Implementing Applicative Build Systems Monadically

Yi Fan (Bob) Yang
bobyf@gmail.com
Facebook

Neil Mitchell
ndmitchell@gmail.com
Facebook

1 Introduction

Build systems are typically driven by a user script, e.g. the build system Buck [1] is driven by BUCK files. Mokhov et. al. [7] classified build systems as either *applicative* or *monadic*, based on whether the build system required *static dependencies*, or allowed *dynamic dependencies*, respectively. Buck does not allow dynamic dependencies, so is classified as applicative.

In this paper we show that applicative build systems are only applicative *from the perspective of the user*, and are internally actually monadic. In particular, the actions executed are entirely dependent on the values in the user script. Following this observation, we rewrote Buck, an “applicative build system”, using a monadic incremental computation engine, and saw improvements in both code complexity and performance.

2 Buck Design

Buck builds are configured by Python scripts supplied by the user. We can define a C++ binary (with a target named `main`) which depends on a C++ library (with a target named `hello_string`) as:

```
cxx_binary(
  name = 'main',
  srcs = ['hello-buck.cxx'],
  deps = [':hello_string'],
)

cxx_library(
  name = 'hello_string',
  srcs = ['hello-string.cxx'],
)
```

In this example, the `srcs` attribute says what source code is part of each target, and the `deps` attribute of `main` says the binary depends on the library.

2.1 Buck phases

Buck is implemented as four significant *phases*, as shown in Figure 1. These phases are:

Parsing The parsing step reads all the relevant build files, parsing them and evaluating them as Python. It converts the result of the Python function calls (e.g. `cxx_binary`) to a JSON representation.

Marshaling The resulting JSON is used to generate a *target graph*, which stores typed information about each rules attributes in the nodes. The dependencies in the `deps` attributes are translated into edges between these nodes.

Graph Transformation Given the target graph, Buck then constructs an *action graph*. The action graph nodes are actual actions/commands that can be executed to produce outputs. Edges between nodes represent that one command uses the output of another command. While the target graph is closely based on the information supplied by the user, the action graph is created based on rule implementations that may expand a single target graph node into multiple action graph nodes.

Execution Finally, Buck executes the commands on the action graph, according to the dependency edges, to produce the final build artifact.

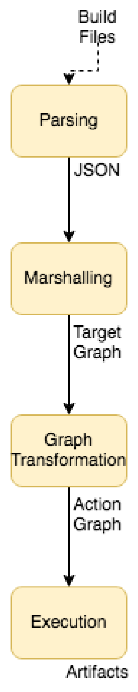

Figure 1. The phases of Buck.

Importantly, the entire graph of actions is calculated *before* any commands are executed. Therefore Buck is an applicative build system.

2.2 Monadic Buck

A monadic build system is one where actions influence the resulting dependency structure. Therefore, what we consider to be an “action” determines whether a system is monadic. If we consider only executing external commands to be actions, then Buck is indeed applicative. However, Buck also reads Python files and evaluates them, and those actions are almost entirely responsible for defining the resulting target graph. The action of converting a target graph to an action graph is also a large action which impacts the resulting graph. Therefore, if we look at all the significant actions within Buck, we see that Buck is definitely monadic.

BAPL '20, London, UK
2020. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Moreover, almost every applicative build system starts by reading the user build script, which is then used to influence the commands performed. Therefore, every build system implementation is monadic. Such a result should not be too surprising – if we look at the implementations in §5 of [7], we find that they are all monadic, even for applicative build systems.

3 Implementing Buck Monadically

In this section we outline how we used the techniques from [7], combined with the observation that applicative build systems are secretly monadic, to implement Buck on a more principled foundation. We first describe a generic monadic computation engine, then translate the phases from §2.1 to it, and finally implement it. This implementation technique does *not* change the power of Buck for end users – targets still cannot have dynamic dependencies.

3.1 An Incremental Computation Engine

A generic incremental computation engine is a stateful object that offers the API `compute(k) -> r`. Given a *computation key* `k`, we produce the corresponding *computation result* `r` of a particular *computation*. The computation for `k` could require *dependencies*, which are the results of other computation keys.

This definition of computations is equivalent to a monadic task from A. Mokhov, et al.[7]. That paper modeled a task as a mapping of keys to values by either an applicative or monadic function. We can treat computation keys the same as keys for tasks, and computation results the same as values for tasks. Since the dependencies of computations are unrestricted, they can be dynamic, corresponding to a monadic function in tasks.

The incremental computation engine can then be thought of as a build engine for monadic tasks, i.e. a monadic build engine.

3.2 Modeling Buck as Computations

We can model all of Buck's phases in §2.1 as computations, illustrated in Figure 2:

Parsing The computation keys are the paths of BUCK files and results are some representation of the parsed information as JSON. There are no *dependencies*, because the parsing of one BUCK file is independent of other BUCK files.

Marshaling This computation has *targets* (identifiers of nodes) as keys, mapping to target graph nodes. The dependencies are on the results of the Parsing computation of the corresponding target.

Graph Transformation This computation maps targets to action graph nodes. This computation both depends on the target graph node of the corresponding target from the Marshaling computation, and the action graph nodes of this computation based on the dependency edges of the corresponding target graph node.

Execution This computation maps a target to a file produced by the build. The dependencies are the action graph node of the corresponding target from the previous Graph Transformation computation, and all the files produced by this computation itself for the dependencies of the action graph node.

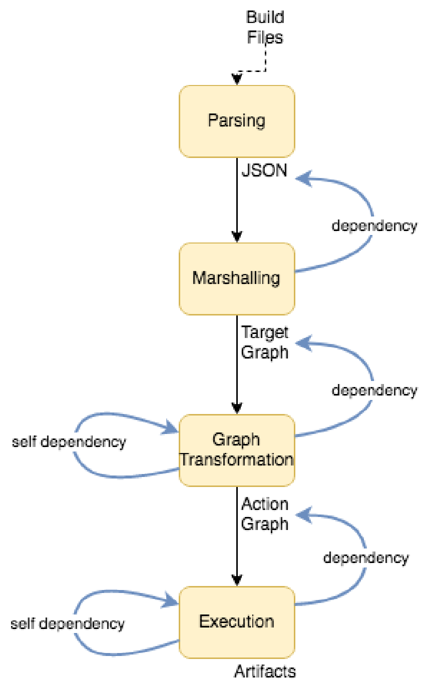

Figure 2. The computations of Buck and its dependencies.

3.3 Implementing the Incremental Computation Engine

One natural way of modeling the incremental computation engine is using the async/await feature [9] popular in many programming languages, so a call waiting for a dependency can cheaply block on it. Using the language Rust, we can encode a computation mapping keys to values as:

```
trait SomeComputation {
  async fn compute(k : ComputationKey) ->
      ComputationResult;
}
```

For example, the Graph Transformation phase computation can be roughly modeled as §3.2:

```
trait GraphTransformationComputation {
  async fn compute(target: BuildTarget) ->
      BuildRule {
    let target_node = MarshalingComputation.
        compute(target).await;
    let dependencies = target_node.deps.map(
      |dep| GraphTransformationComputation.
          compute(dep).await
    );

    return target_node.create_build_rule(
        dependencies);
  }
}
```

3.4 Implementing in Java

While a language featuring async/await would be ideal, Buck is implemented in Java, so async/await is not available. Therefore, we cannot implement our idealized computation API. However, we can implement a somewhat related variant:

The incremental computation engine itself has a simple API

```
interface ComputationEngine {
  Future<Result> compute(Key key);
}
```

The computations itself will implement the following interface:

```
interface Computation<Key, Result> {
  Set<ComputeKey> deps1(Key key);
  Set<ComputeKey> deps2(Key key, Environment
       deps);
  Result compute(Key key, Environment deps);
}
```

Instead of calling a single `compute` method, the code driving the `Computation` performs the following steps:

1. First call `deps1` with a key to get an initial set of dependencies the key statically depends upon.
2. Next call `deps2` passing in all the computed values requested by `deps1` in the `Environment` type.
3. Finally call `compute` passing all the values requested by both `deps1` and `deps2`.

The three method implementation is less safe than the original, as the implementation must ensure it only requests things from the environment it has requested in advance. Instead of having one phase of dependencies (as per applicative), or multiple phases (as per monadic), we get exactly two phases – which still has the same power as a monadic system. We can show the correspondence by writing an interpreter for the free dependency monad from §7.1 of [6]:

```
data Action k v = Finish v
                | Depend k (v -> Action k v)

type Key = Action k v
type Val = v

dep1 :: Key -> [Key]
dep1 (Finish v) = []
dep1 (Depend k f) = [resolve k]

dep2 :: Key -> [Val] -> [Key]
dep2 (Finish v) [] = []
dep2 (Depend k f) [v] = [f v]

compute :: Key -> [Val] -> Val
compute (Finish v) [] = v
compute (Depend k f) [_, v] = v
```

Here we have implemented an interpreter for the general monadic `Action` type using the three method abstraction, showing this formulation is equally powerful. Importantly, we have had to change the key type to contain additional information, namely the closure of what steps to perform next. To retrieve the closure to start, we assume there is a function `resolve`, based on `Tasks` from [7].

4 Learnings

Writing Buck using a monadic build system (i.e. the generic incremental computation engine) yielded a lot of benefits in terms of code complexity, and performance, and made it possible to consider future extensions to the Buck API.

4.1 Code and Performance

Before writing Buck as generic computations, the various phases were implemented as separate components with their own parallelism and caching. In the new model, the computation engine manages all caching and scheduling of work, leaving the computations themselves to only implement the core logic. A large amount of code that was previously repeated has been centralized, offering three compelling benefits:

1. Certain areas of code are substantially simpler, reducing lines of code in some areas by as much as 70%.
2. While the code is simpler it is also more correct. We eliminated several hard to track down deadlock and concurrency bugs since all computations now rely on a standard and well-tested execution engine.
3. We increased performance on some benchmarks by up to 3 times, by allowing us to focus optimization efforts on a single execution engine.

4.2 User-facing Monadic Buck?

By writing Buck on a monadic build system, we noticed that the applicative nature of Buck is mostly a consequence of limiting the build API. We could add an extra computation dependency to the rules like in Figure 3.

This change is equivalent to allowing the action graph creation to access the action execution, which means that the target graph to action graph computation now has access to the build results, thereby allowing actions to be created dynamically based on the results of executing other actions. The build dependency graph can then no longer be fully determined before any execution occurs, making Buck monadic from the users perspective. We could further improve user's dynamic power by adding another computation dependency from parsing to execution. Now, users could even dynamically alter parsing based on some rules execution.

Importantly, the core build system doesn't need to change at all, as it is already internally monadic. However, designing the appropriate APIs and retaining the graph query functionality may pose problems. As future work, we see value in investigating the feasibility to extend Buck in this direction, given the benefits offered by monadic build systems such as Shake [5, 8].

4.3 Java is not a good fit

Recall that the computation API from §3.4 had two stages of dependency discovery. This choice was forced because Java does not have async/await or coroutines. Futures were too heavy weight, so we introduced our own executors relying on multiple steps.

While having this API is powerful, the usability is sub-optimal. When multiple stages of dynamic dependencies are required, we are forced to split them into many computations where each has 2 stages of dynamic dependencies with lots of boiler plate code – it

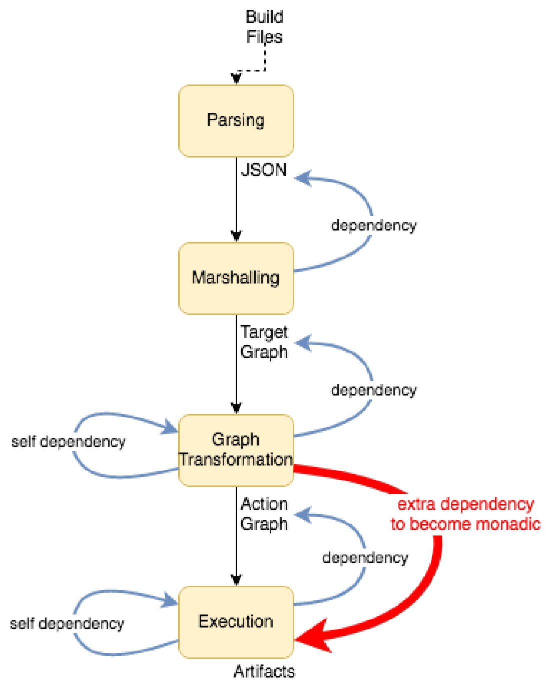

Figure 3. Computation dependencies of a Monadic Buck.

is quite error prone. We ended up introducing additional types of Computations to aid this type of pattern. We view this transformation as akin to manual lambda lifting [4].

Ideally, a high-performance monadic build system is best served in a language with co-routines and async/await.

5 Conclusion

In this paper we have shown that most applicative build systems are secretly monadic, all you have to do is look at them from the right perspective. Furthermore, this observation can be used to simplify the engineering of an applicative build system, irrespective of the power offered to users.

Once a build system is internally monadic, limiting users to an applicative API is a *choice* rather than an *engineering constraint*. With that flexibility, we hope to explore whether a more powerful API benefits the users, or harms users by reducing how analyzable a graph is. As we have seen with parsers, there are benefits to applicative parsers [10], and to those constrained to arrows [2], but there are still good reasons to sometimes use a monadic parser [3]. It may turn out that build systems need the same variety of approaches.

References

[1] 2013. Buck. (2013). https://buck.build/
[2] John Hughes. 2000. Generalising monads to arrows. *Science of computer programming* 37, 1-3 (2000), 67–111.
[3] Graham Hutton and Erik Meijer. 1998. Monadic Parsing in Haskell. *J. Funct. Program.* 8, 4 (July 1998), 437–444.
[4] Thomas Johnsson. 1985. Lambda lifting: transforming programs to recursive equations. In *Proc. FPCA '85*. Springer-Verlag, 190–203.
[5] Neil Mitchell. 2012. Shake before Building: Replacing Make with Haskell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. Association for Computing Machinery, New York, NY, USA, 55–66.
[6] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2020. Build Systems à la Carte: Theory and Practice. *Journal of Functional Programming* (2020). To appear in JFP.
[7] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build Systems à La Carte. *Proc. ACM Program. Lang.* 2, ICFP (September 2018), 79:1–79:29.
[8] Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. 2016. Non-recursive Make considered harmful: build systems at scale. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, 170–181.
[9] Semih Okur, David L Hartveld, Danny Dig, and Arie van Deursen. 2014. A study and toolkit for asynchronous programming in C#. In *Proceedings of the 36th International Conference on Software Engineering*. 1117–1127.
[10] Marcos Viera, S. Doaitse Swierstra, and Eelco Lempsink. 2008. Haskell, Do You Read Me? Constructing and Composing Efficient Top-down Parsers at Runtime. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 63–74.