :: (Monad m, Eq k, Eq v) => k -> (k -> m (Hash v)) -> CT k v -> m [v]
```

The function `recordCT` looks like `recordVT`, but instead of just passing the hash of the resulting value, we require the actual value. The `verifyVT` has been replaced with `constructCT`, which instead of taking the hash of the current value as *input*, returns a list of suitable values as *output*. If the current value in the store matches one of the possible values, the build can skip this key. If the resulting list is empty, the key must be rebuilt. However, if the current value does not match the store, and there is a possible value, we can use any value from the constructive list *without* doing any work to build it, and copy it into the store.

4.2.4 Deep Constructive Traces. Constructive traces always verify keys by looking at their immediate dependencies, which must have first been brought up to date, meaning that the time to verify a key depends on the number of transitive dependencies. A *deep* constructive trace optimises this process by only looking at the terminal *input keys*, ignoring any intermediate dependencies. The operations capturing this approach are the same as for constructive traces in §4.2.3, but we use the names `recordDCT` and `constructDCT`, where the underlying `DCT` representation need only record information about hashes of inputs, not intermediate dependencies.

Current build systems using deep constructive traces always record hashes of terminal *input keys*, but the technique works equally well if we skip any number of dependency levels (say n levels). The input-only approach is the special case of $n = \infty$, and constructive traces are the special case of $n = 1$. When $n \geq 2$, deep constructive traces require the tasks to be *deterministic*, as otherwise it is possible to violate correctness, as illustrated by an example in §6.4 (see Fig. 11).

A downside of deep constructive traces is that they cannot support early cutoff (§2.3), other than at n levels of dependencies. On the other hand, these traces are particularly useful for *shallow builds*, as discussed in §6.4.

4.3 Efficient Data Structures for Traces

In the examples above, we have used abstract types for the traces. Concretely, in our example code, they are all recorded as lists of:

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

Table 2. Build systems à la carte.

Rebuilding strategy		Scheduling algorithm: Topological §4.1.1	Restarting §4.1.2	Suspending §4.1.3
Dirty bit	§4.2.1	MAKE	EXCEL	-
Verifying traces	§4.2.2	NINJA	-	SHAKE
Constructive traces	§4.2.3	CLOUDBUILD	BAZEL	-
Deep constructive traces	§4.2.4	BUCK	-	NIX

Here `r` is either `Hash v` (verifying traces) or `v` (constructive traces). A real system is highly likely to use a more optimised implementation. Some of the most obvious optimisations are:

- Any system using verifying traces is unlikely to see significant benefit from storing more than one `Trace` per key[8]. Therefore, such systems can store `Map k (Trace k v (Hash v))`, where the initial `k` is the `key` field of `Trace`.
- Any system using `Applicative` dependencies can omit the dependency keys from the `Trace` since they can be recovered from the `key` field.
- Any `Applicative` build system using constructive traces, e.g. CLOUDBUILD, can index directly from the key and results to the output result – i.e. `Map (k, [Hash v]) 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.
- Many cloud build systems store hashes of values in the trace information, then have a separate content-addressable cache which associates hashes with their actual contents.

4.4 Build Systems à la Carte

With the information in this section we can build Table 2, which tabulates combinations of the scheduling algorithm and the rebuilding strategy, providing 12 possible build systems, 8 of which are inhabited by existing build systems (we discuss these systems in §2 and §7.1). Of the remaining 4 spots, all result in workable build systems. The most interesting unfilled spot in the table is suspending constructive traces, which would provide many benefits, and which we title CLOUD SHAKE and explore further in §5.4.

5 BUILD SYSTEMS, CONCRETELY

In the previous sections we discussed the types of build systems, and how they can be broken down into two main components: a scheduler and a rebuilder. In this section we make this abstract distinction concrete, by implementing a number of build systems as a composition of a scheduler and a rebuilder. Our design is expressed by these types (Fig. 5):

```
type Scheduler c i ir k v = Rebuilder c ir k v -> Build c i k v
type Rebuilder c   ir k v = k -> v -> Task c k v -> Task (MonadState ir) k v
```

A `Scheduler` is a function that takes a `Rebuilder` and uses it to construct a `Build` system, by choosing which keys to rebuild in which order. The `Rebuilder` makes use of the persistent build information `ir`, while the scheduler might augment that with further persistent information of its own, yielding `i`.

A `Rebuilder` takes three arguments: a key, its current value, and a `Task` that can (re)compute the value of the key if necessary. It uses the persistent build information `ir` (carried by the state monad) to decide whether to rebuild the value. If doing so is unnecessary, it returns the current value; otherwise it runs the supplied `Task` to rebuild it. In both cases it can choose to update the persistent build information `ir` to reflect what happened. So a `Rebuilder` wraps a `Task c k v`,

[8]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.

```
-- Make build system; stores current time and file modification times
type Time       = Integer
type MakeInfo k = (Time, Map k Time)

make :: Ord k => Build Applicative (MakeInfo k) k v
make = topological modTimeRebuilder

-- A task rebuilder based on file modification times
modTimeRebuilder :: Ord k => Rebuilder Applicative (MakeInfo k) k v
modTimeRebuilder key value task = Task $ \fetch -> do
    (now, modTimes) <- get
    let dirty = case Map.lookup key modTimes of
            Nothing -> True
            time -> any (\d -> Map.lookup d modTimes > time) (dependencies task)
    if not dirty then return value else do
        put (now + 1, Map.insert key now modTimes)
        run task fetch

-- A topological task scheduler
topological :: Ord k => Scheduler Applicative i i k v
topological rebuilder tasks target = execState $ mapM_ build order
  where
    build :: k -> State (Store i k v) ()
    build key = case tasks key of
        Nothing -> return ()
        Just task -> do
            store <- get
            let value = getValue key store
                newTask :: Task (MonadState i) k v
                newTask = rebuilder key value task
                fetch :: k -> State i v
                fetch k = return (getValue k store)
            newValue <- liftStore (run newTask fetch)
            modify $ putValue key newValue
    order = topSort (reachable dep target)
    dep k = case tasks k of { Nothing -> []; Just task -> dependencies task }

-- Standard graph algorithms (implementation omitted)
reachable :: Ord k => (k -> [k]) -> k -> Graph k
topSort   :: Ord k => Graph k -> [k] -- Throws error on a cyclic graph

-- Expand the scope of visibility of a stateful computation
liftStore :: State i a -> State (Store i k v) a
liftStore x = do
    (a, newInfo) <- gets (runState x . getInfo)
    modify (putInfo newInfo)
    return a
```

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

which unconditionally rebuilds the key, to make a `Task (MonadState ir) k v`, which rebuilds the key only if necessary, and does the necessary book-keeping. Note that the resulting `Task` is always monadic; static dependency analysis can be performed on the original `Task Applicative` if needed.

The scheduler calls the rebuilder, but passes it a `fetch` function that the latter calls when it needs the value of a dependent key. This callback returns control to the scheduler, which may in turn call the rebuilder to bring the dependent key up to date, and so on.

These two abstractions are the key to modularity: *we can combine any scheduler with any rebuilder, and obtain a correct build system.* In this section we will write a scheduler for each column of Table 2, and a rebuilder for each row; then combine them to obtain the build systems in the table's body.

5.1 Make

An implementation of Make using our framework is shown in Fig. 7. As promised, its definition is just the application of a `Scheduler`, `topological`, to a `Rebuilder`, `modTimeRebuilder`. We discuss each component in turn, starting with the rebuilder.

The `modTimeRebuilder` uses the pair `MakeInfo k = (now, modTimes)` as persistent build information, carried by a state monad. This `MakeInfo` comprises the *current time* `now :: Time` and the map `modTimes :: Map k Time` of *file modification times*. We assume that the external system, which invokes the build system, updates `MakeInfo` reflecting any file changes between successive builds.

The rebuilder receives three arguments: a `key`, its current `value`, and the applicative `task` that can be used to rebuild the `key` if necessary. The rebuilder first decides if the `key` is `dirty` by consulting `modTimes`: if the `key` is not found, that must mean it has never been built before; otherwise `modTimeRebuilder` can see if any of the `task`'s dependencies (computed by `dependencies`) are out of date. If the `key` is `dirty`, we use `run task` to rebuild it, and update the state with the new modification time of the `key`[9]; otherwise we can just return the current `value`.

Make's scheduler, `topological`, processes keys in a linear `order` based on a topological sort of the statically known dependency graph (see §6.2 for parallel Make). Our definition in Fig. 7 is polymorphic with respect to the type of build information `i` and is therefore compatible with any applicative `rebuilder`. The scheduler calls the supplied `rebuilder` on every `key` in the `order`, and runs the obtained `newTask` to compute the `newValue`. Note that `newTask` has access only to the `i` part of the `Store i k v`, but the rest of the `do` block runs in the `State (Store i k v)` monad; we use the (unremarkable) helper function `liftStore` to fix the mismatch. The `newTask` finds values of the `key`'s dependencies via the `fetch` callback, which is defined to directly read the `store`.

The pre-processing stage uses the function `dependencies`, defined in §3.7, to extract static dependencies from the provided applicative `task`. We compute the linear processing `order` by constructing the graph of keys `reachable` from the `target` 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].

Note that the function `dependencies` can only be applied to applicative tasks, which restricts Make to static dependencies, as reflected in the type `Build Applicative`. Moreover, any other build system that uses the `topological` scheduler will also inherit the same restriction.

5.2 Excel

Our model of Excel uses the `restarting` scheduler and the `dirtyBitRebuilder`, see Fig. 8. The persistent build information `ExcelInfo k` is a pair: (i) a map `k -> Bool` associating a dirty bit with every key, and (ii) a calc chain of type `[k]` recorded in the previous build (§2.2).

The external system, which invokes Excel's build engine, is required to provide a transitively closed set of dirty bits. That is, if a cell is changed, its dirty bit is set, as well as the dirty bit of any other cell whose value might perhaps change as a result. It is OK to mark too many cells as dirty; but not OK to mark too few.

The `dirtyBitRebuilder` is very simple: if the `key`'s dirty bit is set, we `run` the `task` to rebuild the `key`; otherwise we return the current `value` as is. Because the dirty cells are transitively closed,

[9]The real Make relies on the file system to track file modification times, but we prefer to make it explicit here.

```
-- Excel build system; stores a dirty bit per key and calc chain
type Chain k = [k]
type ExcelInfo k = (k -> Bool, Chain k)

excel :: Ord k => Build Monad (ExcelInfo k) k v
excel = restarting dirtyBitRebuilder

-- A task rebuilder based on dirty bits
dirtyBitRebuilder :: Rebuilder Monad (k -> Bool) k v
dirtyBitRebuilder key value task = Task $ \fetch -> do
    isDirty <- get
    if isDirty key then run task fetch else return value

-- A restarting task scheduler
restarting :: Ord k => Scheduler Monad (ir, Chain k) ir k v
restarting rebuilder tasks target = execState $ do
    chain    <- gets (snd . getInfo)
    newChain <- liftChain $ go Set.empty $ chain ++ [target | target `notElem` chain]
    modify $ mapInfo $ \(ir, _) -> (ir, newChain)
  where
    go :: Set k -> Chain k -> State (Store ir k v) (Chain k)
    go _    []         = return []
    go done (key:keys) = case tasks key of
        Nothing -> (key :) <$> go (Set.insert key done) keys
        Just task -> do
            store <- get
            let newTask :: Task (MonadState ir) k (Either k v)
                newTask = try $ rebuilder key (getValue key store) task
                fetch :: k -> State ir (Either k v)
                fetch k | k `Set.member` done = return $ Right (getValue k store)
                        | otherwise           = return $ Left k
            result <- liftStore (run newTask fetch) -- liftStore is defined in Fig. 7
            case result of
                Left dep -> go done $ dep : filter (/= dep) keys ++ [key]
                Right newValue -> do modify $ putValue key newValue
                                     (key :) <$> go (Set.insert key done) keys

-- Convert a total task into a task that accepts a partial fetch callback
try :: Task (MonadState i) k v -> Task (MonadState i) k (Either e v)
try task = Task $ \fetch -> runExceptT $ run task (ExceptT . fetch)

-- Expand the scope of visibility of a stateful computation (implementation omitted)
liftChain :: State (Store ir k v) a -> State (Store (ir, Chain [k]) k v) a
```

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

unlike Make's `modTimeRebuilder`, the `dirtyBitRebuilder` does not need to modify `i` to trigger rebuilds of dependent keys.

Excel's `restarting` scheduler processes keys in the order specified by the calc `chain`. During the build, it constructs a `newChain` for the next build and maintains a set of keys `done` that have been processed. For each non-input `key`, the scheduler tries to rebuild it using a partial `fetch` callback that returns `Either k v` instead of `v`. The callback is defined to fail with `Left dep` when asked for the value of a dependency `dep` that has not yet been processed (and hence may potentially be dirty); otherwise it returns the current value of the dependency by looking it up in the `store`.

```haskell
-- Shake build system; stores verifying traces
shake :: (Ord k, Hashable v) => Build Monad (VT k v) k v
shake = suspending vtRebuilder

-- A task rebuilder based on verifying traces
vtRebuilder :: (Eq k, Hashable v) => Rebuilder Monad (VT k v) k v
vtRebuilder key value task = Task $ \fetch -> do
    upToDate <- verifyVT key (hash value) (fmap hash . fetch) =<< get
    if upToDate then return value else do
        (newValue, deps) <- track task fetch
        modify $ recordVT key (hash newValue) [ (k, hash v) | (k, v) <- deps ]
        return newValue

-- A suspending task scheduler
suspending :: Ord k => Scheduler Monad i i k v
suspending rebuilder tasks target store = fst $ execState (fetch target) (store, Set.empty)
  where
    fetch :: k -> State (Store i k v, Set k) v
    fetch key = do
        done <- gets snd
        case tasks key of
            Just task | key `Set.notMember` done -> do
                value <- gets (getValue key . fst)
                let newTask :: Task (MonadState i) k v
                    newTask = rebuilder key value task
                newValue <- liftRun newTask fetch
                modify $ \(s, d) -> (putValue key newValue s, Set.insert key d)
                return newValue
            _ -> gets (getValue key . fst) -- fetch the existing value

-- Run a task using a callback that operates on a larger state (implementation omitted)
liftRun :: Task (MonadState i) k v
        -> (k -> State (Store i k v, Set k) v) -> State (Store i k v, Set k) v
```

Fig. 9. An implementation of Shake using our framework.

After the newTask is executed (with the help of liftStore) there are two cases to consider:

- The newTask has failed, because one of its dependencies dep has not yet been processed. This indicates that the calculation chain from the previous build is incorrect and needs to be adjusted by moving the dep in front of the key, so that we can restart building the key after the dep is ready.
- The newTask succeeded. The resulting newValue is written to the store, the key is marked as done, and Excel continues to build the rest of the chain.

Note that the task returned by the rebuilder expects a total callback function and cannot be directly executed with the partial callback fetch. We fix the mismatch with the function try that relies on the standard monad transformer ExceptT from the transformers library. We also need the helper liftChain, whose implementation we omit since it is analogous to liftStore in Fig. 7.

5.3 Shake

Our model of Shake is shown in Fig. 9. It stores verifying traces VT k v defined in §4.2.2 as persistent build information and is composed of the suspending scheduler and the vtRebuilder.

The rebuilder executes the verification query verifyVT to determine if the key is upToDate. If it is, the rebuilder simply returns the key's current value. Otherwise it executes the task, obtaining both a newValue and the key's dynamic dependencies deps (see the definition of track in §3.7), which are subsequently recorded in a new verification trace using recordVT.

The suspending scheduler uses a recursive fetch callback, defined similarly to the busy build system (§3.3), that builds a given key, making sure not to duplicate work when called on the same key again in future. To achieve that, it keeps track of keys that have already been built in a set done :: **Set** k. Given a non-input key that has not yet been built, we use the supplied rebuilder to embed the build information i into the task. We then execute the obtained newTask by passing it the fetch function as a callback for building dependencies: the newTask will therefore be suspended while its dependencies are being brought up to date. The newValue obtained by running the newTask is stored, and the key is added to the set done.

The fetch computation runs in the **State** (**Store** i k v, **Set** k) monad. To make **MonadState** i access the i inside the **Store** we use the helper function liftRun (which uses a **newtype** to provide a **MonadState** instance that sees through into the **Store**).

5.4 Cloud Build Systems: Bazel, CloudBuild, Cloud Shake, Buck and Nix

Fig. 10 shows our models of several cloud build systems. Bazel, CloudBuild and Cloud Shake are based on constructive traces (§4.2.3), whereas Buck and Nix use their deep variant (§4.2.4).

The implementation of ctRebuilder is analogous to that of vtRebuilder in Fig. 9, but the verifyVT query is replaced with a more powerful query to constructCT that returns a list of suitable cachedValues by looking them up the cloud cache. If the current value is in the list, we can use it as is. Otherwise, if the list is non-empty, we can use an arbitrary cachedValue. Finally, if the cache has no suitable values, we fall back to executing the task. The obtained newValue and the task's dependencies are recorded as a new constructive trace for future use.

The Bazel build system uses a restarting scheduler whose implementation we omit. It is similar to Excel's restarting scheduler defined in Fig. 8, but instead of building keys in the order specified by the persistently stored calc chain, Bazel uses a *build queue*. The build starts with the queue containing all dirty keys. Similar to Excel, the rebuilding of a key extracted from the queue may fail because one of its dynamic dependencies is dirty. In this case the key is marked as *blocked* and its rebuilding is deferred. Whenever a key is successfully rebuilt, all keys that were previously blocked on it are added back to the queue, and their build is eventually restarted.

Note that although both our model and Bazel's actual implementation supports dynamic dependencies, it is currently not possible to define new monadic build rules in the language available to users. Instead, users have to rely on a (rich) collection of predefined built-in rules, which cover many important instances of dynamic dependencies.

By switching to the topological scheduler, we obtain a model of Microsoft's CloudBuild – an applicative build system that combines conventional scheduling of statically known directed acyclic graphs of dependencies with constructive traces [Esfahani et al. 2016]. Note that we need to convert a monadic ctRebuilder into an applicative one by applying an adapter function adaptRebuilder, which unwraps a given **Task Applicative** and wraps it into **Task Monad**.

Our models of Buck [Facebook 2013] and Nix [Dolstra et al. 2004] use the dctRebuilder, a rebuilder based on deep constructive traces (§4.2.4), whose implementation we omit since it is very similar to that of ctRebuilder. Buck uses the topological scheduler and is therefore an applicative build system, whereas Nix uses the suspending scheduler and is monadic.

Using the abstractions built thus far, we have shown how to combine schedulers with rebuilders to reproduce existing build systems. To us, the most interesting build system as yet unavailable would combine a suspending scheduler with constructive traces – providing a cloud-capable build

```haskell
-- Bazel build system; stores constructive traces
bazel :: (Ord k, Hashable v) => Build Monad (CT k v) k v
bazel = restarting2 ctRebuilder -- implementation of 'restarting2' is omitted (22 lines)

-- A rebuilder based on constructive traces
ctRebuilder :: (Eq k, Hashable v) => Rebuilder Monad (CT k v) k v
ctRebuilder key value task = Task $ \fetch -> do
    cachedValues <- constructCT key (fmap hash . fetch) =<< get
    case cachedValues of
        _ | value `elem` cachedValues -> return value
        cachedValue:_ -> return cachedValue
        [] -> do (newValue, deps) <- track task fetch
                 modify $ recordCT key newValue [ (k, hash v) | (k, v) <- deps ]
                 return newValue

-- Cloud Shake build system, implementation of 'suspending' is given in Fig. 9
cloudShake :: (Ord k, Hashable v) => Build Monad (CT k v) k v
cloudShake = suspending ctRebuilder

-- CloudBuild build system, implementation of 'topological' is given in Fig. 7
cloudBuild :: (Ord k, Hashable v) => Build Applicative (CT k v) k v
cloudBuild = topological (adaptRebuilder ctRebuilder)

-- Convert a monadic rebuilder to the corresponding applicative one
adaptRebuilder :: Rebuilder Monad i k v -> Rebuilder Applicative i k v
adaptRebuilder rebuilder key value task = rebuilder key value $ Task $ run task

-- Buck build system, implementation of 'topological' is given in Fig. 7
buck :: (Ord k, Hashable v) => Build Applicative (DCT k v) k v
buck = topological (adaptRebuilder dctRebuilder)

-- Rebuilder based on deep constructive traces, analogous to 'ctRebuilder'
dctRebuilder :: (Eq k, Hashable v) => Rebuilder Monad (DCT k v) k v

-- Nix build system, implementation of 'suspending' is given in Fig. 9
nix :: (Ord k, Hashable v) => Build Monad (DCT k v) k v
nix = suspending dctRebuilder
```

Fig. 10. BAZEL, CLOUD SHAKE, CLOUDBUILD, BUCK and NIX implemented in our framework; `restarting2` differs from `restarting` used by EXCEL (Fig. 8) in that it does not rely on the calc chain.

system that is minimal, and supports both early cutoff and monadic dependencies. Using our framework it is possible to define and test such a system, which we call CLOUD SHAKE. All we need to do is combine `suspending` with `ctRebuilder`, as shown in Fig. 10.

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 errors 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

We have given simple implementations assuming a single thread of execution, but 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 scheduler:

(1) The `topological` scheduler can build the full dependency graph, and whenever all dependencies of a task are complete, the task itself can be started.
(2) The `restarting` scheduler can be made parallel in a few ways, but the most direct is to have *n* threads reading keys from the build queue. As before, if a key requires a dependency not yet built, it is moved to the end – the difference is that sometimes keys will be moved to the back of the queue not because they are out of date but because of races with earlier tasks that had not yet finished. As a consequence, over successive runs, potentially racey dependencies will be separated, giving better parallelism over time.
(3) The `suspending` scheduler 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 schedulers 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 output. Alas, the real-world is not so obliging. Some examples of impure tasks include:

Untracked dependencies Some tasks depend on untracked values – 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 build, e.g. Excel provides a "function" RANDBETWEEN producing a fresh random number in a specified range on each recalculation. Similarly, build systems like Make and Shake provide volatile *phony rules*. The main difference from non-deterministic tasks is that volatile tasks cannot be cached. They are best modelled as depending on a special key RealWorld whose value is changed in every build.

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 the CPU model and microcode version. Tasks that do have untracked dependencies can be marked as volatile, a technique Excel takes with the INDIRECT function, removing the untracked dependency at the cost of minimality.

Most of the implementations in §5 can deal with non-determinism, apart from Buck, which relies on deterministic tasks, and in turn can optimise the number of roundtrips required to the server.

One tempting way of modelling non-determinism is to enrich `Tasks` 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 :: Tasks MonadPlus String Integer
sprsh3 "B1" = Just $ Task $ \fetch -> (+) <$> fetch "A1" <*> (pure 1 <|> pure 2)
sprsh3 _    = Nothing
```

Such tasks can be modelled in our framework by adjusting the correctness definition (§3.6): instead of requiring that the `result`ing value *equals* the one obtained by recomputing the `task`:

`getValue k result == compute task result`,

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

`` getValue k result `elem` computeND task result ``,

where `computeND :: Task MonadPlus k v -> Store i k v -> [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.

6.4 Cloud Implementation

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

Communication When traces or contents are stored in the cloud, communication can become a bottleneck, so it is important to send only the minimum amount of information, optimising with respect to build system invariants. For example, incremental data processing systems in the cloud, such as Reflow [GRAIL 2017], need to efficiently orchestrate terabytes of data.

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.

Eviction The cloud storage, as modelled in §4.2.3, grows indefinitely, but often resource constraints require evicting old items from the store. When evicting an old value `v`, one can also evict all traces mentioning the now-defunct `hash v`. However, for shallow builds (see below), it is beneficial to keep these traces, allowing builds to "pass-through" hashes whose underlying values are not known, recreating them only when they must be materialised.

Shallow builds Building the end result, e.g. an installer package, often involves many intermediate tasks. The intermediate results may be large, so some cloud build systems are designed to build end products *without materialising* intermediate results, producing a so-called *shallow build* (see an example in §2.4). Some build systems go even further, integrating with the file system to only materialise the file when the user accesses it [Microsoft 2017].

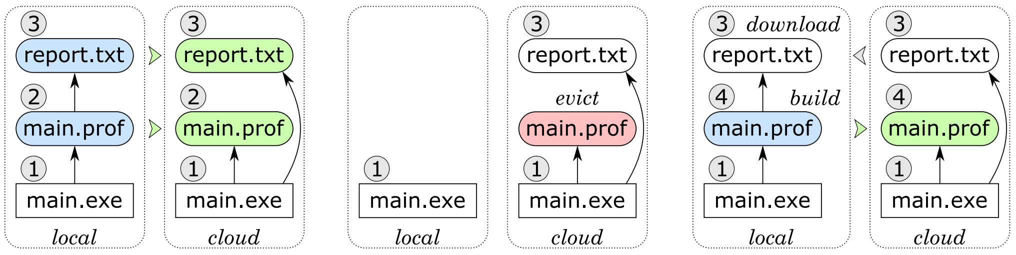
To legitimise shallow builds, we need to relax the correctness Definition 3.1 as follows. Let the `shallow` store correspond to the result of a shallow build. Then `shallow` is correct, if *there exists* a `result` which satisfies all requirements of the Definition 3.1, *such that* `shallow` agrees with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

`getValue k shallow == getValue k result`,

and on the target `key`:

`getValue key shallow == getValue key result`.

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



(a) Initial build (b) Clean up and evict main.prof (c) Build main.prof and report.txt

Fig. 11. A Frankenbuild example: (a) build a human-readable profiling report for main.exe from information dump main.prof produced by a profiling tool, then record deep constructive traces in the cloud, (b) remove built files locally and evict main.prof from the cloud storage, (c) build main.prof by executing the profiler (profiling is non-deterministic, hence the new hash value), then build report.txt by downloading it from the matching deep constructive trace in the cloud, resulting in a Frankenbuild because main.prof and report.txt are inconsistent. New and evicted cloud storage entries are highlighted; file hashes are shown in circles.

Deep constructive traces (§4.2.4) combined with task non-determinism (§6.3) can lead to very subtle bugs, especially in the cloud setting. Fig. 11 shows a *Frankenbuild* [Esfahani et al. 2016] example, where the target report.txt, which is downloaded from the cloud, is inconsistent with its immediate dependency main.prof. This inconsistency is caused by two factors: (i) inherent non-determinism of profiling: running a profiling tool on the very same main.exe will produce different main.prof results every time; and (ii) relying on deep constructive traces, which cache build results based only on the hashes of terminal task inputs (in this case main.exe). Note that the resulting store is incorrect according to all three definitions of correctness: the main definition (§3.6), the variant for non-deterministic tasks (§6.3) and the variant for shallow builds (this section).

6.5 Self-tracking

Some build systems, for example Excel and Ninja, are capable of recomputing a task if either its dependencies change, *or* the task 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 **Tasks Monad** – concretely, first look up the formula, then interpret it:

```
sprsh4 "B1" = Just $ Task $ \fetch -> do
    formula <- fetch "B1-formula"
    evalFormula fetch formula
```

The build systems that have precise self-tracking are all ones which use a *non-embedded domain specific language* to describe build tasks. Those which use a full programming language, e.g. Shake, are faced with the challenge of implementing equality on arbitrary task functions. For such build systems, the pessimistic assumption that any change to the build system potentially changes any build task 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, which can be directly expressed in build systems, such as Pluto [Erdweg et al. 2015]. Another example 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 Excel 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 circular tasks, but 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 dependencies, a choice that most build systems also follow. There are computation frameworks that support tasks with cyclic dependencies under the assumption that tasks are *monotonic* in a certain sense, e.g. see [Pottier 2009] and [Radul 2009].

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, some build systems provide greater flexibility, e.g. Shake permits polymorphic keys and values, allowing types that are stored only in the Shake's build information.

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 task 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 we therefore do not model it in this paper[10].

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. Below we list some notable examples:

- Dune [Jane Street 2018] is a build system designed for OCaml/Reason projects. Its distinguishing feature is that it uses *arrows* [Hughes 2000] rather than monads to model dynamic dependencies, which simplifies static dependency approximation.

[10]See modules Build.Multi and Build.Task.Typed in our framework for models of multiple-output and polymorphic tasks.

- Ninja [Martin 2017] combines the `topological` scheduler of Make with the verifying traces of Shake – our associated implementation provides such a combination. Ninja is also capable of modelling build rules that produce multiple results, a limited form of polymorphism §6.7.
- Nix [Dolstra et al. 2004] has coarse-grained dependencies, with precise hashing of dependencies and downloading of precomputed build products. We provided a model of Nix in §5.4, although it is worth noting that Nix is not primarily 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.
- Redo [Bernstein 2003][Grosskurth 2007][Pennarun 2012] almost exactly matches Shake at the level of detail given here, differing only on aspects like polymorphic dependencies §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. Tup also automatically deletes stale results.

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

```
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. The key difference from our **Tasks** abstraction is that instead of supplying a mapping from outputs to tasks, Fabricate supplies a list of statements, in an order, without declaring what each line produces. There is no need to schedule the statements, and not enough information to do so.

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 key, 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, e.g. see 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 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 (with a few open-source implementations, e.g. Incremental by Jane Street [2015]). We plan to investigate how these insights can be utilised by build systems as future work.

[11]Shake provides support for Fabricate-like build systems – see Development.Shake.Forward in the Shake library.

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 **Tasks** 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 c_{ij} is identified by **C** i j.

```
data Key = A Int | B Int | C Int Int deriving Eq

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

When asked to build **C** 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.

8 CONCLUSIONS

We have investigated multiple build systems, showing how their properties are consequences of two implementation choices: what order you build in and how you decide whether 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 suspending cloud build system. Armed with that blueprint we hope to actually implement such a system as future work.

ACKNOWLEDGMENTS

Thanks to anonymous reviewers and everyone else who provided us with feedback on earlier drafts: Ulf Adams, Arseniy Alekseyev, Dan Bentley, Martin Brüstel, Ulan Degenbaev, Jeremie Dimino, Andrew Fitzgibbon, Georgy Lukyanov, Simon Marlow, Evan Martin, Yaron Minsky, Guillaume Maudoux, Philip Patsch, Michael Peyton Jones, Andrew Phillips, François Pottier, Rohit Ramesh, Irakli Safareli, Zhen Zhang. Your contributions were incredibly valuable.

Andrey Mokhov is funded by a Royal Society Industry Fellowship IF160117 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.

Daniel J. Bernstein. 2003. Rebuilding target files when source files have changed. (2003). http://cr.yp.to/redo.html.

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

Koen Claessen. 1999. A poor man's concurrency monad. *Journal of Functional Programming* 9, 3 (1999), 313—-323.

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/.

GRAIL. 2017. Reflow: A system for incremental data processing in the cloud. (2017). https://github.com/grailbio/reflow.

Alan Grosskurth. 2007. *Purely top-down software rebuilding*. Master's thesis. University of Waterloo.

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

John Hughes. 2000. Generalising monads to arrows. *Science of computer programming* 37, 1-3 (2000), 67–111.

Solomon Hykes. 2013. Docker container: A standardized unit of software. (2013). https://www.docker.com/what-container.

Jane Street. 2015. Incremental: A library for incremental computations. (2015). https://github.com/janestreet/incremental.

Jane Street. 2018. Dune: A composable build system. (2018). https://github.com/ocaml/dune.

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.

François Pottier. 2009. Lazy least fixed points in ML. (2009). http://gallium.inria.fr/~fpottier/publis/fpottier-fix.pdf.

Alexey Radul. 2009. *Propagation Networks: A Flexible and Expressive Substrate for Computation*. Ph.D. Dissertation. MIT.

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