Under consideration for publication in J. Functional Programming

Build Systems à la Carte: Theory and Practice

ANDREY MOKHOV
School of Engineering, Newcastle University, United Kingdom
Jane Street, London, United Kingdom
(*e-mail:* andrey.mokhov@ncl.ac.uk)

NEIL MITCHELL
Facebook, London, United Kingdom
(*e-mail:* ndmitchell@gmail.com)

SIMON PEYTON JONES
Microsoft Research, Cambridge, United Kingdom
(*e-mail:* simonpj@microsoft.com)

Abstract

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 a 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.

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 give 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; early cutoff; self-tracking build systems; and the type of persistently stored build information. In §2 we identify some of these 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) and *whether or not a task is rebuilt* (§5). These choices turn out to be orthogonal, which leads us to a new classification of the design space (§6).
- 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, BUCK, NIX, and EXCEL[1], each by the composition of the two design choices (§6). 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. Writing this paper gave us the insights to combine dynamic dependencies and cloud build systems in a principled way; we evaluate the result in §7.
- We can use the presented abstractions to more clearly explain details from the original SHAKE paper (§5.2.2, §7.2) and develop new cloud build features, which are already in use in industry and in the GHC build system (§§7.4-7.5).

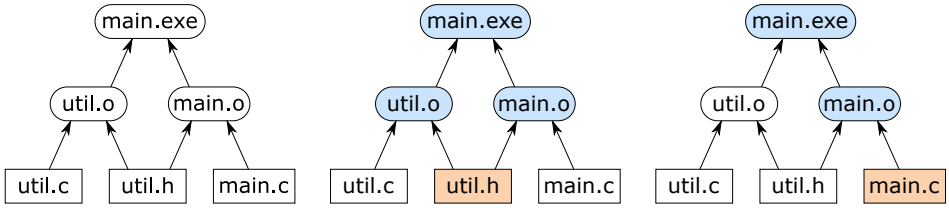
In short, instead of seeing build systems as unrelated points in space, we now see them as locations in a connected landscape, leading to a better understanding of what they do and how they compare, and making it easier to explore other points in the landscape. While we steer clear of many engineering aspects of real build systems, in §8 we discuss these aspects in the context of the presented abstractions. The related work is covered in §9.

This paper is an extended version of an earlier conference paper (Mokhov *et al.*, 2018). The key changes compared to the earlier version are: (i) we added further clarifications and examples to §3, in particular, §3.8 is entirely new; (ii) §4 and §5 are based on the material from the conference paper but have been substantially expanded to include further details and examples, as well as completely new material such as §5.2.2; (iii) §7 is completely new; (iv) §8.1 and §§8.6-8.9 are almost entirely new, and §8.3 has been revised. The new material focuses on our experience and various important practical considerations, hence justifying the "and Practice" part of the paper title.

2 Background

Build systems automate the execution of repeatable tasks, at a scale from individual users up to large organisations. In this section we explore the design space of build systems, using four 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 §6 and §9.

[1] EXCEL appears very different to the others but, seen through the lens of this paper, it is very close.
[2] All our models are executable and are available on Hackage as build-1.0.



(a) Task dependency graph (b) Full rebuild (c) Partial rebuild

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.

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 (or even in parallel) 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, although there are rare exceptions, see §8.5.

[3] There are numerous implementations of MAKE and none comes with a formal specification. In this paper we use a simple 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. To account for this, the actual binary for gcc is often also listed as a dependency, along with any supporting binaries, or standard libraries (such as stdio.h), that are used by gcc.

The fewer tasks are executed in a partial rebuild, the better. To be more specific, consider the following property, which is essential for build systems; indeed, it is their raison d'être:

Definition (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.

This property is tightly linked to build system *correctness*, which we will be ready to define in §3.6; for now, we will use minimality as a guiding principle when exploring the design space of build systems.

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 §4.1 and §6.2.

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 B1.

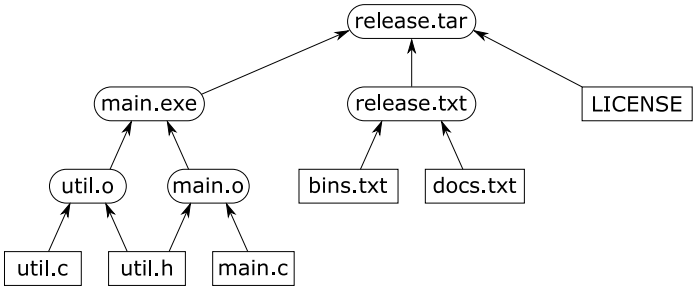
Unlike MAKE, EXCEL does not need to know all task dependencies upfront. Indeed, some dependencies may change *dynamically* during computation. For example:

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

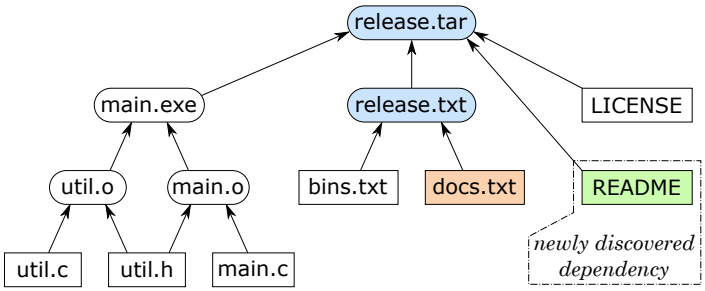
The formula in B1 uses the INDIRECT function, which takes a string and returns the value of the cell with that name. The string evaluates to "A1", so B1 evaluates to 10. However, the dependencies of the formula in B1 are determined by the value of C1, so it is impossible to compute the dependency graph before the build starts. In this particular example the value of C1 is a constant, but it might instead be the result of a long computation chain – so its value will only become available during the build.

To support dynamic dependencies, EXCEL's calculation 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

[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.



(a) Dependency graph produced after the previous build.



(b) The input file docs.txt was modified, hence we rebuild release.txt and release.tar, discovering a new dependency README in the process.

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

aborts. Notice that build always succeeds regardless of the initial calc chain (barring truly circular dependencies); the calc chain is just an optimisation. We refer to this algorithm as *restarting*, and discuss it in more detail in §4.2 and §6.3.

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 §8.8.

2.3 SHAKE: Dynamic Dependencies without Remorse

SHAKE was developed to solve the issue of dynamic dependencies (Mitchell, 2012) without sacrificing the minimality requirement.

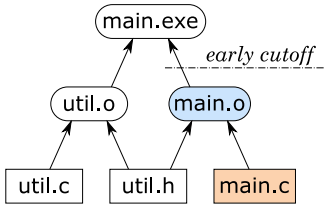


Fig. 3: An early cutoff example: if a comment is added to main.c, the rebuild is stopped after detecting that main.o is unchanged, since this indicates that main.exe and its dependents do not need to be rebuilt.

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 lists all release files. This file is produced by concatenating input text files bins.txt and docs.txt, which list binary and documentation files of the project.
- release.tar is the archive built by executing the command tar on the release files.

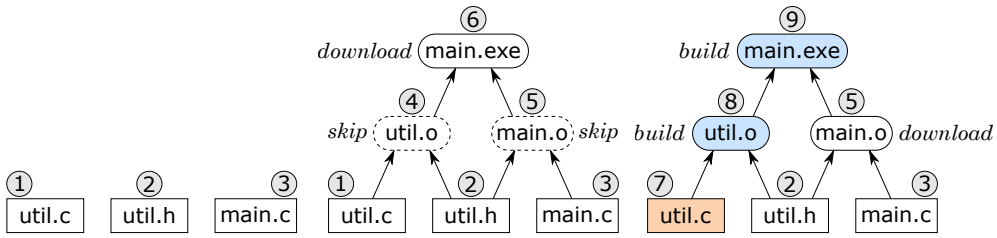
The dependencies of release.tar are not known statically: they are determined by the content of release.txt, which might not even exist before the build. Makefiles cannot express such dependencies, requiring workarounds such as *build phases*, which are known to be problematic (Mokhov *et al.*, 2016).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

```
"release.tar" %> \_ -> do
    need ["release.txt"]
    files <- lines <$> readFile "release.txt"
    need files
    cmd "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, to decide which files need to be rebuilt, it stores the *dependency graph* that is constructed during the previous build (instead of just file modification times or a linear chain). 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 §9. 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 further discussion in §4.3 and a concrete implementation in §6.4.

SHAKE also supports the *early cutoff optimisation*, which is illustrated in Fig. 3. When it executes a task and the result is unchanged from the previous build, it is unnecessary



(a) Download sources (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.

to execute the dependent tasks, and hence SHAKE can stop a build earlier. Not all build systems support early cutoff: SHAKE and BAZEL (introduced below) do, but MAKE and EXCEL do not; see §5.1 for an explanation of why.

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 the 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[6], 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 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

[6] In practice, old entries are regularly evicted from the cloud storage, as further discussed in §8.4.

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.

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. Note that in practice the cost of duplicate work due to the use of a restarting scheduler may often be just a small fraction of the overall build cost (§4.3).

To support cloud builds, BAZEL maintains (i) a *content-addressable cache* that maps the hash of a file's content to the actual content of that file; (ii) a memo table that records all executed build commands with their input and output file hashes. The memo table allows the build engine to bypass the execution of a task, by predicting the hash of the result from the hashes of its dependencies; then the content-addressable cache allows the engine to download the result (if needed) based on the result hash. Further details and a concrete implementation will be provided in §5.3 and §6.5.

2.5 Summary

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

- MAKE stores file modification times, or rather, it relies on the file system to do that.
- EXCEL stores one dirty bit per cell and the 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.

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 §6.5 we demonstrate how to combine the advantages of SHAKE and BAZEL.

3 Build Systems, Abstractly

We have introduced a number of components and characteristics of build systems: tasks, dependencies, early cutoff, minimality, etc. It is easy to get confused. To make all this more concrete, this section presents executable abstractions that can express all the intricacies of build systems discussed in §2, and allow us to construct complex build systems from simple primitives. Specifically, we present the *task* and *build* abstractions in §3.2 and §3.3, respectively. Sections §4, §5 and §6 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 up to date values.

We model a build system concretely, as a Haskell program. 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 associated 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.

3.2 The Task Abstraction

Our first main abstraction is for *task descriptions*:

```
newtype Task  c k v = Task (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` with every non-input

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

run :: c f => Task c k v -> (k -> f v) -> f v
run (Task task) fetch = task fetch

-- 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 (see §4) 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.

key; input keys are associated with `Nothing`. The highly-abstracted type `Task` describes how to build a value 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:

```
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. To run a `Task`, we simply apply the function it holds to a suitable callback (see the definition of the function `run` in Fig. 5).

```
-- 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 identity for monoid m
    Const x <*> Const y = Const (x <> y) -- <> is the binary operation for m

-- Standard types from Control.Monad.Trans.Writer
newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }
tell :: Monad m => w   -> WriterT w m () -- write a value to the log
lift :: Monad m => m a -> WriterT w m a  -- lift an action into WriterT
```

Fig. 6: Standard library definitions.

The code to "compute the value of a formula" in `sprsh1` looks a bit mysterious because it takes place in an `Applicative` computation (McBride & 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, using a callback (`fetch`) to find the values of its dependencies, and limiting side effects to those described by `c`.

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` that, when given a key, brings the key up to date in the store, and returns its value. The function `fetch` runs in the standard Haskell `State` monad (Fig. 6) initialised with the incoming `store` by `execState`. To bring a key `k` up to date, `fetch` asks the task description `tasks` how to compute its value. 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 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 §6.

Nevertheless, `busy` can easily handle the example task description `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 correct 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 – and each call would recompute the formula in `B1`.

To avoid the recomputation, `busy` can keep the set of processed keys in the state monad (in addition to the `store`), treat processed keys as inputs in the `getValue` branch of the `fetch` callback, and include the key `k` into the set of processed keys in the `putValue` branch. This eliminates unnecessary work *within a single build*, but the next build needs to recursively recompute all target's dependencies again even if no inputs changed. To save work *between builds*, it is necessary to store some build information `i` persistently.

3.4 The Need for Polymorphism in Task

The previous example illustrates why the `Task` abstraction is polymorphic in `f`. Recall its definition from §3.2:

```
newtype Task c k v = Task (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 section §6. As we shall see, each build system will use a different `f`, so the task description must not fix `f`.

But the task description cannot possibly work for *any* `f` whatsoever; 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` and, in exchange, the task has access to the operations of class `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, and practically useful, 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`. A simple example of a task with dynamic dependencies is EXCEL formula `INDIRECT("A" & C1)` from §2.2.
- `Task Functor` is somewhat degenerate: a functorial task description cannot even use the application operator `<*>`, which limits dependencies to a linear chain, as e.g. in Docker containers (Hykes, 2013) (ignoring the recent multi-stage builds). 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 Selective` corresponds to task descriptions with *conditional statements*, e.g. EXCEL formula `IF(C1=1,B2,A2)`, where it is possible to statically *over-approximate* the set of task dependencies. Here `Selective` is a type class of *selective applicative functors* (Mokhov *et al.*, 2019), which allows us to model build systems like DUNE (Jane Street, 2018) using the presented framework.
- `Task MonadFail` corresponds to monadic tasks that may fail. For example, the formula `A1/A2` may fail due to division by zero. We will discuss this in §8.1.
- `Task MonadPlus`, `Task MonadRandom` and their variants can be used for describing tasks with a certain type of non-determinism, as discussed in §8.3.
- `Task (MonadState i)` will be used in §6 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 `Task Monad k v`. Consider this spreadsheet example:

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

Note that B1 and B2 statically form a dependency cycle, so MAKE would not be able to order the tasks topologically, but EXCEL, which uses dynamic dependencies, is perfectly happy. We can express this spreadsheet using our task abstraction as follows:

```
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`. Note that in this example one can statically determine the sets of possible dependencies of the formulae IF(C1=1,B2,A2) and IF(C1=1,A1,B1) but this cannot be done in general – recall the spreadsheet with the formula INDIRECT("A" & C1) from §2.2, where the argument of the INDIRECT function is a string computed dynamically during the build. Such tasks can also be captured using `Tasks Monad`:

```
sprsh3 :: Tasks Monad String Integer
sprsh3 "B1" = Just $ Task $ \fetch -> do
    c1 <- fetch "C1"
    fetch ("A" ++ show c1)
sprsh3 _ = Nothing
```

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 §6.

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 fetch)
  where
    fetch :: k -> Identity v
    fetch 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). This is another use of polymorphism in `f`, discussed in §3.4. 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 and pass to the `task`. The result has type `Identity v`, which we unwrap with `runIdentity`.

Definition (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 `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 §8.5.

The presented definition of correctness needs to be adjusted for build systems that support non-deterministic tasks and shallow cloud builds, as will be discussed in sections §8.3 and §8.4, 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: as you can see in the definition of `Task` in Fig. 5, a task is just a function, so all we can do is call it. 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 & Kaposi (2014), but it is extremely cool. We will see a practical use for `dependencies` when implementing applicative build systems, see §6.2.

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`. We need this function to be polymorphic over `m`, because each build system will execute tasks in its own monad, as we shall see in §6.

Here is an implementation of `track` based on the standard Haskell `WriterT` *monad transformer* (Liang *et al.*, 1995), whose main types are listed in Fig. 6:

```
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 function uses the `WriterT` transformer for recording additional information – a list of key/value pairs `[(k, v)]` – when executing 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 §6.4.

Here we show an example of tracking monadic tasks when `m = IO`, by defining a corresponding `fetchIO` of type `String -> IO Integer`, which allows us to demonstrate the dynamic nature of monadic dependencies in GHCi.

```
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 using `fetchIO`.

3.8 Examples of Tasks

In this section we give examples of tasks whose definitions involve different constraints on the computation context: `Functor`, `Applicative`, `Monad` and `MonadState s`. The purpose of these examples is to continue building the intuition behind the `Task` abstraction, and prepare the reader for richer types of tasks that will appear in §6 and §8.

We start with one of the favourite examples for functional programmers – the *Fibonacci sequence* $F_n = F_{n-1} + F_{n-2}$:

```
fibonacci :: Tasks Applicative Integer Integer
fibonacci n = if n < 2 then Nothing else
    Just $ Task $ \fetch -> (+) <$> fetch (n - 1) <*> fetch (n - 2)
```

Here the keys $n < 2$ are input parameters, and one can obtain the usual Fibonacci sequence by picking $F_0 = 0$ and $F_1 = 1$, respectively. Any minimal build system will compute the sequence with memoization, i.e. without recomputing the same value twice.

Dependencies of elements of the Fibonacci sequence are known statically, hence we can express it using `Tasks Applicative`, and benefit from static dependency analysis (§3.7):

```
λ> dependencies (fromJust $ fibonacci 5)
[4,3]
```

Interestingly, the *Ackermann function* – a famous example of a function that is not primitive recursive – cannot be expressed as an applicative task, because it needs to perform an intermediate recursive call to determine the value of one of its dependencies $A(m, n-1)$:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

We therefore use `Tasks Monad` to express this function, binding the dynamic dependency to variable `index`:

```
ackermann :: Tasks Monad (Integer, Integer) Integer
ackermann (m, n)
  | m < 0 || n < 0 = Nothing
  | m == 0    = Just $ Task $ const $ pure (n+1)
  | n == 0    = Just $ Task $ \fetch -> fetch (m-1, 1)
  | otherwise = Just $ Task $ \fetch -> do index <- fetch (m, n-1)
                                           fetch (m-1, index)
```

Functorial tasks are less common than applicative and monadic, but there is a classic example too – the *Collatz sequence*, where given an initial value c_0, we calculate the next value c_n from c_{n-1} either by dividing c_{n-1} by 2 (if it is even) or multiplying it by 3 and adding 1 (if it is odd):

```
collatz :: Tasks Functor Integer Integer
collatz n | n <= 0    = Nothing
          | otherwise = Just $ Task $ \fetch -> f <$> fetch (n - 1)
  where
    f k | even k    = k `div` 2
        | otherwise = 3 * k + 1
```

Functorial tasks correspond to computations with a linear dependency chain. For example, computing the element c_8 of the Collatz sequence starting from $c_0 = 6$ leads to the following dependency chain: $c_0 = 6 \to 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1 = c_8$.

Collatz sequence is a good example of the early cutoff optimisation (§2.3): if we recompute c_8 starting from a different initial value $c_0 = 40$, the resulting computation will have a large overlap with the previous one: $c_0 = 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1 = c_8$. We can therefore stop the recomputation after just two steps, since $c_2 = 10$ has not changed.

Note that we can statically extract even more precise dependency information from functorial tasks compared to applicative tasks. Indeed, we statically know that a `Task Functor` has *exactly one* `dependency`:

```
dependency :: Task Functor k v -> k
dependency task = getConst (run task Const)
```

The `Tasks` abstraction allows us to express pure functions in a way that is convenient for their memoization and incremental recomputation (see §9.3 for a discussion on memoization). If we furthermore need to share computation results via a cloud cache, we can use `Tasks (MonadState s)` that will play an important role in §6. Intuitively, by making a shared state of type `s` available to a task, we give it the abilities to lookup and update cached computation results using the `MonadState` methods `get` and `modify`. For example, below we implement a cloud version of the Ackermann task that uses a `Cache` of type `Map (Integer, Integer) Integer` for sharing results of known Ackermann values.

```
type Cache = Map (Integer, Integer) Integer

cloudAckermann :: Tasks (MonadState Cache) (Integer, Integer) Integer
cloudAckermann (m, n)
    | m < 0 || n < 0 = Nothing
    | m == 0    = Just $ Task $ const $ pure (n+1)
    | n == 0    = Just $ Task $ \fetch -> fetch (m-1, 1)
    | otherwise = Just $ Task $ \fetch -> do
        cache <- get
        case Map.lookup (m, n) cache of
            Nothing -> do index <- fetch (m, n-1)
                          value <- fetch (m-1, index)
                          modify (Map.insert (m, n) value)
                          return value
            Just value -> return value
```

The main case ($m > 0 \wedge n > 0$) starts by looking up the pair of indices `(m, n)` in the `cache`. If the cache has `Nothing`, we calculate the resulting `value` as before and `modify` the cache accordingly; otherwise, if we have a cache hit, we return the obtained value immediately, skipping the actual calculation and thus potentially saving a large amount of work. Indeed you do not want to recompute $A(4,2) = 2^{65536} - 3$ unnecessarily; all of its 19,729 decimal digits have already been helpfully computed, e.g. see Kosara (2008). We will use `MonadState` tasks in our models of cloud build systems in §6.

4 Schedulers

The focus of this paper is on a variety of implementations of `Build c i k v`, given a *user-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 in §5) and then concretely in §6.

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"). In this section we discuss schedulers, leaving rebuilders for §5.

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. The following subsections explore the properties of the three schedulers, and possible implementations.

4.1 Topological Scheduler

The topological scheduler pre-computes a linear order of tasks, which when followed ensures dependencies are satisfied, then executes the required tasks in that order. Computing such a linear order is straightforward – given a task description and a target `key`, first find the (acyclic) graph of the `key`'s dependencies, then compute a topological order. Taking the MAKE example from Fig. 1, we might compute the following order:

1. main.o
2. util.o
3. main.exe

Given the dependencies, we could have equally chosen to build util.o first, but main.exe *must* come last.

The advantage of this scheme is simplicity – compute an order, then execute tasks in that order. In addition, any missing keys or dependency cycles can be detected from the graph, and reported to the user before any work has commenced.

The downside of this approach is that it requires the dependencies of each task in advance. As we saw 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.

4.2 Restarting Scheduler

To handle dynamic dependencies we cannot precompute a static order – we must interleave running tasks and ordering tasks. One approach is just to build tasks in an arbitrary order, and if a task calls `fetch` on an out-of-date key dep, abort the task and build dep instead. Returning to the example from Fig. 1, we might build the tasks as follows:

1. main.exe (abort because it depends on util.o which is out of date)
2. main.o
3. util.o
4. main.exe (restart from scratch, completing successfully this time)

We start with main.exe (an arbitrary choice), but discover it depends on main.o, so instead start building main.o. Next we choose to build util.o (again, arbitrarily), before finally returning to main.exe that now has all its dependencies available and completes successfully.

This approach works, but has a number of disadvantages. Firstly, it requires a technical mechanism to abort a task, which is easy in our theoretical setting with `Task` (see an implementation in §6.3) but leads to engineering concerns in the real world. Secondly, it is not minimal in the sense that a task may start, do some meaningful work, and then abort, repeating that same work when restarted.

As a refinement, 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 from which it can compute a linear order. Since this information may become outdated, BAZEL may also need to abort a task if a newly discovered dependency is out of date.

4.3 Suspending Scheduler

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 when needed. Using the example from Fig. 1, we would build:

- main.exe (suspended)
  ↪ main.o
- main.exe (resumed then suspended again)
  ↪ util.o
- main.exe (completed)

We start building main.exe first as it is the required target. We soon discover a dependency on main.o and suspend the current task main.exe to build main.o, then resume and suspend again to build util.o, and finally complete the target main.exe.

This scheduler (when combined with a suitable rebuilder) provides a minimal build system that supports dynamic dependencies. In our model, a suspending scheduler is easy to write – it makes a function call to compute each dependency. However, a more practical implementation is likely to build multiple dependencies in parallel, which then requires a more explicit task suspension and resumption. To implement suspension there are two standard approaches:

- Blocking threads or processes. This approach is relatively easy, but can require significant resources, especially if a large number of tasks are suspended. In languages with cheap green threads (e.g. Haskell) the approach is more feasible, and it was the original approach taken by SHAKE.
- Continuation-passing style (Claessen, 1999) can allow the remainder of a task to be captured, paused, and resumed later. Continuation passing is efficient, but requires the build script to be architected to allow capturing continuations. SHAKE currently uses this approach.

While a suspending scheduler is theoretically optimal, in practice it is better than a restarting scheduler only if the cost of avoided duplicate work outweighs the cost of suspending tasks. Note furthermore that the cost of duplicate work may often be just a fraction of the overall build cost.

5 Rebuilders

A build system can be split into a scheduler (as defined in §4) and a *rebuilder*. 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?

While §2 explicitly listed the schedulers, the rebuilders were introduced more implicitly, primarily by the information they retain to make their decisions. From the examples we have looked at we see four fundamental rebuilders, each with a number of tweaks and variations within them.

5.1 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. Taking the example from Fig. 1(c), if main.c changes then it would be marked dirty, and main.o and main.exe would be rebuilt as they transitively depend on main.c.

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 (i) if its formula statically refers to a dirty cell, or (ii) if the formula calls a *volatile* 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 (§2.1). However, to achieve early cutoff (§2.3) it would be important to clear the dirty bit after a computation that did not change the value and make sure that keys 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 files 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.

A dirty-bit rebuilder is useful to reduce memory consumption, and in the case of MAKE, to integrate with the file system. However, as the examples show, in constrained environments where a dirty bit is chosen, it is often done as part of a series of compromises.

It is possible to implement a dirty-bit rebuilder that is minimal and supports early cutoff. To do so, the build system should start with all inputs that have changed marked dirty, then a key must be rebuilt if any of its direct dependencies are dirty, marking the key dirty only if the result has changed. At the end of the build all dirty bits must be cleared. This approach only works if all targets are rebuilt each time because clearing dirty bits of keys that are not transitive dependencies of current targets will cause them to incorrectly not rebuild subsequently. To avoid resetting the dirty bit, it is possible to use successive execution numbers, which ultimately leads to an approach we call verifying step traces in §5.2.2.

5.2 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, keeping a *trace* which we can use to *verify* existing values. Taking the example from Fig. 4(c), we might record that the key util.o (at hash 8) depended on the keys util.c (at hash 7) and util.h (at hash 2). Next time round, if the scheduler decides that it is time for util.o to be rebuilt and all keys still have the same hashes as in the recorded trace, there is nothing to do, and we can skip rebuilding. If any of the hashes is different, we rebuild util.o, and record a trace with the new values.

For traces, there are two essential operations – adding a new trace to the trace store, and using the trace store 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` can store only hashes of those values, with type `Hash v`. 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 use `verifyVT`, supplying the key, the hash of its current value, a function for obtaining the hash of the post-build value of any key (using a scheduling strategy as per §4), and the existing trace store `VT`. 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.

The most complex argument of `verifyVT` is a function `fetchHash :: k -> m (Hash v)` to obtain the hash of the post-build value of any key. With an applicative task, `fetchHash` will be called on the statically known task dependencies. However, with a monadic task, the dependencies are not known from the task alone, they are only recorded from previous executions stored in `VT`. If the build system has two traces for a given key `k`, they will both request the same dependency first, since `Task Monad` is deterministic. However, based on that first result, they may then request different subsequent dependencies using

`fetchHash`. A curious result is that for suspending schedulers (§4.3) in many cases the actual build steps are performed as a consequence of checking if a key needs rebuilding!

A verifying trace, and other types of traces discussed in this section, support dynamic dependencies and minimality; furthermore, all traces except for deep traces (§5.4) support the early cutoff optimisation (§2.3).

5.2.1 Trace Representation

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`. Concretely, in our implementations from §6, traces are recorded as lists of:

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

Where `a` is `Hash v` for verifying traces (and `v` for constructive traces, discussed later in §5.3). A real system is highly likely to use a more optimised implementation.

The first optimisation is that any system using `Applicative` dependencies can omit the dependency keys from the `Trace` since they can be recovered from the `key` field (§3.7).

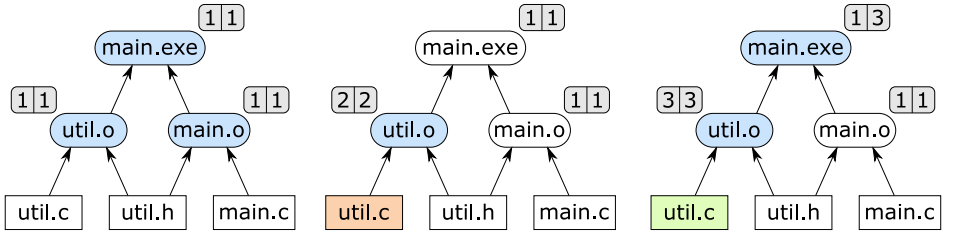
The next optimisation is that there is only very minor benefit from storing more than one `Trace` per key. Therefore, verifying traces can be stored as `Map k (Trace k v (Hash v))`, where the initial `k` is the `key` field of `Trace`, thus making `verifyVT` much faster. Note that storing only one `Trace` per key means that if the dependencies of a key change but the resulting value does not, and then the dependencies change back to what they were before, there will be no valid `Trace` available and the key will therefore have to be rebuilt, whereas a complete list of all historical traces would allow the rebuilding to be skipped. On the other hand, bounding the number of `Trace` structures by the number of distinct keys, regardless of how many builds are executed, is a useful property.

5.2.2 Verifying Step Traces

The SHAKE build system and the associated paper – see §2.3.3 in Mitchell (2012) – use a different trace structure, called *verifying step traces*, which stores less data than verifying traces, and has slightly different early cutoff semantics. Rather than storing the `Hash v` for each dependency, it instead stores `built` time and `changed` time for each `key`, and a list of dependency keys (without the hashes). The resulting `StepTrace` type resembles:

```
data StepTrace k v = StepTrace { key     :: k
                               , result  :: Hash v
                               , built   :: Time
                               , changed :: Time
                               , depends :: [k] }
```

The `built` field is when the `key` last rebuilt. The `changed` field is when the `result` last changed – if the last build changed the value, it will be equal to `built`, otherwise it will be older. The function `recordVT` consults the previous step traces to know whether to keep



(a) Initial full build (b) Change util.c, build util.o (c) Restore util.c, build main.exe

Fig. 7: An example of verifying step traces. The small rectangles show the `changed` (left) and `built` (right) timestamps of each non-input key in the trace store.

the previous `changed` value or change it to `built`. The function `verifyVT` is a bit more subtle; given a key `k` and the hash of its current value `h`, it performs the following steps:

- Find the latest (with respect to the field `built`) step trace matching `k`. If it does not exist, return `False`: `k` was never built before and cannot be verified.
- If `h` does not equal the `result` field of the trace, return `False`: the current `k`'s value was changed externally and thus needs rebuilding.
- For each key `d` in `depends`:
  - Make sure `d` is up-to-date, suspending the current task if needed;
  - If `d`'s latest `changed` time is greater than `k`'s `built` time, return `False`.
- Return `True`: the current `k`'s value is up-to-date.

This approach preserves minimality and early cutoff. A variant with only one `Time` field would lose early cutoff, and indeed corresponds quite closely to MAKE. Furthermore, the `Time` stamp only needs to record which execution of the build is running, so every key built in the same run can share the same `Time` value – it just needs to be monotonically increasing between runs.

This optimisation is useful, at least in the case of SHAKE, to save space. A typical cryptographic hash takes up 32 bytes, while a key (in SHAKE) is an `Int` taking only 4 bytes. Furthermore, SHAKE permits values to be arbitrarily large, and supports a custom value equality (two values can be bit-for-bit unequal but considered equal by SHAKE), hence `Hash v` is not a valid encoding. For applicative tasks, `depends` can be omitted, making the size of a `StepTrace` $O(1)$ instead of $O(n)$, where n is the number of dependencies.

While verifying step traces are mostly an optimisation, there are some observable differences from verifying traces, as demonstrated by an example in Fig. 7. We first make the full build: all keys get a `built` and `changed` of timestamp 1. Next we change util.c and build util.o; the latter is changed as a result and hence both `built` and `changed` are increased to 2. Finally, we change util.c back to what it was originally, and build main.exe. With verifying traces, the hashes of the dependencies of main.exe would be equal to the initial build, and main.exe would not need rebuilding. With verifying step traces, the `changed` field of util.o would increase once more, and main.exe would therefore be rebuilt. As shown in Fig. 7(c), the `changed` field of main.exe remains 1, since the actual value is unchanged. Other than when building subsets of the targets, we are unaware of any other situation where verifying step traces are less powerful.

5.3 Constructive Traces

A verifying trace records only hashes or time stamps, so that it can be small. In contrast, a *constructive* trace also stores the resulting value. Concretely, it records a list of `Trace k v v`, where `Trace` is as defined in §5.2.1. 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 access this additional information with these 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` is similar to `recordVT` from §5.2, but instead of just passing the hash of the resulting value, we pass the actual value. The function `verifyVT` has been replaced with `constructCT`, which instead of taking the hash of the current value as *input*, returns a list of possible values as *output* – there may be more than one, because some build tools are non-deterministic, see §8.3.

Regardless of the chosen scheduler (§4), there are three cases to consider when using a rebuilder based on constructive traces:

- If `constructCT` returns the empty list of possible values, the key must be rebuilt.
- If the current value in the store matches one of the possible values, the build system can skip this key. Here a constructive trace is used for verifying the current value.
- If the current value in the store does not match any possible value, we can use any of the possible values *without* doing any work to build it, and copy it into the store.

Any `Applicative` build system using constructive traces, e.g. CloudBuild (§6.5), can index directly from the key and the hashes of its dependencies to the resulting value, e.g. using a `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 locally, and then make a single call to the server to retrieve the result.

In practice, many cloud build systems store hashes of values in the trace store, i.e. use `Trace k v (Hash v)` entries just like verifying traces, and have a separate content-addressable cache which associates hashes with their actual contents.

5.4 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 §5.3, but we use the names `recordDCT` and `constructDCT`, where the underlying `DCT` representation need only record information about hashes of inputs, not intermediate dependencies.

Concretely, taking the example from Fig. 1, to decide whether main.exe is out of date, a *constructive* trace would look at util.o and main.o (the immediate dependencies), whereas a *deep constructive* trace would look at util.c, util.h and main.c.

Table 2: Build systems à la carte.

		Scheduling algorithm		
Rebuilding strategy		Topological §4.1	Restarting §4.2	Suspending §4.3
Dirty bit	§5.1	MAKE	EXCEL	-
Verifying traces	§5.2	NINJA	-	SHAKE
Constructive traces	§5.3	CLOUDBUILD	BAZEL	-
Deep constructive traces	§5.4	BUCK	-	NIX

An advantage of deep constructive traces is that to decide if main.exe is up to date only requires consulting its inputs, not even considering util.o or main.o. Such a feature is often known as a *shallow build*, as discussed in §8.4.

There are two primary disadvantages of deep constructive traces:

- **Tasks must be deterministic**: If the tasks are not *deterministic* then it is possible to violate correctness, as illustrated by the example in §8.4, see Fig. 12.
- **No early cutoff**: Deep constructive traces cannot support early cutoff (§2.3), since the results of intermediate computations are not considered.

Current build systems using deep constructive traces always record hashes of terminal input keys, but the technique also works 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 $n = 1$. By picking values of n in between we would regain some early cutoff, at the cost of losing such simple shallow builds, while still requiring determinism.

6 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 (§4) and a rebuilder (§5). 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. The result can be summarized in Table 2, which tabulates 12 possible combinations, 8 of which are inhabited by existing build systems (we discuss these systems in §2 and §9.1). Of the remaining 4 spots, all result in workable build systems. The most interesting unfilled spot in the table corresponds to a suspending scheduler composed with a constructive trace rebuilder. Such a build system would provide many benefits; we title it CLOUD SHAKE and explore further in §6.5.

6.1 Concrete Implementations

We can define schedulers and rebuilders more concretely with the following 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`, 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 runs the `Task` returned by the rebuilder passing it a `fetch` callback that the task uses to find values of its dependencies. The callback returns control to the scheduler, which may in turn call the rebuilder again to bring another key up to date, and so on.

These two abstractions are the key to modularity: *we can compose 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 compose them to obtain the build systems in the table's body.

6.2 MAKE

An implementation of MAKE using our framework is shown in Fig. 8. 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`[7]; 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 §8.2 for parallel MAKE). Our definition in Fig. 8 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

[7] The real MAKE relies on the file system to track file modification times, but we prefer to make this explicit in our model.

```haskell
-- 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. 8: An implementation of MAKE using our framework.

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`. Any other build system that uses the `topological` scheduler will inherit the same restriction.

6.3 EXCEL

Our model of EXCEL uses the `restarting` scheduler and the `dirtyBitRebuilder`, see Fig. 9. The persistent build information `ExcelInfo k` is a pair of: (i) a map `k -> Bool` associating a dirty bit with every key, and (ii) a calc chain of type `[k]` recorded from 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, 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`.

After the `newTask` is executed (using `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 executed with the partial callback `fetch`. We fix the mismatch with the function `try` that relies on the standard monad transformer `ExceptT` (Liang *et al.*, 1995). The helper `liftChain` is analogous to `liftStore` in Fig. 8, so we omit its implementation.

```
-- 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 | Set.member k done = return $ Right (getValue k store)
                    | otherwise         = return $ Left k
        result <- liftStore (run newTask fetch) -- liftStore is in Fig. 8
        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 (omitted)
liftChain :: State (Store ir k v) a -> State (Store (ir, Chain [k]) k v) a
```

Fig. 9: An implementation of EXCEL using our framework.

```
-- 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 | Set.notMember key 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 (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. 10: An implementation of SHAKE using our framework.

6.4 SHAKE

Our model of SHAKE (Fig. 10) stores verifying traces `VT k v` defined in §5.2 as persistent build information and is composed of the `suspending` scheduler and the `vtRebuilder`.

The rebuilder performs the `verifyVT` query 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 the trace store 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`).

As discussed in §5.2.2, SHAKE actually uses verifying step traces, but here we choose to focus on the more explicit verifying traces. We have implemented verifying step traces in our framework, and they compose with schedulers as you would hope.

6.5 Cloud Build Systems: BAZEL, CLOUDBUILD, CLOUD SHAKE, BUCK *and* NIX

Fig. 11 shows our models of several cloud build systems. BAZEL, CLOUDBUILD and CLOUD SHAKE are based on constructive traces (§5.3), whereas BUCK and NIX use deep constructive traces (§5.4).

The implementation of `ctRebuilder` is analogous to that of `vtRebuilder` in Fig. 10, 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. 9, 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. Similarly 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 collection of predefined built-in rules, which cover many common 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 with constructive traces (Esfahani *et al.*, 2016). We convert a monadic `ctRebuilder` into an applicative one by applying an adapter `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 rebuilder based on deep constructive traces (§5.4), called `dctRebuilder`, whose implementation we omit since it is very similar to that of `ctRebuilder`. BUCK uses the `topological` scheduler and is an applicative build system, whereas NIX uses the `suspending` scheduler and is therefore monadic.

```
-- Bazel build system; stores constructive traces
bazel :: (Ord k, Hashable v) => Build Monad (CT k v) k v
bazel = restartingQ ctRebuilder

-- A restarting scheduler based on a build queue, omitted (22 lines)
restartingQ :: (Hashable v, Eq k) => Scheduler Monad (CT k v) (CT k v) k v

-- 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. 10
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. 8
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. 8
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. 10
nix :: (Ord k, Hashable v) => Build Monad (DCT k v) k v
nix = suspending dctRebuilder
```

Fig. 11: BAZEL, CLOUD SHAKE, CLOUDBUILD, BUCK and NIX in our framework.

Using the abstractions built thus far, we have shown how to compose schedulers with rebuilders to reproduce existing build systems. To us, the most interesting build system as yet unavailable would compose a suspending scheduler with constructive traces – providing a cloud-capable build 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 compose `suspending` with `ctRebuilder`, as shown in Fig. 11.

7 Experience

We have presented a framework that can describe, and indeed execute in prototype form, a wide spectrum of build systems. But our ultimate goal is a practical one: to use these insights to develop better build systems. Our earlier work on SHAKE (Mitchell, 2012), and applying SHAKE to building GHC (Mokhov *et al.*, 2016), makes progress in that direction.

Based on the theory developed in this paper we have extended SHAKE to become CLOUD SHAKE, the first cloud-capable build system to support both early cutoff and monadic dependencies (§6.5), and used it to implement GHC's (very substantial) build system, HADRIAN (Mokhov *et al.*, 2016). In this section we reflect on our experience of turning theory into practice.

7.1 Haskell as a Design Language

Build systems are surprisingly tricky. It is easy to waffle, and remarkably hard to be precise. As this paper exemplifies, it is possible to use Haskell as a *design language*, to express quite abstract ideas in a precise form – indeed, precise enough to be executed.

Moreover, doing so is extremely beneficial. The discipline of writing executable prototypes in Haskell had a profound effect on our thinking. It forced misconceptions to surface early. It required us to be explicit about side effects. It gave us a huge incentive to design abstractions that had simple types and an explicable purpose.

Consider, for example, our `Task` type:

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

We started off with a much more concrete type, and explored multiple variants. During those iterations, this single line of code gave us a tangible, concrete basis for productive conversations, much more so than general debate about "tasks".

It is also worth noting that we needed a rather expressive language to faithfully express the abstractions that seem natural in this setting. In the case of `Task` we needed: a data constructor with a polymorphic field; a higher-kinded type variable `f :: * -> *`; an even more abstracted kind for `c :: (* -> *) -> Constraint`; and, of course, type classes. Our models have since been translated to Rust (Gandhi, 2018) and Kotlin (Estevez & Shetty, 2019), and in both cases there was a loss of precision due to language-specific limitations.

When thinking about the type constructors over which `f` might usefully range, it turned out that we could adopt the *existing* abstractions of `Functor`, `Applicative`, `Monad` and so on. That in turn led us to a new taxonomy of build systems – see §3.4. In the other direction, trying to express an *existing build system* DUNE in our models led us to finding a new abstraction – the `Selective` type class (Mokhov *et al.*, 2019), which turned out to be useful outside the build systems domain.

The effect of using a concrete design language went well beyond merely *expressing* our ideas: it *directly influenced* our thinking. For example, here are definitions for a scheduler and rebuilder, from §6.1:

```
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
```

These powerful and modular abstractions, which ultimately formed part of the conceptual structure of the paper, emerged fairly late in the project as we repeatedly reviewed, re-

designed, and refactored our executable prototypes. It is hard to believe that we could have developed them without the support of Haskell as a design language.

7.2 *Experience from* SHAKE

The original design of SHAKE has not changed since the initial paper, but the implementation has continued to mature – there have been roughly 5,000 subsequent commits to the SHAKE project[8]. These commits added concepts like *resources* (for handling situations when two build tasks contend on a single external resource), rewriting serialisation to be faster, documentation including a website[9], and add a lot of tests. The biggest change in that time period was an implementation change: moving from blocking threads to continuations for the suspending scheduler. But otherwise, almost all external and internal details remain the same[10]. We consider the lack of change suggestive that SHAKE is based on fundamental principles – principles we can now name and describe as a consequence of this paper.

There are two main aspects to the original SHAKE paper (Mitchell, 2012) that are described more clearly in this paper. Firstly, the rebuilder can now be described using verifying step traces §5.2.2, with a much clearer relationship to the unoptimised verifying traces of §5.2. Secondly, in the original paper the tasks (there called "actions") were described in continuation-passing style using the following data type[11]:

```
data Action k v a = Finished a
                  | Depends k (v -> Action k v a)
```

In this paper we describe tasks more directly, in a monadic (or applicative or functorial) style. But in fact the two are equivalent: `Task Monad k v` is isomorphic to `Action k v v`. To be concrete, the functions `toAction` and `fromAction` defined below witness the isomorphism in both directions.

```
instance Monad (Action k v) where
    return = Finished
    Finished x    >>= f = f x
    Depends ds op >>= f = Depends ds $ \v -> op v >>= f

toAction :: Task Monad k v -> Action k v v
toAction (Task run) = run (\k -> Depends k Finished)

fromAction :: Action k v v -> Task Monad k v
fromAction x = Task (\fetch -> f fetch x)
  where
    f _     (Finished v  ) = return v
    f fetch (Depends d op) = fetch d >>= f fetch . op
```

[8] See `https://github.com/ndmitchell/shake`.
[9] See `https://shakebuild.com`.
[10] The most visible change is purely notational: switching from `*>` to `%>` for defining rules, because a conflicting `*>` operator was added to the Haskell `Prelude`.
[11] The original paper uses concrete types `Key` and `Value`. Here we generalise these types to `k` and `v`, and also add `a` so that `Action k v` can be an instance of `Monad`.

Similarly, in the original paper MAKE tasks were described as:

```
data Rule k v a = Rule { depends :: [k], action :: [v] -> a }
```

Assuming the lengths of the lists `[k]` and `[v]` always match, the data type `Rule k v v` is isomorphic to `Task Applicative k v`, and we can define a similar `Applicative` instance and conversion functions.

By expressing these types using `Task` we are able to describe the differences more concisely (`Monad` vs `Applicative`), use existing literature to determine what is and isn't possible, and explore other constraints beyond just `Monad` and `Applicative`. These and other isomorphisms for *second-order functionals*, i.e. functions of the form

```
forall f. c f => (k -> f v) -> f a
```

for various choices of `c`, are studied in depth by Jaskelioff and O'Connor (2015).

7.3 Experience from CLOUD SHAKE

Converting SHAKE into CLOUD SHAKE was not a difficult process once armed with the roadmap in this paper. The key was the introduction of two new functions:

```
addCloud :: k -> Ver -> Ver -> [[(k, Hash v)]] -> v -> [k] -> IO ()
lookupCloud :: (k -> m (Maybe (Hash v))) -> k -> Ver -> Ver
            -> m (Maybe (v, [[k]], IO ()))
```

These functions are suspiciously like `recordCT` and `constructCT` from §5.3, with their differences perhaps the most illustrative of the changes required[12].

- Two `Ver` arguments are passed to each function. These are the versions of the build script, and the rule for this particular key. If either version changes then it is as though the key has changed, and nothing will match. These versions are important to avoid using stale build products from previous versions of the build script.
- The list of dependencies to `addCloud` is a list of lists, rather than a simple list. The reason is that SHAKE allows a list of dependencies to be specified simultaneously, so they can all be built in parallel.
- The `addCloud` function also takes a list of keys `[k]`, being the files that this rule produces. These produced files include those which are output keys from a rule and those declared with the function `produces`.
- The `lookupCloud` function allows an explicit `Nothing` when looking up a dependent key, since some keys are not buildable.
- The `lookupCloud` function returns at most one result, rather than a list. This change was made for simplicity.

[12] We have made some minor changes from actual SHAKE, like replacing `Key` for `k`, to reduce irrelevant differences.

To integrate these functions into SHAKE we found the most expedient route was to leave SHAKE with verifying traces, but if the verifying trace does not match, we consult the constructive trace. By bolting constructive traces onto the side of SHAKE we avoid re-engineering of the central database. We have not found any significant downsides from the bolt-on approach thus far, so it may be a sensible route to go even if developing from scratch – allowing an optimised verified trace implementation in many cases, and falling back to a more complex implementation (requiring consulting remote servers) only rarely.

The one thing we have not yet completed on the engineering side is a move to hosting caches over HTTP. At the moment all caches are on shared file systems. This approach can use mounted drives to mirror HTTP connections onto file systems, and reuse tools for managing file systems, share caches with rsync, and is simple. Unfortunately, on certain operating systems (e.g. Windows) mounting an HTTP endpoint as a file system requires administrator privileges, so an HTTP cache is still desirable.

7.4 Experience from Using CLOUD SHAKE

While we expected the GHC build system to be the first to take advantage of CLOUD SHAKE, we were actually beaten to it by Standard Chartered who reported[13]:

> *Thanks for the symlinks release, we just finished upgrading this build system to use --share. ... Building from scratch with a warm cache takes around 5 seconds, saving us up to 2 hours. Not bad!*

Converting to a build suitable for sharing is not overly onerous, but nor is it trivial. In particular, a cloud build is less forgiving about untracked operations – things that are wrong but usually harmless in local builds often cause serious problems in a cloud setting. Some things that require attention in moving to a cloud build:

- **Irrelevant differences**: A common problem is that you do not get shared caching when you want it. As one example, imagine two users install gcc on different paths (say /usr/bin/gcc and /usr/local/bin/gcc). If these paths are recorded by the build system, the users won't share cache entries. As another example, consider a compiler that embeds the current time in the output: any users who build that file locally won't get any shared caching of subsequent outputs. Possible solutions include using relative paths; depending only on version numbers for system binaries (e.g. gcc); controlling the environment closely (e.g. using NIX); and extra flags to encourage compilers to be more deterministic.
- **Insufficient produced files**: A build rule must declare all files it produces, so these can be included in the cache. As an example using Haskell, compilation of Foo.hs produces Foo.hi and Foo.o. If you declare the rule as producing Foo.hi, and other rules *depend on* Foo.hi, but *also use* Foo.o after depending on Foo.hi, a local build will probably work (although treating Foo.o as a proper dependency would definitely be preferable). However, if Foo.hi is downloaded from a remote cache, Foo.o will not be present, and subsequent commands may fail (e.g. linking). In practice, most

[13] `https://groups.google.com/d/msg/shake-build-system/NbB5kMFS34I/mZ9L4TgkBwAJ`

issues encountered during the move to cloud builds for GHC were caused by failing to declare produced files.
- **Missing dependencies**: While missing dependencies are always a problem, the move to a cloud build makes them more serious. With local builds, outputs will be built at least once per user, but with a cloud build they might only be built *once ever*.

To help with the final two issues – insufficient dependencies and produced files – we have further enhanced the SHAKE lint modes, coupling them to a utility called FSATrace, which detects which files are read/written by a command line execution. Such information has been very helpful in making the GHC build cloud ready (Eichmann, 2019).

7.5 Experience from Building GHC with SHAKE

HADRIAN is a build system for the Glasgow Haskell Compiler (The GHC Team, 2019). It was developed to replace a MAKE-based build system and solve multiple scalability and maintainability challenges. As discussed in detail by Mokhov *et al.* (2016), most of these challenges were consequences of two key shortcomings of MAKE: (i) poor abstraction facilities of makefiles, notably the need to program in a single namespace of mutable string variables, and (ii) the lack of dynamic dependencies (§2.3). HADRIAN benefits both from SHAKE's features and from the host language Haskell, making the new GHC build system easier to understand and maintain.

Interestingly, although SHAKE is not a self-tracking build system (§8.8), HADRIAN implements a little domain-specific language for constructing build command lines, and then tracks command lines by treating them as a type of values – an example of *partial self-tracking* made possible by SHAKE's support for key-dependent value types (§8.6).

The development of CLOUD SHAKE allows GHC developers to benefit from caching build results between builds. Building GHC 8.8 from scratch takes ∼1 hour on Windows using HADRIAN or the original MAKE-based build system. This time includes building the compiler itself, 29 bundled libraries, such as base (each in vanilla and profiled way), and 6 bundled executables, such as Haddock. One would hope that the build cache would be particularly useful for GHC's continuous integration system that builds and tests every commit but our experience has been mixed. In an ideal case, when a commit does not affect the resulting GHC binaries, e.g. only modifies tests, HADRIAN can build GHC from scratch in just 3 minutes, by simply creating symbolic links to the previously built results stored in the cache. However, if a commit modifies the "Stage 1 GHC" executable – an intermediate compiler built as part of the GHC bootstrapping – any further cache hits become unlikely, thus limiting benefits of the build cache to Stage 1 only (Eichmann, 2019).

A small number of GHC build rules cannot be cached. These rules register libraries in the package database, and rather than producing files, they mutate a shared file. CLOUD SHAKE provides a way to manually label such build rules to exclude them from caching.

One of the benefits of using SHAKE is that we have access to high quality build profiling information, allowing us to compute critical paths and other metrics; see Mitchell (2019) for an overview of SHAKE's profiling features. This information has shown us, for example, that more CPUs would not help (on unlimited CPUs the speed up would be less than 10%), and that a handful of build tasks (two anomalously slow Haskell compilations, and calls to slow single-threaded configure) take up a significant fraction of build time (at least 15%).

8 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.

8.1 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". There are many ways of modelling errors, and in this section we give three simple examples.

One simple approach is to include failures into the type of values `v`, for example, to model a partial store we can use an algebraic data type isomorphic to `Maybe`:

```
data Value = FileNotFound | FileContents String
```

This is convenient if *tasks are aware of failures*. For example, a task may be able to cope with missing files, e.g. if `fetch "username.txt"` returns `FileNotFound`, the task could use the literal string `"User"` as a default value. In this case the task will *depend* on the fact that the file username.txt is missing, and will need to be rebuilt if the user later creates this file. In general, we can use values of type `Either e v` when dealing with failures of type `e`. To automatically convert a "failure-free" task into an equivalent task operating with values of type `Either e v` we can use the following function:

```
liftEither :: Task Monad k v -> Task Monad k (Either e v)
liftEither task = Task $ \fetch ->
    runExceptT $ run task (ExceptT . fetch)
```

Here `liftEither` wraps the result of the given `fetch :: k -> Either e v` into `ExceptT` and then runs the task in the `ExceptT` monad transformer (Liang *et al.*, 1995).

Another approach is to include failures into the computation context `f`. Recall from §3.4 that we require tasks to be polymorphic in `f`, and can therefore choose to execute a `Task` in an `f` with failures, the simplest example being `f = Maybe`. Below we define a callback that returns `Just value` for keys A1 and A2 but fails with `Nothing` on all other keys:

```
fetchA1A2 :: String -> Maybe Integer
fetchA1A2 k = Map.lookup k (Map.fromList [("A1", 10), ("A2", 20)])
```

We can directly run any `Task` with this callback. For example, the task B1 = A1 + A2 from `sprsh1` in §3.2 returns `Just 30`, whereas the task B2 = B1 * 2 returns `Nothing` because `fetchA1A2` fails on B1.

This approach is convenient if *tasks are not aware of failures*, e.g. we can model EXCEL formulae as pure arithmetic functions, and introduce failures "for free" if/when needed by instantiating `Tasks` with an appropriate `f`. In a real system this `f` would be more complex than just `Maybe`, for example `MaybeT (State Spreadsheet)`, thus allowing us

to combine failures with access to EXCEL's spreadsheet state by using the `MaybeT` monad transformer (Liang *et al.*, 1995).

Finally, the task itself might not want to encode failures into the type of values `v`, but instead *demand that* `f` *has a built-in notion of failures*. This can be done by choosing a suitable constraint `c`, such as `Alternative`, `MonadPlus` or even better something specific to failures, such as `MonadFail`. Then both the callback and the task can reuse the same failure mechanism as shown below:

```
class Monad m => MonadFail m where
    fail :: String -> m a

sprsh4 :: Tasks MonadFail String Integer
sprsh4 "B1" = Just $ Task $ \fetch -> do
    a1 <- fetch "A1"
    a2 <- fetch "A2"
    if a2 == 0 then fail "division by 0" else return (a1 `div` a2)
sprsh4 _ = Nothing
```

With this approach we can implement a build system that accepts `Tasks MonadFail k v` and handles errors by aborting the build early and returning `Either String (Store i k v)` instead of just `Store i k v` as in our `Build` abstraction §3.3. One possible implementation of such a failure-handling build system is based on adding an extra `Either String` layer into the monad stack, e.g. augmenting the monad `State (Store i k v)` used by the schedulers in §6 with exceptions. We omit the actual implementation: while fairly direct it is also tedious due to additional wrapping and unwrapping.

8.2 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 are restricted 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, if the build order is persisted over successive runs (as in EXCEL), potentially racey dependencies will be separated, giving better parallelism over time.
3. The `suspending` scheduler can be made parallel by starting multiple dependencies in parallel. One approach is to make the request for dependencies take a list of keys, as implemented by SHAKE. Another approach is to treat the `Applicative` dependencies of a `Task Monad` in parallel, as described by Marlow *et al.* (2014).

Once sufficient parallelism is available the next challenge is preventing excess parallelism and machine resource starvation, which is usually achieved with a thread pool/limit.

The actual implementation of the parallel schedulers is not overly onerous, but neither is it beautiful or informative, so we omit it.

8.3 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 any result from a possible set. As an example, GHC programs compiled using parallelism can change the order in which unique variables are obtained from the name 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 readily modelled as depending on a special key RealWorld whose value is changed in every build.

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 with untracked dependencies can be marked volatile, a technique EXCEL takes with the INDIRECT function, fixing the untracked dependency at the cost of minimality.

Most of the implementations in §6 can deal with non-determinism, apart from BUCK, which uses the assumption of task determinism to reduce the number of cloud lookups: if all tasks are deterministic, one needs just a single cloud lookup for obtaining the value of the target key from the hash of its terminal inputs. This is highly desirable not only for shallow builds, but for non-shallow builds too, because under the assumption of task determinism *all* intermediate values are fully determined by the terminal inputs, and can therefore be requested in a single batch query to the server.

One way of modelling non-determinism is to *demand that* `f` *has a built-in source of non-determinism*, for example, by enriching `Tasks Monad` to `Tasks MonadRandom` that has access to an additional method `getRandom :: (Integer, Integer) -> m Integer`, i.e. a source of random numbers in a specified range. Here is a task description corresponding to a spreadsheet with the formula B1 = A1 + RANDBETWEEN(1,2):

```
sprsh5 :: Tasks MonadRandom String Integer
sprsh5 "B1" = Just $ Task $ \fetch -> do a1 <- fetch "A1"
                                         r  <- getRandom (1,2)
                                         return (a1 + r)
sprsh5 _    = Nothing
```

Such tasks can be modelled in our framework by adjusting the correctness definition (§3.6): instead of requiring that the produced value *equals the result* of recomputing the task, we now require that produced value *belongs to the set of possible results* of recomputing the task, e.g. the set {A1 + 1, A1 + 2} in the above example.

Interestingly, `Task MonadRandom` is powerful enough to express 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.

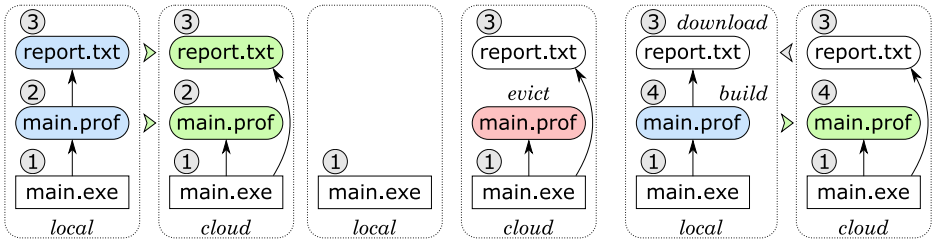
8.4 Cloud Implementation

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 §5.3, 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 target, e.g. an installer package, often involves many intermediate tasks. The values produced by these intermediate tasks may be large, so some cloud build systems are designed to build end targets *without materialising* any intermediate values, 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).

Shallow builds have a slightly weaker correctness property than in the Definition 3.6. Instead of demanding that *all* keys reachable from the target match, we only require matches for the target itself and the *input keys* reachable from the target.

As described in §7.4, non-determinism (§8.3) is harmful for cloud builds, reducing the number of cache hits. However, for deep constructive traces (§5.4) it is much worse, even leading to *incorrect results*. Fig. 12 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



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

Fig. 12: Frankenbuild example: (a) build a human-readable profiling report for main.exe from information dump main.prof produced by a profiling tool, saving deep constructive traces in the cloud, (b) remove built files locally and evict main.prof from the cloud storage, (c) rebuild main.prof (profiling is non-deterministic, hence a new hash value), then build report.txt by downloading it from the matching deep constructive trace in the cloud. The result is a Frankenbuild because main.prof and report.txt are inconsistent. New and evicted cloud storage entries are highlighted; file hashes are shown in circles.

that cache build results based on the hashes of terminal task inputs (in this case main.exe). The result violates all three definitions of correctness: the main definition (§3.6), the variant for non-deterministic tasks (§8.3) and the variant for shallow builds (this section).

8.5 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.

We choose not to deal with cyclic dependencies – a choice that most build systems also follow. There are computation frameworks that support dependency cycles under the assumption that tasks are *monotonic* in a certain sense (Pottier, 2009; Radul, 2009).

8.6 Key-dependent Value Types

Key-dependent value types allow a build system to work with multiple different types of values, where the type of any particular value is determined by the key. As an example of

why this might be useful, consider a build system where keys can be files (whose contents are strings) or system executables (represented by their version number) – using a single type for both values reduces type safety. SHAKE permits such key-dependent value types, e.g. see the *oracle* rule in Mitchell (2012), and users have remarked that this additional type safety provides a much easier expression of concepts (Mokhov *et al.*, 2016).

We can encode key-dependent value types using generalised algebraic data types, or GADTs (Peyton Jones *et al.*, 2006). The idea is to replace the callback `fetch :: k -> f v` by its more polymorphic equivalent `fetch :: k v -> f v`, where `k v` is a GADT representing keys tagged by the type `v` of corresponding values. The variable `k` has changed from kind $*$ (a type), to `* -> *` (a type function), permitting the key to constrain the type of the value. The idea is best explained by way of an example:

```
data Version = Version { major :: Int, minor :: Int }
    deriving (Eq, Ord)

data Key a where
    File    :: FilePath -> Key String
    Program :: String -> Key Version
```

Here we extend the usual mapping from file paths to file contents with an additional key type `Program` which maps the name of a program to its installed `Version`. The task abstraction needs to be adjusted to cope with such keys (the suffix `T` stands for "typed"):

```
type Fetch k f = forall v. k v -> f v

newtype TaskT c k v = TaskT (forall f. c f => Fetch k f -> f v)

type TasksT c k = forall v. k v -> Maybe (TaskT c k v)
```

The changes compared to the definition in §3.2 are minimal: (i) the `TaskT` now uses a typed `Fetch` callback (we define a separate type synonym only for readability), and (ii) the type of `TasksT` is now polymorphic in `v` instead of being parameterised by a concrete `v`. The example below demonstrates how `fetch` can be used to retrieve dependencies of different types: the rule `release.txt` concatenates the contents of two `File`s, while the rule `main.o` uses the numeric `Program "gcc"` to determine how the `source` file should be compiled.

```
example :: TasksT Monad Key
example (File "release.txt") = Just $ TaskT $ \fetch -> do
    readme  <- fetch (File "README")
    license <- fetch (File "LICENSE")
    return (readme ++ license)
example (File "main.o") = Just $ TaskT $ \fetch -> do
    let source = "main.c"
    version <- fetch (Program "gcc")
    if version >= Version 8 0 then compileNew source
                              else compileOld source
example _ = Nothing
```

Note that like all key-dependent tasks, this example could be expressed without key-dependence, at the cost of some type safety. As we will see in §8.7 and §8.8, using key-dependent value types can make it easier to work with more complicated tasks.

8.7 Multiple-Output Build Tasks

Some build tasks produce multiple output keys – for example ghc A.hs produces both A.hi and A.o. This pattern can be encoded by having a key A.hi+A.o which produces both results, then each of A.hi and A.o can extract the relevant result from A.hi+A.o. We can express this pattern more clearly with the extra type safety from §8.6:

```
data Key a where
    File  :: FilePath   -> Key String
    Files :: [FilePath] -> Key [String]

task :: TasksT Monad Key
task (File "A.hi") = Just $ TaskT $ \fetch -> do
    [hi,o] <- fetch (Files ["A.hi","A.o"])
    return hi
task (File "A.o") = Just $ TaskT $ \fetch -> do
    [hi,o] <- fetch (Files ["A.hi","A.o"])
    return o
task (Files ["A.hi","A.o"]) = Just $ TaskT $ \fetch ->
    compileWithGHC "A.hs"
```

One awkward aspect is that both A.hi and A.o must ask for exactly the same `Files` key. If one `File` key swapped the order of the list to `Files` then it would likely run GHC twice. To help construct a well-formed multiple-output build task it is convenient to partition the set of keys into *elementary subsets*. We can encode such a partition with a function from any key to all the members of its subset (in a consistent order).

```
type Partition k = k -> [k]
```

In our example the partition function on either of the output names `"A.hi"` or `"A.o"` would return `["A.hi","A.o"]`.

With a suitable `Partition` it is possible to create a mapping that resolves `File` keys automatically into the correct `Files` key.

```
task :: Partition FilePath -> TasksT Monad Key
task partition (File k) = Just $ TaskT $ \fetch -> do
    let ks = partition k
    let Just i = elemIndex k ks
    vs <- fetch (Files ks)
    return (vs !! i)
task (Files ["A.hi","A.o"]) = Just $ TaskT $ \fetch ->
    compileWithGHC "A.hs"
... -- more tasks for elementary subsets as required
```

In the above function we compute the elementary subset `ks` of the given key `k`, find the index of `k` within the subset (using `elemIndex`), run the `fetch` to build every key in the subset, then extract out the value corresponding to `k` (using the indexing operation `!!`).

8.8 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:

```
sprsh6 "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; that is, a task is described by a data structure (or syntax tree), and tasks can be compared for equality by comparing those data structures. Build systems that use a full programming language, e.g. SHAKE, are faced with the challenge of implementing equality on arbitrary tasks – and a task is just a function. For such build systems, the only safe approach is to assume (pessimistically) that any change to the build system potentially changes any build task – the classic example being build tasks depending on the makefile itself.

Below we show how to implement self-tracking in a build system that allows users to describe build tasks by *scripts* written in a non-embedded domain specific language. We will denote the type of scripts by `s`, and will assume that scripts are indexed by keys `k` just like all other values `v`. More specifically, we use the following GADT to tag keys `k` with corresponding result types: `s` for scripts, and `v` for other values.

```
data Key k v s a where
    Script :: k -> Key k v s s -- Keys for build scripts
    Value  :: k -> Key k v s v -- Keys for all other values
```

The function `selfTracking` defined below is a generalisation of the approach explained in the above EXCEL example `sprsh6`. The function takes a parser for scripts, of type `s -> Task Monad k v`, and a description of *how to build all scripts*, of type `Tasks Monad k s`. For `sprsh6`, the latter would simply fetch B1-formula when given the key B1 and return `Nothing` otherwise, but the presented approach can cope with much more sophisticated scenarios where scripts themselves are derived from "script sources", e.g. all C compilation scripts can be obtained from a single pattern rule, such as gcc -c [source] -o [object]. The

resulting typed task description `TasksT Monad (Key k v s)` tracks both values and scripts that compute them, and is therefore self-tracking.

```
selfTracking :: (s -> Task Monad k v) -> Tasks Monad k s
                                      -> TasksT Monad (Key k v s)
selfTracking parse tasks key = case key of
    Script k -> getScript <$> tasks k
    Value  k -> runScript <$> tasks k
  where
    -- Build the script and return it
    getScript :: Task Monad k s -> TaskT Monad (Key k v s) s
    getScript task = TaskT $ \fetch -> run task (fetch . Script)
    -- Build the script, parse it, and then run the obtained task
    runScript :: Task Monad k s -> TaskT Monad (Key k v s) v
    runScript task = TaskT $ \fetch -> do
        script <- run task (fetch . Script)
        run (parse script) (fetch . Value)
```

It is possible to implement `selfTracking` without relying on GADTs and typed tasks presented in §8.6, at the cost of using partial functions.

8.9 File Watching vs Polling

Most build systems use files for their inputs. When running a build, to check if a file is up-to-date, one option is to take the modification time or compute the content hash of each input file. However, for the largest projects, the mere act of checking the modification time for all the input files can become prohibitive. To overcome that problem, build systems targeted at large scale, e.g. BAZEL and BUCK, rely on file watching API's to detect when a file has changed. Using that information the build can avoid performing many file accesses.

File-watching build systems face new engineering challenges. Firstly, a running task may become *obsolete* due to a concurrent modification of its direct or transitive dependencies. To minimise the amount of unnecessary work, a scheduler can *cancel* the obsolete task and restart it once its dependencies are up-to-date. Note that topological schedulers (§4.1) cannot be easily adapted to support such situations, since a new topological sort may be required when file dependencies change. A further complication is that spurious dependency cycles may form during a series of file modifications, and a file-watching build system integrated with an IDE should be able to cope with such spurious cycles gracefully instead of terminating with an error.

Build systems that run continuously are also more likely to encounter errors caused by concurrent modification of *build outputs*. For example, if an output file is checked into the source repository then downloading a new version of the file can interfere with the build task producing it, resulting in a corrupted output that the build system will be unable to detect. This problem can be solved by ensuring that tasks have an *exclusive access* to their outputs, e.g. by sandboxing the tasks whose outputs can be modified externally.

9 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.

9.1 Other Build Systems

Most build systems, when viewed at the level we talk, can be captured with minor variations on the code presented in §6. Below we list some notable examples:

- DUNE (Jane Street, 2018) is a build system designed for OCaml/Reason projects. Its original implementation used *arrows* (Hughes, 2000) rather than monads to model dynamic dependencies, which simplified static dependency approximation. DUNE was later redesigned to use a flavour of selective functors (Mokhov *et al.*, 2019), making it a closer fit to our abstractions.
- 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 multiple value types §8.6.
- 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 §6.5, 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 new features.
- REDO (Bernstein, 2003; Grosskurth, 2007; Pennarun, 2012) almost exactly matches SHAKE at the level of detail given here, differing only in aspects like rules producing multiple files §8.6. While REDO predates SHAKE, they were developed independently; we use SHAKE as a prototypical example of a monadic build system because its implementation presents a closer mapping to our `Task` abstraction.
- 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 main difference from our `Tasks` abstraction is that instead of supplying a mapping from keys to tasks, a FABRICATE script supplies a list of build statements, in a *user-scheduled order*, without declaring what each statement reads or write.

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[14]. However, in our pure model the mapping is not perfect as `gcc` writes to arbitrary files whose locations are not known in advance. One way of capturing arbitrary writes in our model is to switch from one callback `fetch` to *two callbacks*, say `read` and `write`, allowing us to track both reads and writes separately.

9.2 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 §8.6 can be used to model this feature to a limited degree). Another important optimisation that self-adjusting computation engines often support is the incremental processing of *deltas*, where instead of marking a value as "changed to 8", one can mark it as "changed by $+1$", assuming it was equal to 7 before. When a delta is small, it can often be propagated to the output more efficiently than by recomputing the output value from scratch.

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.

9.3 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 (§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.

[14] SHAKE provides support for FABRICATE-like build systems – see Development.Shake.Forward.

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 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.

10 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, which we have implemented and use in our day-to-day development.

Acknowledgements

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's research was funded by a Royal Society Industry Fellowship IF160117 on the topic "Towards Cloud Build Systems with Dynamic Dependency Graphs".

References

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

Acar, Umut A, Blume, Matthias, & Donham, Jacob. (2007). A consistent semantics of self-adjusting computation. *Pages 458–474 of: European Symposium on Programming*. Springer.

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

Capriotti, Paolo, & Kaposi, Ambrus. (2014). Free applicative functors. vol. 153. Open Publishing Association.

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

Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2001). *Introduction to algorithms*. MIT Press.

De Levie, R. (2004). *Advanced excel for scientific data analysis*. Advanced Excel for Scientific Data Analysis. Oxford University Press.

Demers, Alan, Reps, Thomas, & Teitelbaum, Tim. (1981). Incremental evaluation for attribute grammars with application to syntax-directed editors. *Pages 105–116 of: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM.

Dolstra, Eelco, De Jonge, Merijn, Visser, Eelco, *et al.*. (2004). Nix: A safe and policy-free system for software deployment. *Pages 79–92 of: LISA*, vol. 4.

Eichmann, David. (2019). *Exploring Cloud Builds in Hadrian*. `https://web.archive.org/web/20191008171120/https://well-typed.com/blog/2019/08/exploring-cloud-builds-in-hadrian/`.

Erdweg, Sebastian, Lichter, Moritz, & Weiel, Manuel. (2015). A sound and optimal incremental build system with dynamic dependencies. *Acm sigplan notices*, **50**(10), 89–106.

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

Estevez, Paco, & Shetty, Devesh. (2019). *Translation of Build Systems à la Carte to Kotlin*. `https://web.archive.org/web/20191021224324/https://github.com/arrow-kt/arrow/blob/paco-tsalc/modules/docs/arrow-examples/src/test/kotlin/arrow/BuildSystemsALaCarte.kt`.

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

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

Gandhi, Varun. (2018). *Translation of Build Systems à la Carte to Rust*. `https://web.archive.org/web/20191020001014/https://github.com/cutculus/bsalc-alt-code/blob/master/BSalC.rs`.

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

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

Grosskurth, Alan. (2007). *Purely top-down software rebuilding*. M.Phil. thesis, University of Waterloo.

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

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

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

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

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

Jaskelioff, Mauro, & O'Connor, Russell. (2015). A representation theorem for second-order functionals. *Journal of functional programming*, **25**.

Kosara, Robert. (2008). *Decimal expansion of A(4,2)*. `https://web.archive.org/web/20080317104411/http://www.kosara.net/thoughts/ackermann42.html`.

Levenshtein, Vladimir I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Pages 707–710 of: Soviet physics doklady*, vol. 10.

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

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

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

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

Microsoft. (2011). *Excel recalculation (msdn documentation)*. `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*. `https://www.gvfs.io/`.

Mitchell, Neil. (2012). Shake before building: Replacing Make with Haskell. *Pages 55–66 of: ACM SIGPLAN Notices*, vol. 47. ACM.

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

Mitchell, Neil. (2019). *Ghc rebuild times – shake profiling*. `https://neilmitchell.blogspot.com/2019/03/ghc-rebuild-times-shake-profiling.html`.

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

Mokhov, Andrey, Mitchell, Neil, & Peyton Jones, Simon. (2018). Build systems à la carte. *Proc. acm program. lang.*, **2**(ICFP), 79:1–79:29.

Mokhov, Andrey, Lukyanov, Georgy, Marlow, Simon, & Dimino, Jeremie. (2019). Selective applicative functors. *Proc. acm program. lang.*, **3**(ICFP).

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

Peyton Jones, Simon, Vytiniotis, Dimitrios, Weirich, Stephanie, & Washburn, Geoffrey. (2006). Simple unification-based type inference for GADTs. *Pages 50–61 of: ACM SIGPLAN Notices*, vol. 41. ACM.

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

Radul, Alexey. (2009). *Propagation networks: A flexible and expressive substrate for computation*. Ph.D. thesis, MIT.

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

The GHC Team. (2019). *The Glasgow Haskell Compiler homepage*. `https://www.haskell.org/ghc/`.