Rules Tofu

Engineering Design Document

Jack Bradshaw

Sep 2025

Introduction

Open Tofu (hereafter just “Tofu”) is an open source ecosystem for Infrastructure as Code (IAC). It offers considerable benefits to platform engineers, and is a common choice for teams who require an open source tool (i.e. not Terraform). Integration into Bazel is presently limited, and the existing community-maintained ruleset (rules_tf) is only capable of linting and validation, with no support for compilation, external dependency management, and testing, meaning a hermetic and reliable E2E build is presently not possible with Bazel. Engineers must instead resort to manual build orchestration and other build tools, which introduces the possibility of human error and prohibits full Bazel monorepo development.

That simply will not do. This document contains the design for a robust and complete ruleset which allows user to:

- Import external Tofu dependencies into the workspace so they can be referenced as targets.
- Compile Tofu sources into libraries so they can be exported and referenced in other targets.
- Compile Tofu sources into binaries so they can be executed to provision infrastructure.
- Statically validate and lint Tofu sources to check for correctness and consistency.
- Running tests against Tofu sources so the contents can be verified and maintained.

Together these operations unlock the full potential of Bazel and IAC.

Background

This section provides background on Tofu for readers who are familiar with Bazel but not Tofu. It does not cover Bazel itself, and the Bazel docs are recommended prior reading. Furthermore, it covers areas of Tofu that are relevant to Bazel integration, and elides other details. For a broader understanding of the Tofu language itself, the Tofu docs as recommended. The information presented here is a synthesis of the official documentation and direct experimentation.

Overview

Tofu is an interpreted language for provisioning infrastructure. Four components are relevant to Bazel integration:

1. Tofu Providers, which are supporting utilities for interacting with specific providers.
2. Tofu Configurations, which are collections of infrastructure configuration files.
3. Tofu State, which is a snapshot of provisioned infrastructure.
4. Tofu CLI, which is a general purpose utility and interpreter for the ecosystem.
5. Tofu Dependency Management, which is the protocols and practices for resolving/downloading external dependencies.

Tofu Providers are pre-built binaries that modify infrastructure based on configurations. They are they executable programs which translate the desired state to actual provisioned infrastructure. Tofu Providers can be written in various programming languages and communicate with the Tofu CLI via a pre-defined interface.

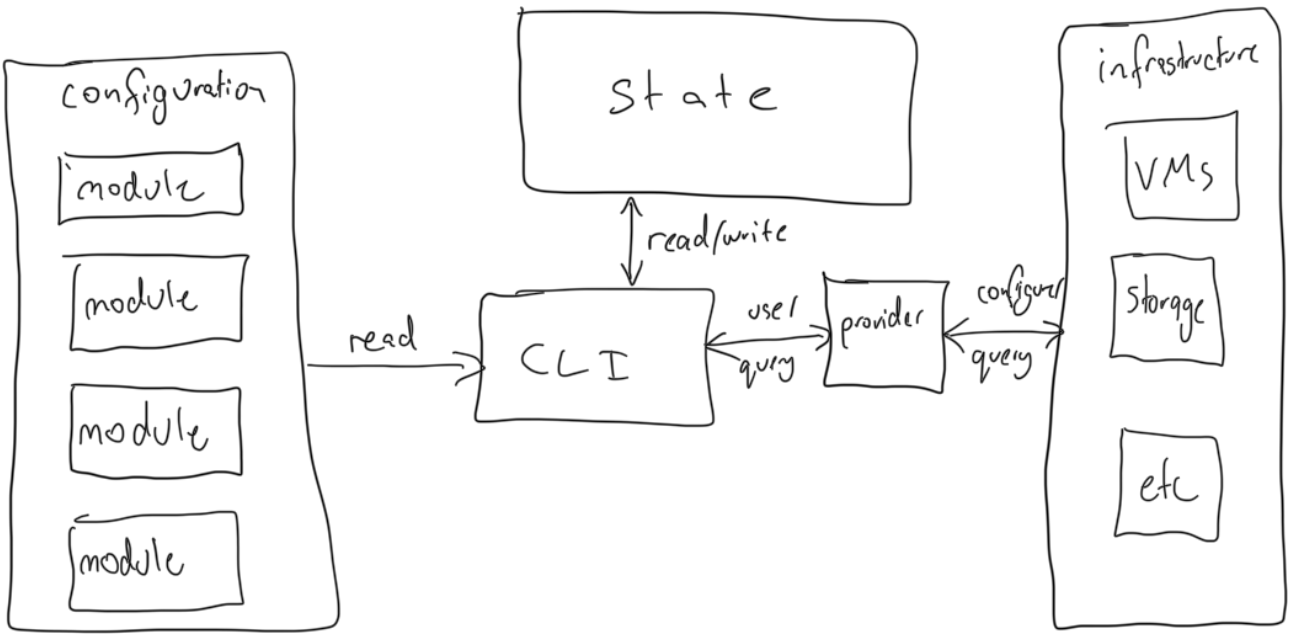
Tofu Configurations are collections of HCL source files and other data files which define the desired state of infrastructure. A configuration is effectively a declaration of the intended state, but not an imperative system for reaching that state. Tofu Configurations are structured into modules, which are reusable units of source code.

Tofu State describes infrastructure as it is currently provisioned. It's used by Tofu to track what has already been provisioned and effectively make changes when configurations change. For the most part, state is not a concern of the ruleset, and is considered a runtime implementation detail.

The Tofu CLI provides core functionality that all Tofu programs use, and is a critical component of the ruleset. It downloads external dependencies, validates and lints sources, and interprets sources to provision infrastructure. It is available from various sources including GitHub releases.

Tofu Dependency Management is the complex process of downloading modules/providers from remote sources for use in local programs. It has considerable complexity and occurs when Tofu programs are initialised (details of initialisation provided in the Tofu CLI section).

Together these areas explain how Tofu functions at a high level, and each is detailed below.



Tofu Providers

Providers are binaries that perform CRUD operations on infrastructure. High level details:

- Providers are not directly invoked by users, instead they are used in configurations, and Tofu orchestrates the literal calls based on the configuration.
- Providers are available for large cloud platforms (GCP, AWS, Azure), traditional hosting services (e.g. DigitalOcean), and various other miscellaneous services.
- Providers are precompiled but not necessarily statically linked, meaning they have no declared transitive dependencies but may require host libraries at runtime.
- Providers are distributes as sets, with one binary for each major OS/architecture.

Providers are usually sourced from remote locations. Details:

- Providers are identified by a namespace and a type. For example, the hashicorp/aws has the namespace “hashicorp” and the type “aws”.
- Providers can be downloaded from the public registry, which has the namespace “registry.opentofu.org”, or other registries (including private registries).
- Providers are versioned using the versioning constraint system.

Filesystem mirrors can be used to redirect lookups to local directories. Mirrors are declared in tofurc files and passed in to the Tofu CLI at runtime, for example, below is a `tofurc` file that redirects all provider lookups to `/dev/disk2/providers`:

```

provider_installation {

  filesystem_mirror {

    path = "/dev/disk2/providers"
```

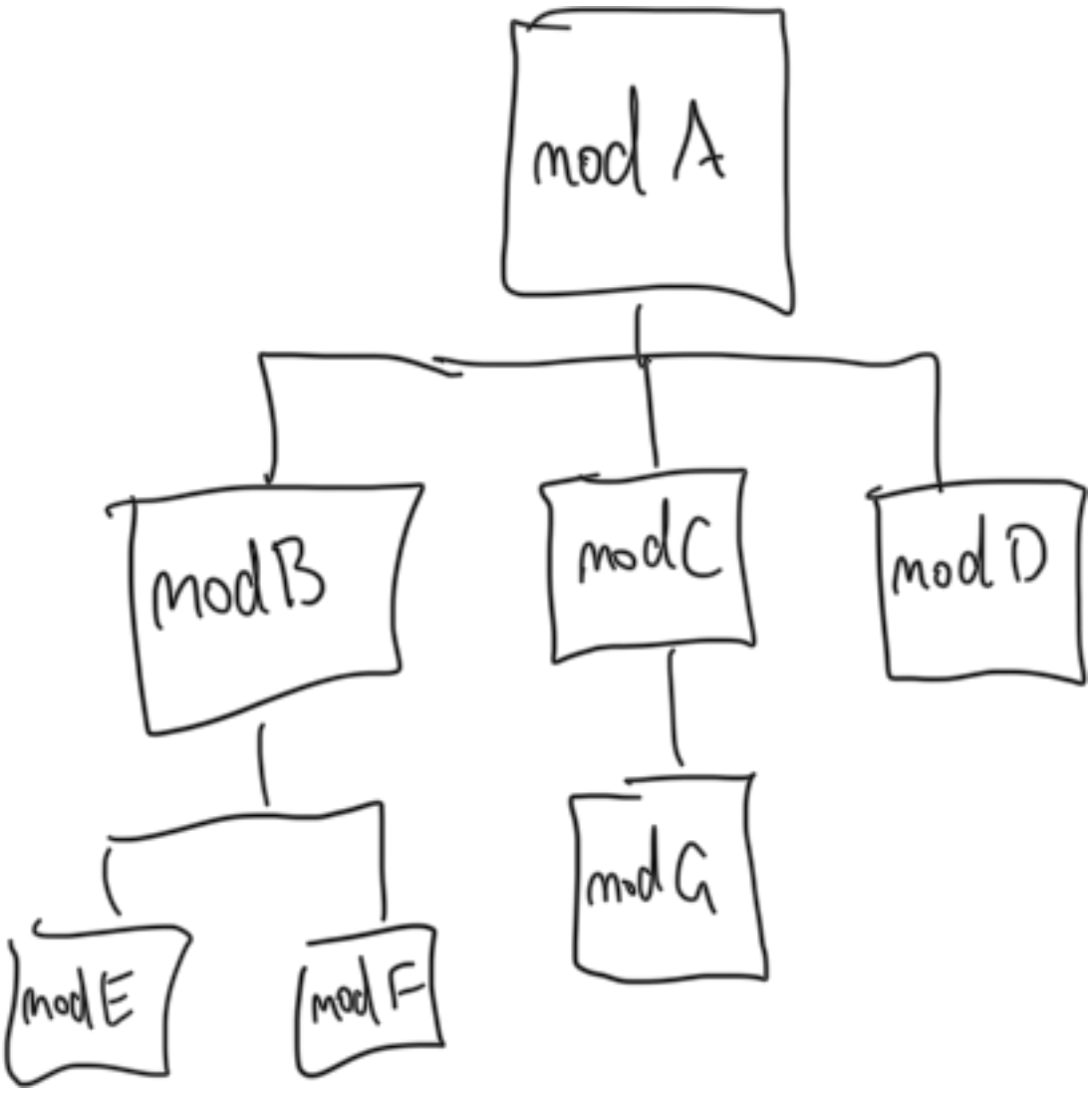
```
    include = ["*"]
  }
  direct {
    exclude = ["*"]
  }
}
```

The filesystem mirror directory must follow a strict format defined by the Tofu CLI. Each provider binary must be stored at $hostname/$namespace/$type/$version/$os_$arch, where the hostname matches the registry hostname, and the other values derive from the provider itself. The final binary file is named according to is os and architecture without an extension.

In summary, providers are binaries that directly interact with infrastructure, and Tofu uses them to provision/decomission infrstructure as directed by the user's configuration.

Tofu Configurations

Tofu configurations declare the desired state of infrastructure without specifying how that state should be reached. They are written in HCL, and since OpenTofu is interpreted, configurations are organised and distributed as directory trees of source files without an intermediate compilation step. Each directory in the source tree is a "module" in OpenTofu terminology, and when interpreted by the CLI, its contents are merged into a single logical unit (meaning file separation is for human readability not functional purposes). Every Tofu plan necessarily has a root module, which is defined as the directory where the Tofu CLI is invoked (i.e. not a property of modules themselves), or via a CLI argument.



Despite forming a directory tree, modules in subdirectories (i.e. submodules) are not automatically available to modules in parent directories, and modules must import each other to share functionality via a `module` blocks in their source files (known as module calls). There are two kinds of module calls:

- Local calls, which use relative path references to specify the directory containing the module. Paths are relative to the calling module. Local references are generally of the form ./dir1/dir2/dir3/etc but can also use up references (e.g. ../siblingdir).
- Remote calls, which use a variety of source addresses to reference modules in remote locations (i.e. http, git, s3 buckets, etc). Usually they follow the structure "$protocol$hostname/$namespace/$name/$provider", but $provider is just a string and does not imply a definite provider reference, and protocol may be elided for http. A version is usually also specified using the same the version constraint system as providers.

Below is an example of a local call and a remote call:

```

module "local" {

  source = "./modules/childmodule"
```

```
}

module “remote” {
  source = “terraform-google-modules/network/google”
  version = “>= 1.0.0”
```

Module sources and versions are not necessarily constant and can use variables, but there are limitations. The variables must be static, meaning they must be sourced from constants, CLI arguments, environmental variables, and other locations that are known during initialisation (explained in Tofu CLI below). Below is an example of a module that uses variables in the source address:

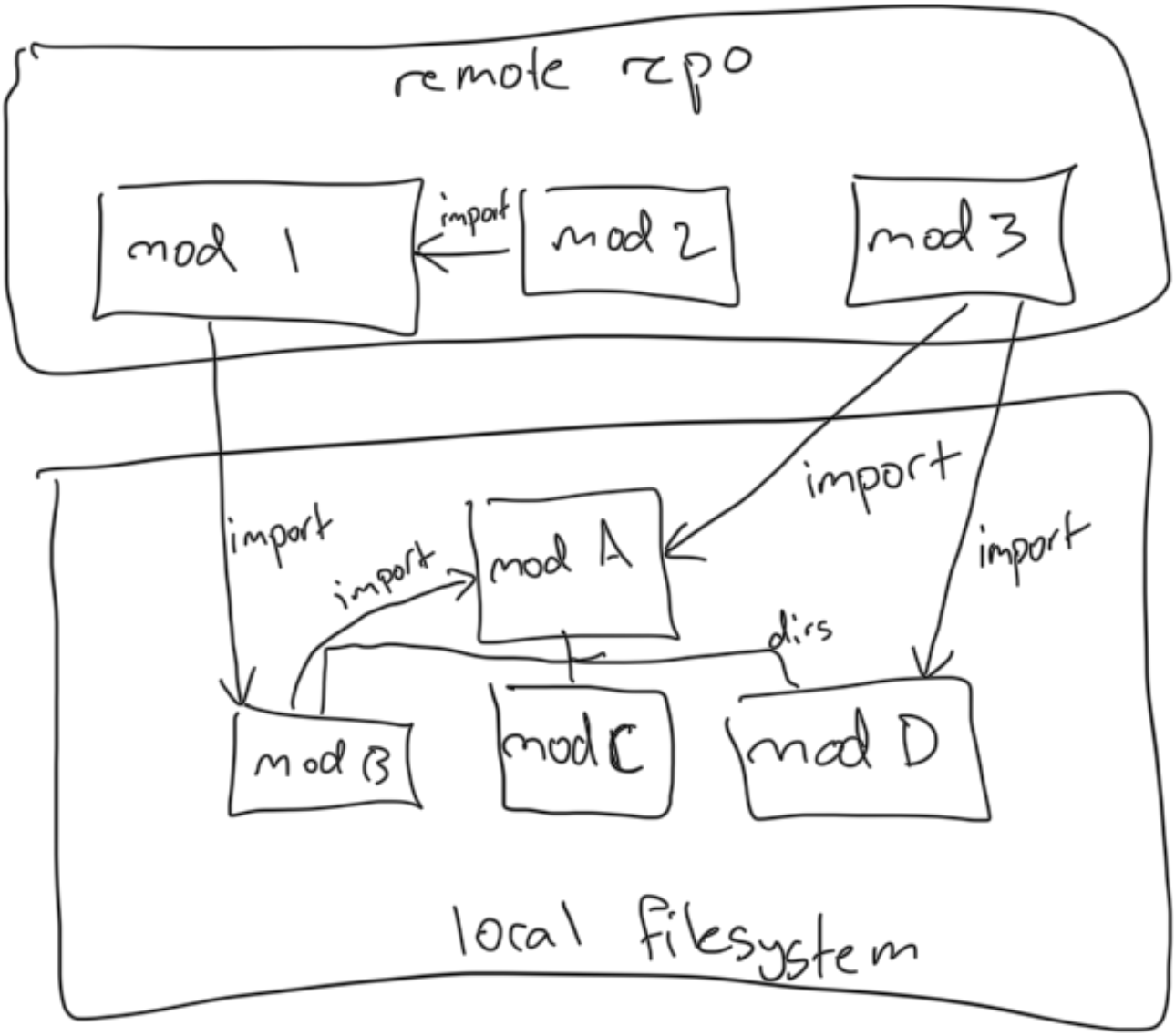
```
locals {
        modules_repo = "github.com/myorg/tofu-modules/"
        modules_version = "?ref=v1.20.4"
}
module "storage" {
  source = "${local.modules_repo}/storage${local.modules_version}"
}
module "compute" {
  source = "${local.modules_repo}/compute${local.modules_version}"
}
```

Modules may also use loops and conditionals to repeatedly call a module with different arguments, but loops and conditions cannot be used in the source and version attributes. Modules blocks must have unique names in the module they are defined in, but repeated blocks do not violate this rule because they necessarily have the same source and version. Below is an example of a module declaration that uses loops to instantiate two AWS modules, each with a different values for “name”, but note that name is a custom input defined by the AWS module, not a standard attribute of modules:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "4.0"
  for_each = toset(["hello", "world”])
  name     = "${each.key}"
}
```

```

Called modules may also call modules, therefore module dependencies create a directed acyclic graph. The image below illustrates an example with three remote modules (1, 2, 3) and four local modules (A, B, C, D), where B and C are subdirectories of A. Module 1 references module 2; module B references module 1; module A references modules B and 3; and module D references module 3. With this setup the transitive closure of A includes modules 1, 2, 3, and B, but not modules C or D.




In addition to importing other modules, modules may also declare dependencies on providers using the `required_provider` statement. Each declaration is versioned using the aforementioned versioning constraint system. Below is an example of a provider declaration:

```
required_providers {
  google = {
    source = "hashicorp/google"
      version = "6.0"
    }
}
```
```

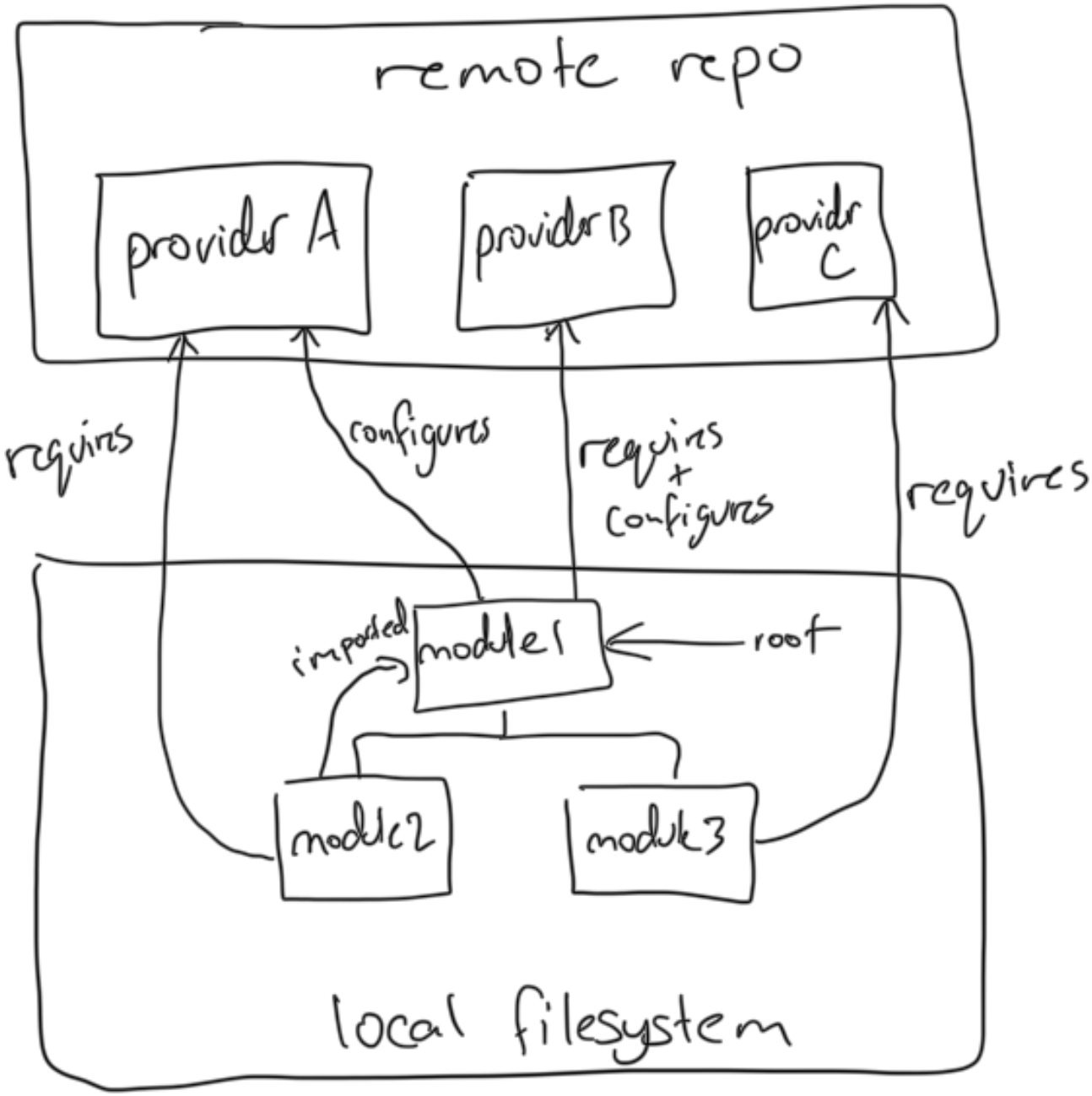
```

Any module may make a declaration, but only the root module may configuring providers using the `provider` statement, and an error is raised if a non-root module performs configuration. All required providers must be configured for successful execution, therefore the root module has to be somewhat aware of the entire transitive closure. Below is an example of a provider configuration:

```
provider "google" {
  project = "acme-app"
  region = "us-central1"
}
```

Providers do not have direct transitive dependencies (unlike modules) and therefore do not form a directed acyclic dependency graph. Nothing prevents providers from explicitly requiring dependencies on the host via dynamic linking or other runtime behaviours, but that is beyond the scope of Tofu and this document.

The image below illustrates an example with three local modules (1, 2, 3) and three providers (A, B, C), where module 1 is the root, module 1 imports module 2, module 1 requires provider B, module 2 requires provider A, and module 3 requires provider C. The transitive closure of providers for module 1 is A and B, therefore it configures both, despite only directly requiring B. Since module 1 does not import module 3, it cannot and does not configure provider C.
```



In addition to using modules and providers, modules may also read arbitrary files from disk using the file() and templatefile() functions, and file reads may use runtime variables to specify the file path. While this is not best practice, modules can use these functions to read files outside their directory, which has implications for Bazel integration. Any rearranging or renaming within a remotely sourced module could cause runtime failures, so the exact structure must be preserved. Reads outside the remote module's top-level directory are inherently fragile though and are virtually impossible to use without failure (both with and without Bazel).

Tofu State

Tofu is stateful by design and cannot function without state. It records the current infrastructure setup, and is used during provisioning for operations that cannot be performed idempotently. State is generally stored on a server for shared access and secret management (as state can contain secrets),

but it can also be stored locally in a file. State can be configured using the Tofu CLI, environmental variables, values in the configuration files themselves, and more.

Tofu CLI

The Tofu CLI is the general purpose utility in the Tofu ecosystem and enables several critical workflows. Filtering by those relevant to the ruleset:

1. Provisioning Infrastructure.
2. Validation Sources.
3. Running Tests.
4. Fetching dependencies.

All operations use the current working directory as the root module.

Provisioning Infrastructure

This process turns a Tofu configuration into provisioned infrastructure. It involves three commands:

1. Init, which initialises the working directory and downloads remote modules/providers.
2. Plan, which generates a sequence of operations to reach the desired configuration.
3. Apply, which runs the plan to provision infrastructure (may

Two important details:

– This function can be used without network so long as none of the sources it parses require downloading modules/providers from the network.

– Apply prompts the user for approval by default, but this can be disabled with the `-auto-approve` flag.

Validating Sources

The Tofu validate and fmt (format) commands are similar but distinct. Validate checks for syntactic and structural errors in a configuration, whereas format rewrites source files for consistency. Linting is implemented by a separate utility (tflint) and is beyond the scope of this document. Validation can only be run on root modules (and their references submodules), and is not available for isolated non-root modules.

Running Tests

The Tofu test command executes an instrumentation test on a module. It searches for tests, runs them, and prints the results to the console. It will automatically detect tests in the working directory and its immediate “tests” subdirectory, and a specific directory can be specified using the `-test-directory` arg. The working directory must be the module under test, but the module under test does not need to be a fully formed root module, meaning it can have un-configured providers.

Dependency Management

Resolving dependencies occurs as a transparent step during the provisioning infrastructure workflow, but is worth discussing in more detail due to its relevant to Bazel integration. Running `init` begins resolution of external dependencies using the `.terraform` directory in the current working directory. It searches there for modules/providers that satisfy the constraints in the configuration, and downloads new dependencies if required. It's involves a multi-stage process since dependencies themselves can have transitive dependencies. Within the .terraform directory are two subdirectories, modules and providers, and they diverge in how they are populated and structued.

Modules

The modules directory contains remote modules and a JSON file to record details of all the modules transitively used in a configuration. The modules directory is populated according to the following rules:

- A directory directly under modules exists for every remote module transitively called from the root (including remote modules called by remote modules).
- No directories exist for local module calls, including local calls in downloaded remote modules.
- The full contents of each remote module are downloaded, including non-source files such as LICENSEs, READMEs, and any modules that are never called.
- The name of each directory matches the name of the module block that called the module, with all transitive module calls prepended with dot separators.

For example on the last point, if the root module contains `module "foo" { source = "./bar" }` and the bar directory contains `module "baz" { source = "terraform-aws-modules/vpc/aws" }`, the name for the aws module is `foo.baz`. If the baz module were to call another module internally via `module "qux" { source = "terraform-google-modules/vpc/google" }`, the name for the google module would be `foo.baz.qux`. None of the module paths use `bar` because it's the name of a directory, not a module call block.

The contents of the JSON file are populated according to the following rules:

- The JSON file contain an entry for every module that is called relative to the root including local calls and remote calls. It does not contain entries for modules that are downloaded as part of another module but are not transitively called by the root (e.g. test modules in external modules). For example if the root calls module "foo" by path, module "foo" calls module "bar" by remote address, and module "bar" calls module "baz" by local reference within its downloaded files, the root module and all three other modules will have an entry.
- The JSON file entries are a flat list of tuples with four values: key, source, version, and dir.
  - Key is always the name of the module directory under .terraform/modules, and matches the name described in the module directory section.
  - Source is the local or remote address of the module and reflects the actual module call in the configuration file. Remote calls use the fully-qualified remote address (e.g. registry.opentofu.org/terraform-google-modules/network/google), and local calls use the local path (e.g. ./modules/firewall-rules). Values are statically resolved (i.e. no variables).

- Version is the version string for external modules, and is only present for remote addresses.
- Dir is the directory path of the called module relative to the root module, meaning it always begins with .terraform.

When using subdirectory syntax the system the above rules generally apply with some nuance:

- The entire module is still downloaded to the top-level `.terraform/modules` directory, and continues to be named according to the full module call stack (e.g. foo.baz.qux).
- The JSON version attribute is still populated as usual.
- The JSON source attribute is populated with the subdirectory syntax (e.g. registry.opentofu.org/hashicorp/consul/aws//modules/consul-cluster).
- The JSON dir attribute points at the subdirectory that directly corresponds to the subdirectory being referenced (not the top-level directory).
- References within the downloaded module are still resolved into other directories and JSON entries, including up-references to directories above the downloaded subdirectory.

There is no deduplication in this entire system, meaning identical calls to a remote module in the configuration will create multiple directories and JSON entries. For example of module `foo` and module `bar` both call the `terraform-google-modules/network/google` module, it will be downloaded into two separate directories in the .terraform/modules directory, and duplicate entries will be added to the JSON file.

In summary, the terraform module directory stores the remote modules, but no the local modules, and includes all sources for remote modules including non-Tofu source files. Each directory name derives from the call names in configurations, not the source addresses, and the JSON file contains a complete list of all modules transitively called from the root, with source details and directory lookups for each.

Providers

The providers directory follows a series of rules and is populated by the Tofu CLI, but the exact details are not important because filesystem mirrors avoid any need to manually manage its contents.

Versioning

Versioning for modules and providers uses version constraints which work according to the following rules:

- Version constraints are defined as a sign plus a version (e.g. = 5.0.0).
- Version constraints apply to modules and providers alike.
- Version constraints can use equals (=), not-equals (!=), general bounds (>, >=, <=, <), and limited bounds (~>). Limited bounds only allow the right-most value to autoincrement (i.e. ~> 1.0.1 can be upgraded to 1.0.2 but not 1.1.0).
- When resolving modules/providers, if a version in the .terraform directory already matches the constraints, it will be used regardless of whether it is the absolute latest version. If no match is found, the latest matching version is downloaded.

  - Version constraints can also be applied to tofu itself, which allows modules/providers to specify the version of tofu they are compatible with.
  - Multiple versions are not supported, and Tofu always attempts to resolve the latest possible version that satisfies all constraints.

Credentials

TODO need to work out how credentials work

Complexities

The Tofu ecosystem is difficult to integrate with Bazel for a variety of reasons:

- Tofu source files are interpreted not compiled, and there is no clear distinction between build-time and runtime
- Tofu does not have a native archive format for encapsulating libraries, and sources are distributed as directory trees.
- Tofu source files contain both primary source content and package manager directives (i.e. remote module source addresses).
- Remote source addresses can use variables.
- The structure of the dependency cache created by the Tofu CLI mirrors object names defined in source code, not absolute artefact names.
- Tofu natively resolves dependencies at runtime (due to the lack of build/runtime distinction).
- Tofu dependencies are downloaded using a variety of protocols and authentication schemes.
- Providers lookup can be redirected to the local disk with filesystem mirrors but no such system exists for modules.
- Version constraints are defined across the transitive dependency graph .

There are more complexities, but these capture the main points. Integrating Tofu into Bazel is a complex problem because they make it difficult to resolve dependencies at build time, package sources and dependencies together, use the packaged dependencies at runtime, and create a truly hermetic build.

Requirements

This section outlines granular requirements which translate the overall goal of Bazel-Tofu integration into more specific/actionable targets. All requirements are stated in the form of user stories to ensure each is connected to a tangible benefit and is a meaningful requirement. For ease of comprehension, requirements are divided between build-time, runtime, and other.

In the context of these requirements, an executable Tofu binary is defined as a self-contained executable which includes the Tofu CLI and single configuration, and when executed runs the

configuration with the CLI. The executable must package all relevant configurations, providers, and data files. This definition was selected to reduce ambiguity in the requirements and allow pragmatic design without strictly defining the solution.

The following requirements relate to dependency management. As a ruleset user:

1. I want to use remote modules, so I can compose existing code into my configurations.
2. I want to use remote providers, so I can use them in my configurations.
3. I want to use all natively supported source address types, so I have access to all native Tofu dependencies.
4. I want to pin external dependencies, so I know if upstream dependencies change.
5. I want to specify version constraints, so I can control which versions are resolved.

The following requirements relate to build execution. As a ruleset user:

6. I want to build reusable libraries, so I can use follow general engineering best practices.
7. I want to build hermetic binaries, so I can provision infrastructure securely.
8. I want to run tests on binaries, so I can ensure they are functionally correct.
9. I want to import precompiled libraries and binaries, so I can use supplementary dependencies.
10. I want to export libraries and binaries, so I can share my work with others.
    1. I want to export with all dependencies, so I can export self-contained artefacts.
    2. I want to export with no dependencies, so I can export focused artefacts.
    3. I want to export with local dependencies replaced with remote dependencies, so I can work in a monorepo and distribute to package managers.
11. I want to build programs of arbitrary scale, so my operation is not limited by the ruleset.
12. I want actionable and precise error messages, so I can debug effectively.
13. I want builds to be hermetic and reproducible, so I can use Bazel as intended by Google.
14. I want to execute builds on MacOS, Linux and Windows, so I can develop on virtually any machine.

The following requirements are specific to runtime of an executable Tofu binary. As a ruleset user:

15. I want to execute binaries on MacOS, Linux and Windows, so I can provision and interact with infrastructure on virtually any machine.
16. I want to perform any valid Tofu CLI function via the executable binary and have it apply to the packaged module, so I can perform advanced workflows.
17. I want the default workflow to be plan/apply with interaction prompts, so I can perform basic workflows easily and safely (i.e. avoid accidental infrastructure changes).
18. I want to pass through any valid Tofu CLI command to an executable Tofu binary and have it operate on the packaged Tofu module, so I can perform advanced infrastructure management.
19. I want actionable and precise error messages, so I can debug effectively.

The following requirements are general. As a ruleset user:

20. I want extensive documentation, so I can understand and use the tools with confidence.
21. I want to use Bzlmod, so I can use modern Bazel workflows and tools.

Regarding requirement 24, WORKSPACE support is not required, as it will be removed in the next version of Bazel (i.e. 9).

Solution

The ruleset allows Bazel users to reason about Tofu as a compiled language with external dependencies, compiled artefacts, and self-contained executables. It follows the conventions of popular rulesets such as rules_java and rules_jvm_external to create a familiar user experience with as much complexity handled for the user as possible. It can be considered from multiple angles, but is best understood by starting from external dependencies and working through to runtime (i.e. infrastructure provisioning).

External dependencies are downloaded at build-time and stored in external repositories as targets. A series of custom build rules consume those targets to produce library and binary targets that colocate dependencies with first party code. The resulting libraries can be distributed to private and public repositories, and the resulting binaries can be executed (using a custom runtime) to provision infrastructure without runtime dependency resolution. Instrumentation testing is available for binaries to ensure correctness before application to production.

This approach follows established Bazel conventions to create a user experience that is familiar and comfortable to Bazel users without sacrificing the broader Tofu ecosystem and community. The dependency management system allows users to manage external dependencies with discipline and control, and the build rules allow decomposition of Tofu configurations into logical units for composability, testability, and maintainability. With self-contained executables, provisioning becomes a one step process of running a Jar, which is particularly important for complex CI/CD workflows and integration with other programs. All of this combines to turn Bazel into a first-class engineering tool for Infrastructure as Code.

A full summary of the differences between Native Tofu and Bazel Tofu is shown below.

Aspect	Native Tofu	Bazel Tofu
Dependency Resolution Time	Runtime	Build-time
Dependencies Specification Location	In source files	In build-system files
Execution Model	Interpreted	Compiled (interpretation abstracted)
Library Artefact	Directory tree of sources	Single archive object
Executable Artefact	Directory tree of sources	Single executable object
Test Artefact	Directory tree of sources	No well-defined artefact, handled by Bazel infrastructure

Build Task	Navigate to source directory and validate sources with Tofu CLI (only available for root modules)	Run bazel build //foo:lib (available for root and non-root modules).
Run Task	Navigate to source directory and plan/apply sources with Tofu CLI	Run bazel run //foo:bin
Test Task	Navigate to source directory and plan/apply sources with Tofu CLI	Run bazel test //foo:test
Runtime	Tofu CLI installed on target machine	Custom program
Module Composition System	Reference other modules in sources via their local filesystem path or remote source address	Declare Bazel dependencies and reference in sources with their Bazel path.

This approach is made possible through a combination of custom components and standard Bazel integrations. A custom file format was created to hold libraries and executables, two custom programs were written implement the more complex parts of the system, custom rules which expose the functionality to the ruleset user, repository rules to fetch external dependencies, toolchains to integrate the custom programs into the rules, and a module extension to collect configuration information from Bazel modules across the workspace. All together they implement dependency management, build execution, and runtime execution, with many components being used at both build-time and runtime.

Below is a summary of all components:

1. FirmTofu, an archive format for Tofu libraries.
2. Vegan, a program which creates, manipulates, and executes FirmTofu archives.
3. Library Rule, a Bazel build rule which turns sources into libraries.
4. Binary Rule, a Bazel build macro which turns libraries into executables.
5. Test Rule, a Bazel build rule which runs tests on libraries.
6. Import Rule, a Bazel build rule which imports pre-built archives.
7. Export Rule, a Bazel build rule which exports libraries for external package managers.
8. Bundler Macro, a Bazel build macro to bundle a binary with the vegan runtime.
9. Tofu CLI Repository Rule, a Bazel repository rule to fetch the Tofu CLI.
10. Tofu Dependency Repository Rule, a bazel repository rule to fetch external Tofu dependencies.
11. Module Extension, a Bazel module extension for configuring the ruleset.
12. Toolchains, a series of Bazel toolchains to support the build rules/macros.

The following sections discuss the architecture and components in more detail.

Architecture

Architecture covers the broad strokes of the emergent systems and processes, and is divided into:

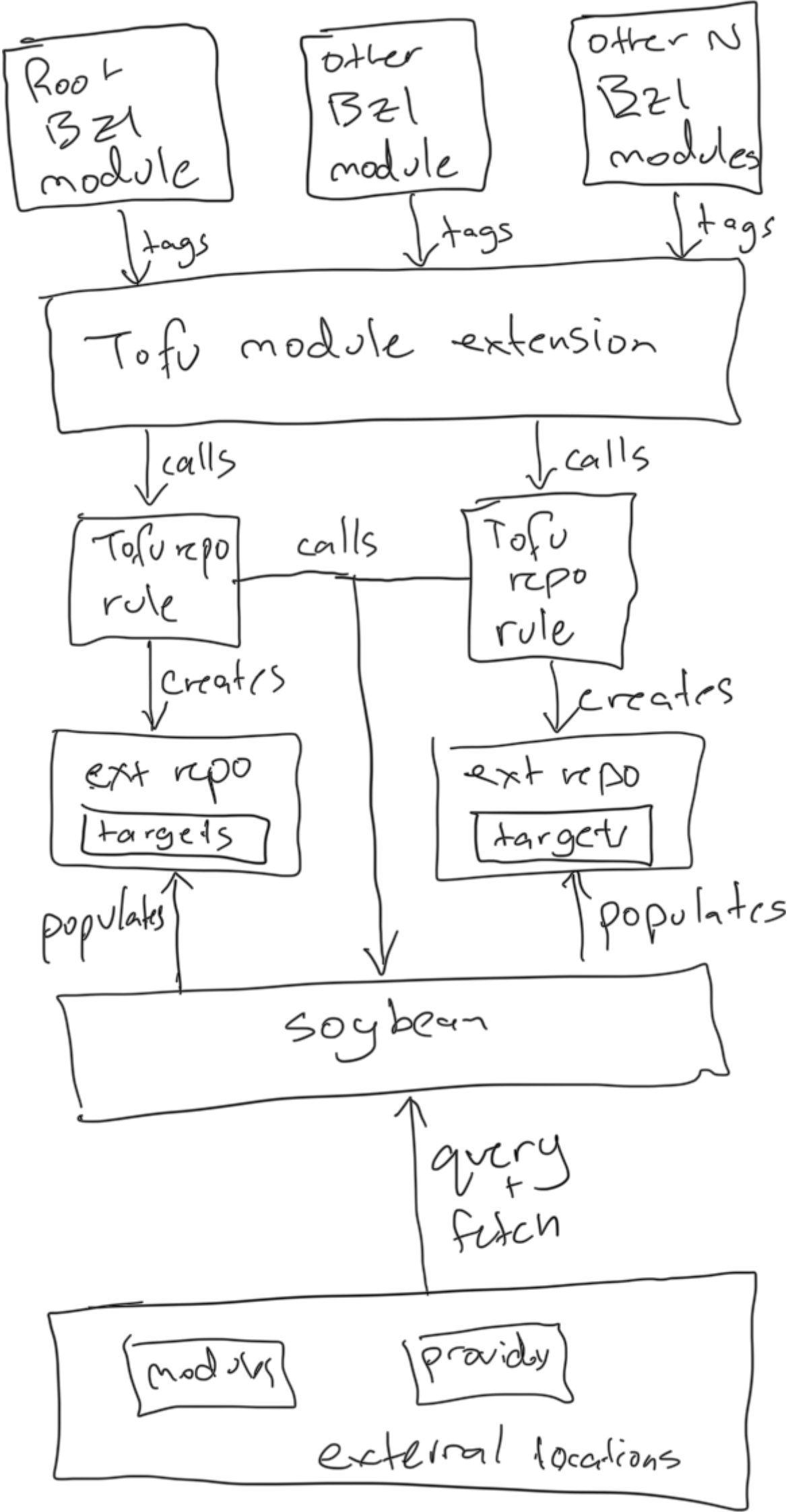
- Dependency Management, which covers the processes for resolving external dependencies.
- Program Composition, which covers the structure of Bazel Tofu programs and their compilation.
- Runtime Execution, which covers how binary and test artefacts are used.

References to the aforementioned components are used throughout to connect each to tangible implementation details.

Dependency Management

The ruleset downloads external dependencies into external repositories and exposes them as targets for use in the build. It uses soybean for the actual download operation, which internally delegates to the Tofu CLI for broad ecosystem compatibility. Dependency specifications are collected from across the Bazel module graph by the module extension and a series of tags, repository rules are used to provision the external repositories, and dependencies are stored in repositories as FirmTofu archives.

The full system integration diagram is shown below.


Root Bzl module
other Bzl module
other N Bzl modules
tags
tags
tags
Tofu module extension
calls
calls
Tofu repo rule
calls
Tofu repo rule
creates
creates
ext repo
targets
ext repo
targets
populates
populates
soybean
query + fetch
modules
provider
external locations

This approach is fairly standard for custom rulesets, and it provides several benefits that are crucial to secure and disciplined engineering. By making dependencies hermetic and inspectable it improves security and maintainability, and gives users an opportunity to intercept and modify dependencies in Bazel directly. By centralising dependencies in one location, it simplifies management and maintenance for users, and by decoupling the fetch operation from Bazel, improves maintainablity of the ruleset itself. Overall it creates a clean separation between dependency management and application logic, which is general engineering best practice.

Versioning System

Three options were considered for versioning:

1. Absolute single versioning, where only a single version of each external dependency may exist in the entire workspace (including Bazel module dependences).
2. Bazel module scoped single versioning, where each Bazel module may have a different version of each external dependency, but only one version within its own scope.
3. Segmented version spaces, where multiple versions of a module/provider may exist, each isolated into a seprate version spaces.

Although option 3 does not align with the general engineering best practice of single versioning, it was selected for pragmatic considerations. In Tofu, each module may declare various constraints for providers and other modules, and resolution fails if the collective constraint set is impossible to satisfy. This is acceptable for small/focused configurations outside Bazel, but when a single space needs to contain the dependencies for multiple unrelated configurations, impossible constraints are virtually guaranteed at scale. Option 3 resolves this issue by giving ruleset users a way to manage multiple version spaces manually, and puts the onus on the user to manage dependencies responsibly. For users who absolutely must have a single-version space, the root Bazel module has the option to enable absolute single versioning and module-scoped single versioning via module extension tags.

Note: The FirmTofu Archive format inherently supports multiple versions without issue because it uses content addressing. More details in the following sections.

Specification Syntax

External dependency specification uses the same syntax as native Tofu.

- Each provider is referenced as a hostname, a namespace, and a type (all strings). By default the public registry hostname is used for any missing hostnames.
- Each module is referenced as as hostname, a namespace, a name, and a provider (all strings).

Two limitations apply:

- External dependencies cannot be specified using subdirectory syntax for simplicity, but that does not preclude the use of subdirectory syntax in first party sources. It simply is not required during dependency resolution, and only top-level modules can be specified.
- Local references are not supported as external dependencies. To use local references they must be converted to first party build targets in the main repository.

Repository Structure

In earlier designs, Soybean would output files to the external repository directly for ease of inspection and modification, but this added significant complexity. Instead it outputs archives to the repository directly and the repository exposes them using the tofu_import rule. This reduces complexity but has the downside of making external dependency inspection harder. This tradeoff was accepted on the basis that ruleset users can still unpack and inspect archives if desired.

Download Agent

Soybean delegates to the Tofu CLI for the actual download operation which has four benefits compared to the build-in Bazel downloader:

1. It inherently supports all the custom protocols and authentication schemes of Tofu (by internally delegating to the Tofu CLI).
2. It ensures the version resolution logic matches the broader Tofu ecosystem without significant reimplementation.
3. It decouples the logic from Starlark and allows thorough testing using a high-level language.
4. It allows the download to be used outside Bazel.

In essence it avoids reinventing the wheel.

Program Composition

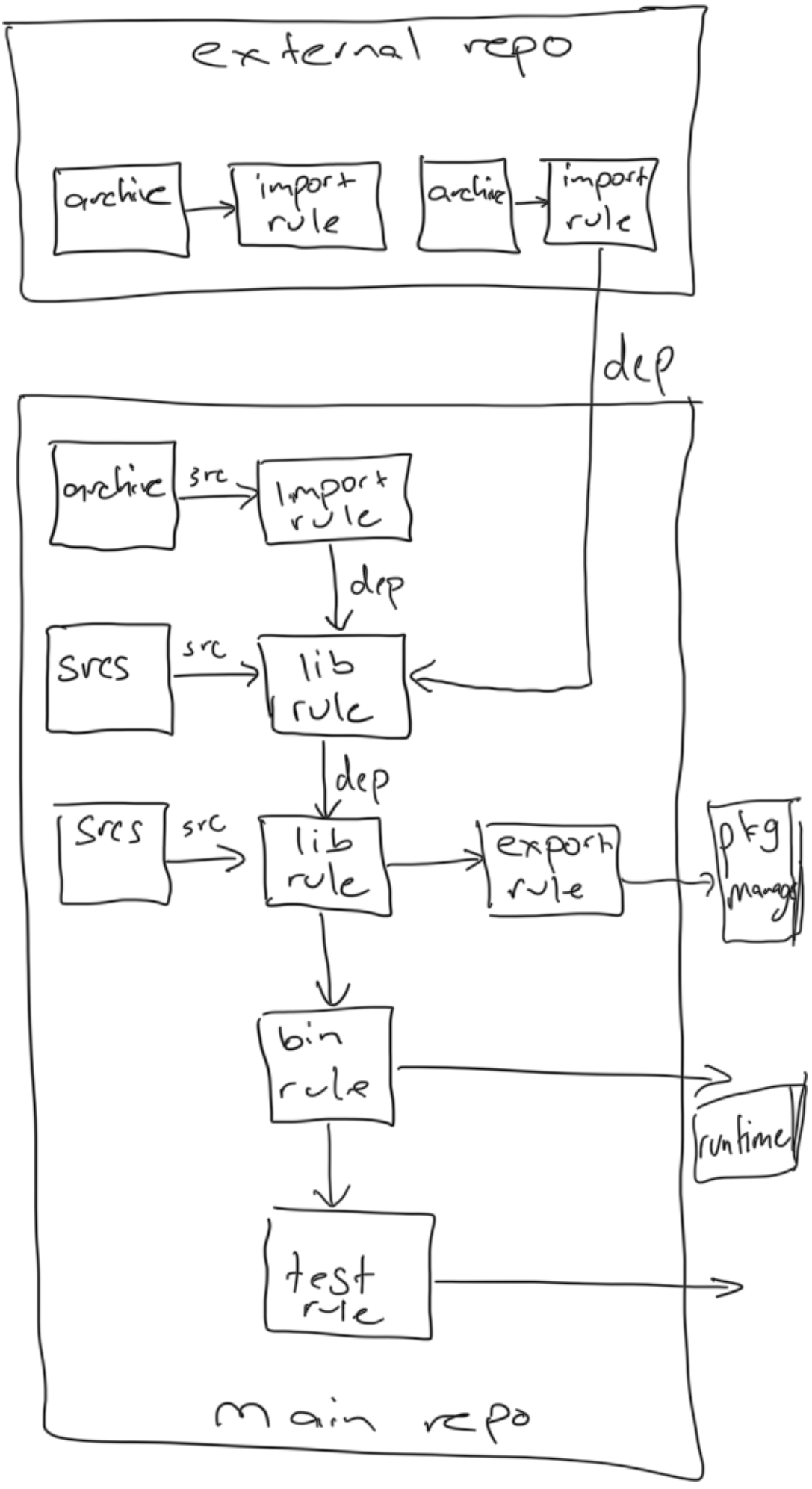
Bazel Tofu programs are composed of targets as usual, and there are six supported target types:

- Provider libraries, which are individual providers.
- Configuration libraries, which are reusable configurations without a root module.
- Binaries, which bundle a configuration with its dependencies and a root module specification.
- Instrumentation tests, which run Tofu tests (using the native test infrastructure).
- Exports, which exposes libraries in a format that is convenient for public release.
- Imports, which pulls precompiled archives into the target graph.

The supported dependencies edges are as follows:

1. Provider depend on other languages to create provider binaries.
2. Configuration libraries depend provider libraries and other configuration libraries.
3. Binaries depend on configuration libraries.
4. Instrumentation tests depend on binaries.
5. Exports depend on configuration libraries.
6. Imports have no dependencies.

The rules work together to compose complex Tofu programs. A system integration diagram is shown below.


external repo
archive
import rule
archive
import rule
dep
archive
src
import rule
dep
srcs
src
lib rule
dep
srcs
src
lib rule
export rule
pkg manager
bin rule
runtime
test rule
main repo

Reference System

At the target level, tofu targets use Bazel deps to depend on each other, and at the source-level, they use a variation of the standard Tofu reference system to import each other. References to external modules and providers use the standard external reference syntax without alteration, and references to other Bazel-defined modules use a custom bazel syntax.

For example, the external `hashicorp/consul/aws` module can be referenced as:

```
module “foo” {
  source = “ hashicorp/consul/aws”
  version = “1.0.0”
}
```

Sub-package references may also be used, for example:

```
module “foo” {
  source = “hashicorp/consul/aws//modules/consul-cluster”
  version = “1.0.0”
}
```

Subpackages must only use directory references after the // and may not include ref tags or other non-directory elements. The details of how these are handled are covered in the Soybean program.

Internal modules are referenced using the custom `bazel/$package/$target` syntax where the variables are underscore-deliminated package. For example, to reference the module defined in `//foo/bar:baz` use:

```
module “foo” {
  source = “bazel/foo_bar/baz”
}
```

Targets are expressly forbidden from including subdirectories in their sources (enforced by the rule), so there is no first party submodule reference system. Such modules should be exposed as separate targets instead. Package names do not include the bazel repository and are merged into a single global namespace. Conflicts are unlikely if users already use their own namespace for their directory structure.

This reference system exists for user convenience, and the labels are stripped at build-time and replaced with content addresses. This is part of the compilation process and is covered in more detail in the Soybean and FirmTofu archive sections.

Quick notes:
1. some modules might be defined in 1P and used directly but need to be stripped before release because they are also released to 3p. For those cases, the id for each module/provider needs to be either an external reference or an internal reference. The default is internal references, but external references can be overridden with an attribute. Both can be referenced in source without issue, it just affects how they get packed into the archive. In terms of version spacing, modules defined in 1P with external references effectively create a new version space. This is necessary to prevent conflicts with a previous version of the module released externally being used as an external dep.

Version Space Constraints

The multiple-version space decision detailed in dependency management was necessary to work around the constraints of the tofu ecosystem, but could lead to unsustainable target graphs if dependencies from different version spaces are mixed. By default, targets do not allow version spaces to be mixed, and will fail if deps are from different spaces are used. Each version space is a separate external repository, so this could be otherwise stated as, targets do not allow dependencies from multiple external repositories to be mixed by default, and ruleset users must explicitly override this.

Each target can set the `allow_all_version_spaces` attribute or the `version_space_allowlist` attribute to override this default behavior. The former is a boolean (default false) that allows all version spaces to be mixed, and the latter is a list that specifies the version spaces that can be mixed. Supplying both attributes is an error.

These version space constraints carry across the dependency graph transitively and affect which targets can depend on each other. A target cannot depend on a target that uses version spaces outside its own allowed version space, which is implemented by passing version space information trough the Bazel provider objects returned by the custom rules.

For example, library target A uses a closed versioning space and only pulls dependencies from one external repo. It requires no custom attributes and uses the default behaviour. Target B uses an open version spaces. It declares `allow_all_version_spaces `as `true` and uses dependencies from external repositories. Target C uses a constrained but open version space draws. It uses the `version_space_allowlist` and declares the space used by target A plus an additional space. In this setup:

- Target A could not depend on targets B or C because their version spaces are broader than its own.
- Target B could depend on either of target A and C, because their version spaces are included in its own.
- Target C could depend on target A because its version space is included in its own, but not target B because its version space is broader than its own.

For convenience and debugging, build flags are available to override all target graph attributes and set the version space behaviour of the entire workspace. Some engineers may wish to use this flag to enable strict version space separation across the repository as a matter of discipline regardless of individual target values. This flag could cause build issues though because it applies across the entire bazl module graph and other modules might not function correctly, so is only recommended for debugging.

This system works around the constraints of the Tofu ecosystem and is a necessary safety precaution. It gives ruleset uses a way to use mutually incompatible dependencies within their workspace while segregating them across the target graph. It defaults to the safest option, which isolates dependencies, but gives engineers the power to make structured exceptions.

Two implementation details: Version spaces are specified in rule attributes as external repository labels, but encoded as a SHA256 hash of the dependency specifications, concatenated in alphabetical order. This allows imports, which cannot be directly tied to a version space, to be mixed with other targets.

Validation

Tofu configurations can only be validated (checked for correctness) if they contain a root module, but libraries do not contain one by definition. As such, a library can only be validated by adding it to a binary or a test and building/testing respectively. This is a general tradeoff of the Tofu ecosystem. A workaround was considered, where libraries could define a validation target attribute to supply a fake root for use in validation, but it was unergonimic (polluted build files) and produces as much code as defining a binary.

Runfiles

Runfiles are handled using the built-in file function of Tofu and no Bazel-specific runfile handling is provided. The data attribute on library targets can be used to include files in a module, and those files can be retrieved at runtime using `file(${path.module}/filename)`. A more robust runfile system was considered, similar to the runfiles mechanism in rules_java, but this approach is the simplest and is entirely compatible with export to native Tofu.

Post-Build Operations

Artefacts produced by the build-process can be used in a variety of ways. Library artefacts can be transformed into native Tofu for export to third party package managers, binary artefacts can be executed to provision infrastructure, and test artefacts can be run to evaluate correctness (note that test artefacts are binary artefacts with test code). Various other workflows are available but these are the main three. All workflows involve passing a FirmTofu archive to the vegan runtime after the build has completed. By default, the `bazel run` function can be applied to a binary target to execute the provisioning workflow, and the `bazel test` function can be applied to a test target to execute the testing workflow.

Further details are covered in the vegan section. Note: Vegan contains the Tofu CLI internally, so the host machine can use vegan without installing Tofu directly.

Components

The components provide the tangible implementation for the outlined architecuture, and several are complex enough to require further detail before implementation can proceed. Details are provided for the FirmTofu archive, the Vegan program, the various build rules, the helper macro, the repository rules, and the module extension. Toolchains are a standard part of Bazel, and no further details are necessary, except to say the Tofu CLI toolchain uses a version downloaded from GitHub (i.e. not bundled with the ruleset).

FirmTofu

The FirmTofu archive format stores compiled Tofu programs. It uses zip as the underlying format for broad OS/tooling compatibility, and its contents are divided into three top-level structures:

1. Modules, a directory containing the modules of the program.
2. Providers, a directory containing the providers of the program.
3. Manifest, a protobuf binary wire format file containing metadata about the archive and the program.

The modules/providers directories contain the compiled modules/providers respectively. Each module/provider is stored in a separate subdirectory (under the appropriate top-level directory), and each is accompanied by a metadata file (a Google protobuf version 3 binary wire format file). The subdirectories and the metadata files are named using a hashing algorithm that creates a content addressable space. Metadata files may exist without corresponding content directories, for cases where only metadata is required, but content directories cannot exist without corresponding metadata files.

This structure creates content addressable storage for modules/providers, which ensures reproducibility across builds, meaning the same set of modules and providers will always produce the exact same archive. Furthermore, it decouples the archive structure from native Tofu, so that upstream

changes to native Tofu do not necessarily invalidate existing archives and can be supported by changing the TofuPress program (critical for long-term sustainability).

A critical function of the metadata files is recording the relationships between modules and providers. These details are required at build-time and runtime for performance optimisation and correct operation. Each metadata file contains both the forward links (i.e. items this item references) and the backward links (i.e. items which reference this item). The forward links are retained so content addressable storage can be transformed back to a nested directory tree at runtime (recall from background section that all files must be placed in their original location), and the backward links ensures performant archive modification when removing modules/providers (discussed below).

A few design choices worth noting:

- Google Protobuf (version 3) was chosen over XML and JSON to ensure files are space-efficient, inherently reproducible, and can be checked against a predefined schema.
- A previous design iteration stored the metadata for each module/provider in the manifest itself, but this was not ideal, as it burdened a single file with the entire contents of the archive and required reading the entire file into memory before making changes.
- A previous design iteration explored further breaking down modules/providers directories into content addressable storage so each file had a unique address. While this may have offered further deduplication across modules, it incurred significant duplication in metadata, and was not pursued. This avoids building a general purpose content addressable storage solution (not the goal) and keeps the format optimised for Tofu programs (modules/providers are the primary unit of reuse).
- While the Bazel ruleset is designed for single-versioning by default, the archive supports multiple versions of modules/providers, which is critical for cases where a single-version cannot satisfy all transitive dependencies.

Overall, this system stores modules and providers in a way that scales to large programs, is resilient to change in the broader ecosystem, and supports the arbitrary file reference system of Tofu programs.

Modules Directory

The modules directory contains the modules of a Tofu program. Each module is stored as a metadata file and an optional subdirectory containing its content. Each subdirectory may contain any number of arbitrary files, but never nested directories.

Below is an example of a modules directory containing three modules, only two of which have content.

/modules

e5a0b54f2c8d2a6a0c8e4f6b2d1c0e8a7f6d4c2b0a9e8d7c6b5a4f3e2d1c0b9a.pb

3a9850a2665812933f52423193494b923985b428234857342938472394872349.pb

f0e1d2c3b4a5968778695a4b3c2d1e0f1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d.pb

/e5a0b54f2c8d2a6a0c8e4f6b2d1c0e8a7f6d4c2b0a9e8d7c6b5a4f3e2d1c0b9a

main.tf

README.md

LICENSE

vars.tf

/3a9850a2665812933f52423193494b923985b428234857342938472394872349

main.tf

vars.tf

Each metadata file contains a single Module message which stores the content address of the corresponding module content directory, external source information (for external modules only), and various references to providers and other modules (as proto file content addresses). A reference is included for every direct submodule, direct supermodule, and direct provider usage, but references are not included for transitive supermodules and submodules as they can be resolved transitively. Submodule references include both the content address of the referenced module and the name the module knows the submodule by (for reconstructing the original directory tree exactly). Supermodule references do not include the name, as that information can be retrieved from the equivalent submodule reference in the supermodule's metadata file. The proto is approximately:

```
message Module {
  string content_id = 0;
  ExternalSourceInformation external_source_information = 1;
  repeated SubmoduleReference referenced_submodule = 2;
  repeated string referenced_supermodule_address = 3;
  repeated string referenced_provider_address = 4;

  message SubmoduleReference {
    string module_address = 0;
    string directory_name = 1;
  }
}
```

Important caveat: External source information is only present for the top-level module of externally sourced modules, meaning their submodules (i.e. subdirectories) lack any external source information, and transitively linked to their source using the reference system. This avoids unnecessary duplication of metadata and allows deduplication when identical submodules are used across multiple modules.

Note: During the compilation process, module/provider references within module source files (.tf, .tofu, .tf.json, .tofu.json) are updated to point at content address. While they must be transformed again at runtime, this intermediate state minimises the coupling between the archive format and native Tofu, which aids long-term maintainability and has other subtle benefits for the overall architecture. This detail relates more to the compilation process itself, but it does characterise the archive format. For

example, a source file containing `required_providers { aws = { source = "hashicorp/aws" version = "~> 5.20" } }`, would be updated to the form `required_providers { aws = { source = "9c03b11f9b7c85e28a9b3d3b7f191a6c4b2de046559385b2e31e7f193c7d0d69" } }` during compilation.

Note: This structure allows two modules that contain the same content but have different directory names to be compressed into a single archive. The directory name difference is captured by the submodule reference message type and can be reconstructed from this information.

Providers Directory

The providers directory contains the providers of a Tofu program. It contains a proto file for each provider, and when present, a subdirectory for the provider content. The subdirectories always contain a binary for each of the supported platforms, each named $os_$arch (no file extension), and they never contain nested directories. Below is an example of a providers directory containing three modules, only two of which have content.

/providers

717d8822bb04b2706e41b8cd25f28d4cf041789f9ca69630fb93a0955092d18d.pb

609f901d9e144657e523e6b7f516a1f8d0887ecc55d26e463489cfac9d659ae2.pb

1f2e3d4c5b6a79889f0e1d2c3b4a567890a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5.pb

/717d8822bb04b2706e41b8cd25f28d4cf041789f9ca69630fb93a0955092d18d

linux_amd64

linux_arm

linux_arm64

darwin_amd64

darwin_arm64

windows_amd64

/609f901d9e144657e523e6b7f516a1f8d0887ecc55d26e463489cfac9d659ae2

linux_amd64

linux_arm

linux_arm64

darwin_amd64

darwin_arm64

windows_amd64

Each metadata file contains a single Provider message, which stores external source information (for external providers only) and references to the modules which use the provider in their `required_providers` blocks. References are not included for modules which transitively use the provider as they can be resolved transitively at runtime. The proto is approximately:

```
```

```
message Provider {
  string module_reference_address = 0;
}
```

Manifest

The manifest file contains a single Manifest message. It records details about the archive itself and details about the Tofu program that cannot be captured in the individual module/provider metadata files. The proto is approximately:

```
message Manifest {
  Archive archive = 0;
  Configuration program = 1;
}
```

The Metadata message contains information about the archive itself, specifically the version of the archive spec that was used when creating the archive. This provides a degree of future proofing if the spec needs to change to support native Tofu changes. It begins at zero and does not use semver because it's unlike to change frequently. The proto is approximately:

```
message Archive {
   int version = 0;
  }
```

The Configuration message contains information about the configuration contained in the archive, specifically the root module and source information about each module/provider (excluding submodules of external modules). The root field is singular which inherently prevents multiple roots. The proto is approximately:

```
  message Configuration {
    string root_module_address = 0;
    map<VersionedSourceInformation, string> external_modules = 1;
    map<VersionedSourceInformation, string> external_providers = 2;
    map<UnversionedSourceInformation, string> local_modules = 3;
    map<UnversionedSourceInformation, string> local_providers = 4;
```

```
  message VersionedSourceInformation {
    string fully_qualified_reference = 0;
    string version = 1;
  }

  message UnversionedSourceInformation {
    string fully_qualified_reference = 0;
    string version = 1;
  }
}
```

The remote module/provider source information messages mirror the Tofu ecosystem equivalents.

Note: In previous design iterations, equivalent XML and JSON manifests were included for human readability, but this was not pursuit to avoid the possibility of manifest divergence (and archive corruption). As protobuf is not human readable, a function was added to the TofuPress program to export the manifest to XML and JSON.

Hashing

Given a directory containing a module/provider, the content address is calculated by hashing the directory contents (but not the directory name). The hash is calculated by transforming the directory into a single contiguous byte stream and calculating the SHA256 digest of the entire stream, with alphabetical ordering of the files. All file names use UTF-8 encoding, and file contents are processed without modification. Each hash is encoded in the standard 64-character lowercase hexadecimal representation.

For example, given a directory (foo) with two files, bar.txt and baz.gcl, the byte stream is effectively “bar.txt” + <contents of bar.txt> + “baz.gcl” + <contents of baz.gcl>, and the final hash is of the form f4a5c3b2e1d0f9a8b7c6d5e4f3a2b1c0d9e8f7a6b5c4d3e2f1a0b9c8d7e6f5a4. This hashing algorithm assumes modules/provider content directories do not have subdirectories, which is a general condition for the archive structure.

Critical detail: Content directories names and metadata file names both use the same hash, meaning metadata files do not use their own hash, but rather the hash of the directory they correspond to. This details breaks from the traditional definition of content addressable storage but is critical to the effective function of the archive. It allows archives to be merged and reduced based exclusively on the contents of the modules/providers, it allows metadata files to exist without content directories, and it ensures cross-cutting details such as references do not affect the address space.

Performance

Performance can be understood in terms of four core operations:

- Creation, which is instantiating a new archive.

- Merging, which is turning multiple archives into one.
- Reduction, which is removing content from an archive.
- Export, which is restoring the original directory tree structure.

These operations can be decomposed into:

1. Instantiating the archive structures (creation).
2. Retrieving modules/providers from content addressable storage (merging/reduction/export).
3. Inserting modules/providers into content addressable storage (creation/merging).
4. Removing modules/providers from content addressable storage (reduction).
5. Reconstituting the original directory tree structure (export).

These operations are implemented by the various programs and in practice their runtime performance depends on their implementation, but the archive itself imposes the following hard limits:

- Instantiation involves creating empty directories and an empty manifest, which is a constant time operation due to the lack of parameterisation.
- Retrieval from content addressable storage involves reading the contents of the file/directory that corresponds to the hash of the contents, which is a constant time operation with respect to the number of modules/providers in the archive, and linear time with respect to the number of items in the module/provider.
- Inserting into constant addressable storage involves calculating the hash of the module/provider, creating/populating a directory with its contents, retrieving the proto file for each referenced module, and adding new references. This is constant time with respect to the number of modules/providers in the archive, linear time with respect to the number of items in the module/provider to add, and linear with respect to the number of references to update.
- Removal is the reverse process of insertion and the performance is identical inperformance.

In practice, removal operations are not singular, meaning removing one module/provider also means removing any which are now transitively disconnected as well. Practical removal therefore requires traversal of the reference graph and is a linear operation with respect to the total number of transitively connected modules/providers (since each must be visited to check whether it too can be removed) and constant with respect to the total number of modules/providers in the archive. Furthermore, practical removal often involves removing all external modules/providers, which can be performed in linear time with respect to the number of module/providers to be removed, due to the presence of the external modules/providers lists in the root manifets. Without these references it would otherwise be a linear operation with respect to the size of the archive (due to the need to iterate over each module/provider and check its source information).

These performance limits are made possibly by the combined use of content addressable storage and the reference system. The former allows content retrieval/insertion/removal to scale independent of the total number of modules/providers in the archive, and is limited only by the performance of the hashing operation itself and the need to update references. By keeping both forward and backward references in the metadata files, referenced content can be discovered in constant time (for a module/provider known by content address), meaning the performance of reference updates is limited

only by the total number of references to update, not the size of the archive. These properties are critical for scale at build and runtime.

Note: This performance evaluation assumes the archive has either been fully loaded into memory or unzipped to disk, but if neither are true, then a load/unzip operation must occur first, which is a linear time operation with respect to the number of modules/providers in the archive. This can sometimes be avoided in practice, but not always, and puts a hard limit on theoretical performance. Future work may examine a different file structure which permits random access without preloading.

Emergent Properties

Any given archive has the following properties:

- Correctness, whether it complies with the format described in this document.
- Completeness, whether it contains all transitive dependencies for all included modules.
- Runnability, whether it contains a main module.
- Minimalism, whether it contains only the files it requires for execution.

An archive is correct only when:

- The top-level modules directory exists (may be empty).
- The top-level providers directory exists (may be empty).
- The top-level modules directory does not contain any subdirectories beyond the first level of subdirectories for each module.
- The top-level providers directory does not contain any subdirectories beyond the first level of subdirectories for each module.
- Each module/provider subdirectory is named as the hash of its contents according to the hash algorithm.
- Each module/provider subdirectory has an equivalent metadata file (the is no requirement that metadata files have equivalent directories though).
- Each provider subdirectory contains a binary for every supported platform.
- The manifest exists in the top-level and is a Google Protobuf (version 3).
- The manifest can be parsed into the Manifest protobuf message without error.
- No extraneous files exist in the archive.
- The root module has no supermodule references in its metadata file.

The presence of extraneous files and directories is always incorrect to avoid supporting arbitrary files during merge/reduce operations, as these could easily cause conflicts or be handled incorrectly. Any file in the top-level directory which is not the manifest is extraneous. Any file in the modules directory or content directory which is not a content addressed proto file or a child of a content addresed directory is extraneous. Any file in a provider content directory which is not one of the expected binaries is extraneous. Any nested directory within a module/provider content directory is extraneous,

as are any files within them. In essence, only the core archive files/directories and the flat list of content items may exist, and all other files are extraneous (and therefore incorrect).

An archive is complete iff it contains the entire transitive closure of modules and providers for its contents (i.e. every module has every other module it requires and every provider it uses). This requires both content and metadata for references modules (i.e. it is insufficient to have only metadata files for referenced modules). An archive which is not complete is necessarily incomplete, but incomplete modules are not necessarily incorrect, and are in fact useful for build-time for performance optimisation.

An archive is runnable if the manifest specifies a root module, and a module which is not runnable is necessarily non-runnable.

An archive is minimal if it contains only the modules/providers that are transitively required by the root module. An archive which is not minimal is necessarily overloaded, and a module which is non-runnable is only minimal if empty.

Vegan

Vegan is a general purpose program for working with compiled Tofu. It performs build-time and a runtime functions, and is written in Java for cross-platform execution. It exposes the following functions:

1. Fetch, which takes a list of dependencies and produces an archive containing the transitive closure of each (all in the same version space).
2. Pack Module, which takes a set of dependencies and a set of module sources and produces an archive containing the modules.
3. Pack Provider, which takes a set of provider binaries and produces an archive containing the provider.
4. Pack Metadata, which takes metadata and produces an archive containing the metadata.
5. Unpack, which takes an archive and unpacks it to the local directory (effectively unzip).
6. Repack, which takes a previous unpacked archive and turns it back into an archive.
7. Export, which takes an archive and produces a native Tofu configuration.
8. Make Root, which takes an archive and sets a contained module to the root module
9. Modify Metadata, which takes an archive and adjusts the metadata of its contents
10. Check Integrity, which takes an archive and determines whether it is correct
11. Merge, which takes a series of archives and produces one merged archive
12. Reduce, which takes an archive and removes content
13. Run, which takes an archive and executes the provisioning workflow on it
14. Test, which takes an archive and executes the testing workflow on it
15. Validate, which takes an archive and checks the contents for completeness and correctness
16. Query, which takes an archive and returns metadata about it

17. Translate Manifest, which takes an archive and exports the manifest in various forms
18. Version, which returns the version of the program

A few rules which apply across all functions unless otherwise stated:

- When selecting a module/provider from an archive, the specification can be a remote addresses (for external modules), a bazel specification (for first party modules), or a content addresses (for both).
- When temporary directories are used for processing, the temporary directory is always deleted before the function returns. This can be disabled with a flag.
- All operations which reference file IO (e.g. zipping, unzipping, writing files) are implemented using an in-memory virtual FS. The exception is the unpack operation which intentionally creates a directory on disk. This aids performance by keeping data in-memory and avoids the need for manual disk cleanup.

Various rules require the Tofu CLI to operate. It does not need to be installed on the host and is bundled with vegan for simplicity. When required, it's unpacked to a temporary directory and used by vegan, then deleted when finished.

Fetch

Fetch downloads remote dependencies and turns them into archives. It accepts remote dependency specifications (as remote addresses), and returns an archive for each. Every archive exists in the same version space, but each contains the transitive closure for a single dependency only.

The high level process for turning module dependencies into archives is:

1. Create a dummy HCL file that depends on each module/provider (with a well known name for each module).
2. Run `tofu init` to download all modules and deps in the .terraform directory.
3. Turn the contents of the .terraform directory into a series of archives.

Steps 1 and 2 are trivial, but step 3 involves more nuance. After step 2 the .terraform directory will contain the transitive closure of the requested dependencies. Inserting its contents into an archive involves two steps:

1. Update module/provider references to use content references.
2. Insert the contents into content addressable storage and populate the metadata/manifest files.

Essentially, after step 2, all modules that are transitively connected to the dummy HCL will have been reformatted from:

module “somename” {

  source = “soemvalue”

}

to

module “some” {

```
  source = “1343147ab252c…”
}
```

Modules that were downloaded but never referenced will not be updated, which is acceptable for the ruleset, because they will never be instantiated or referenced.

Updating references is a recursive problem, because the content address for a module can only be calculated after all its references have been replaced with content addresses for referenced modules. This is effectively a depth-first graph traversal, and fortunately the modules.json file contains enough information for the graph traversal. It does require passing down a module name stack since the keys in the json file are the full module call stack. A recursive algorithm for updating all modules is approximately:

/** File is a directory containing the module, callStack is the list of module calls to get to the module from the root, and manifest is the modules.json file provided by Tofu (or some structure equivalent to it).

```
fun replaceReferencesWithContentHash(module: File, callStack: List<String>, manifest: Manifest) {
  for (file in module.files()) {
    val modules = readModules(file)
    for (module in modules) {
      val fullStack = callStack + “.” + listOf(module.name)
      val referencedDirectory = manifest.getDirectory(fullStack)
      replaceReferencesWithContentHash(referencedDirectory, fullStack, manifest)
      module.setSource(hashOf(referencedDirectory))
    }
  }
}
```

After this has completed recursive execution, all modules calls that are transitively connected to the root module will have had their modules replaced with content hashes of their referenced content. The remaining work involves effectively creating an archive for each of the original dependencies. Each archive must contain the transitive closure of references modules and providers, without including other modules/providers from the version space. Each external module has a top-level directory in .terraform/modules, so the process is effectively selecting those directories, then reading their contents to determine which providers they reference. The algorithm is effectively, for each top-level module dependency:

1. Iterate over the modules.json file and select every entry where the key begins with its well-known name. This selects all modules that are transitively referenced by a call beginning in the module.
2. Filter out any entries that do not have a version tag to remove local references. This filters out its internal local calls.
3. Select the directory attribute of all remaining entries. These directories contain the modules to include.
4. Insert all selected directories into content addressable storage.
5. Iterate over all selected directories, inspect required provider attributes using HCL edit, and select the providers to include. Insert them into content addressable storage.

After this, an archive will exist for every requested provider containing only the provider, and an archive will exist for every requested module, containing the transitive closure of the module (modules and providers included).

Two caveats about this approach:

- It will fail if there are cycles in the dependency graph.
- It will work for any modules which internally use package-syntax references, because the entire top-level module is downloaded to the .terraform directory, and the references in the modules.json file allow the directly referenced package to be found.they create top-level module directories in .terraform and the directory containing the referenced module exists in JSON. The content address replacement step will still point them at content addresses, and the final top-level insertion will put the entire external module in storage.

Pack Module

Pack module turns module sources into archives. It accepts a set of Tofu source files and data files, a set of dependencies (as archives), and a name for the module (using the bazel syntax), and returns an archive containing the compiled module and metadata. All input files are treated as belonging to the same module regardless of location or nesting, and whether dependencies are packed into the archive is configurable with a flag. The implementation is effectively:

1. Iterate over all source files and find references to remote modules/providers. Lookup each reference in the dependencies and replace them with content addresses. Fail if one is not found.
2. Insert the files into content addressable storage.
3. Create a metadata file for the sources that includes references to all dependent modules.

4. Create metadata files for all referenced modules but do not include content.
5. Create a manifest that includes all modules/providers in the archive.

Pack Provider

Pack Provider turns a provider into an archive. It accepts a set of binaries (one for each supported platform), the source information for the provider (either remote address or bazel syntax), and creates a single archive containing the provider. The implementation effectively creates a zip that contains the binaries in content addressable storage, a metadata file for the content, and a manifest file. Since providers do not have dependencies and nothing else is being packed, the metadata and manifest values are minimal and only need to include the source information (no references).

Unpack

Unpack exports archives to the local disk without destroying information and is effectively an unzip operation. It accepts the archive to unpack and performs the operation. Unpack is not to be confused with export.

Repack

Repack undoes the unpack operation. It accepts the location on disk containing the unzipped archive.

Export

Export converts archives to the local disk in a format compatible with native Tofu, and is a transformative operation that cannot be undone (i.e. an archive cannot be created from an exported archive in one step). Export accepts a flag which determines how external dependencies are handled, with one option to link them locally, and one option to use their remote addresses. When the former is used, the packed dependencies are unpacked and linked into the configuration by updating sources in place, and when the latter is used, the external dependencies are not unpacked and they are linked by inserting their original remote addresses into the source in place. The former is ideal for execution, and the latter is ideal for deployment to external package managers.

The implementation for unpacking with local linking is effectively:

1. Create a providers directory and move all providers from the archive to the directory. Use the Tofu packed format so each can be referenced in a filesystem mirror.
2. Create a modules directory and move all non-root modules to the directory. Each top-level remote module is a top-level directory and each first party module is a top-level directory. Other modules (i.e. submodules of remote modules) are placed in the correct location under top-level directories based on the reference information in the archive metadata.
3. Export the root module to the target directory.
4. Iterate over all modules and use HCL edit to replace all module references with local file references and all provider references with the original remote addresses. These details exist in the archive. It's an error if a content address is used in a module but the archive has nothing to replace it with and export fails.
5. Create a tofurc file in the target directory. It must define the provider directory as a provider filesystem mirror.

The implementation for unpacking with remote linking is effectively the same process as above except remote modules and remote providers are not written to disk, and remote module references are updated to their original remote address instead of a local address. No filesystem mirror tofurc is created either. Note: Subpackage references are detected as content address references that point to directories within external modules and are updated to the // syntax,

A previous iteration of the design exported modules and providers to the .terraform directory but this solution was fragile because it relies heavily on the structure of the directory, and required more information to be passed through at build-time.

Make Root

Root sets the root module in an existing archive. It accepts the archive to root, the module to set as root (as a spec), and creates an equivalent archive with the module set as the root. The implementation is effectively:

1. Unzip the archive to a working directory.
2. Find the module in content addressable storage. Fail if it does not exist.
3. Update the root manifest to specify the root module by its content address.
4. Zip the working directory and return the archive.

The default behaviour in step 2 can be disable by passing a flag to allow the root to be missing. It's provided for edge cases and is not used by the ruleset.

Modify Source

Modify source sets the source information of a module/provider. It accepts the module/provider to edit and the new source information for it. It returns a new archive with the updates and does not modify the original archive.

Modify References

Modify references sets the reference information of a module/provider. It accepts the module/provider to edit and a new set of references for it. It returns a new archive with the updates and does not modify the new archives. It's unnecessary to specify back references, as these are updated automatically.

Check Integrity

Check integrity validates the structure of the archive (not the program itself). It accepts an archive and returns a pass/fail. An archive only passes if it meets the definition of correct specified in the FirmTofu archive section.

Merge

Merge combines multiple archives into a single archive. It accepts the archives to merge and produces a single archive containing their content. All modules/providers are combined into a single version space, the root manifests are merged, and the metadata files are merged. The structure of the archive ensures these operations are simple set unions without conflict, meaning module/provider content can

simply be inserted into a combined space, reference lists in metadata files and the root manifest can be concatenated. A few edge cases and properties to mention:

1. Archives from multiple version spaces can be merged without warning because archives do not encode version space constraints from build-time.
2. If more than one archive contains a root module, the operation fails.
3. When metadata references are merged, links are updated in both directions, to ensure the graph remains bidirectional.

By default no integrity checks are performed and archives are assumed to be correct. An optional flag can be passed to run an integrity check on each archive before merging.

Reduce

Reduce removes content from an archive. It accepts an archive and three flags to control its behaviour, and returns a new archive containing a subset of modules/providers.

The first flag controls what modules to target in the retain/remove algorithm. The options are

1. Retain specified modules and remove everything else.
2. Remove specified modules and retain everything else.
3. Retain specified modules and their transitive parent/child modules.
4. Remove specified modules and their transitive parent/child modules.
5. Retain specified modules and their transitive modules (parent/child and calls).
6. Remove specified modules and their transitive modules (parent/child and calls).
7. Retain first party modules and remove everything else.
8. Remove first party modules and retain everything else.
9. Retain first party modules and their transitive modules (parent/child and calls).
10. Remove first party modules and their transitive modules (parent/child and calls).

The second flag controls what providers to remove. The options are:

1. Retain specific providers and remove everything else.
2. Remove specific providers and retain everything else.
3. Retain providers referenced by modules in content.
4. Retain providers referenced by modules in content and modules in transitive references.

The third flag controls what to do with metadata after content has been removed. The options are:

1. Retain metadata for retained content only.
2. Retain metadata for retained content and directly referenced external content (i.e. not transitively referenced content).
3. Retain metadata for retained content and transitively referenced external content.
4. Retain all metadata (i.e. remove nothing).

The flags operate in order, meaning modules are reduced, then providers are reduced, then metadata is reduced. The implementation is complex as it must account for all of the options, but each is effectively a graph traversal using an unexplored set and an explored set, where the references in the metadata files define the edges, and the references in the root manifest identify first party and external components. When retaining metadata based on references, only references to external content can be used, and there is no option to retain references to internal content, as external content cannot reference internal content by design. Overall these options cover all the potential use cases in common (and unlikely) scenarios.

Note: Integrity checks will not pass for some reduced archives. This is by design. Archives that no longer pass integrity checks cannot safely be merged into other archives.

Provision

Run provisions infrastructure. It accepts an archive containing the infrastructure configuration, runs the provisioning workflow on its contents, and outputs a pass/fail status to stdout. The implementation is effectively:

1. Check if the archive has a root module using the query function. Fail if not.
2. Export the module to a temporary directory using the export function.
3. Set the working directory to the temporary directory.
4. Run the Tofu init, and pass the results to stdout.
5. Check the contents of .terraform/modules to ensure nothing has been downloaded. Fail if new modules have been downloaded.
6. Run Tofu plan and print the result to stdout.
7. Prompt the user for permission to continue.
    1. If they deny, delete the temporary directory, and exit.
8. Run Tofu apply functions, and pass the result to stdout.
9. Delete the temporary directory.

Steps 5 and 7 provide safety but can interfere with CI/CD. Flags can be supplied to skip them.

Note: The TF_CLI_CONFIG_FILE environmental variable is used to reference the tofurc file generated by the export function.

Test

Test dynamically validates an archive. It accepts the archive to test, runs a test, and outputs a pass/fail status to stdout. The implementation is effectively the same as provision, except it uses the `tofu test` command instead of `tofu init/plan/apply`. A check is still performed to ensure no modules are downloaded.

Validate

Validate statically validate an archive. It accepts the archive to validate, performs validation, and outputs a pass/fail status to stdout. The implementation is exactly the same as test, except it uses the `tofu validate` command instead of the `tofu test` command.

Query

Query exists to make the contents of the archive observable. It accepts an archive and various flags to customise the query. It provides the following query options:

1. Get the source information for module/provider (specified as content address).
2. Get the content address for a module/provider (specified as content address).
3. Get the references of a module (specified as content address).
4. Get the references of a provider (specified as content address).
5. Get a list of all external source addresses for all modules.
6. Get a list of all external source addresses for all providers.
7. Get a list of all content addresses for all modules.
8. Get a list of all content addresses for all modules.
9. Get the version of the archive.
10. Get the root manifest as JSON.
11. Get the root manifest as XML.
12. Get the emergent properties of the archive (listed in FirmTofu archive section).

Version

Version exists to provides information about the vegan program itself. It accepts no arguments and prints the version of vegan to stdout.

Library Rule

The library build rule (tofu_library) is analogous to java_library. It accepts sources and dependencies, and compiles them into a composable library object (FirmTofu) and a depset.

Create compilable tofu libraries that can be composed and validated automatically, with support for both root modules and non-root modules.

```
tofu_library(
  name,
  srcs,
  deps,
  is_root,
  validation_root,
```

) -> FirmTofu

Library rule prevents files from subdirectories because that does not work with the module structure of tofu. Easy runtime check in implementation.

When a library targets is built, the output is a lightweight archive for performance

Library will only allow .tofu .tf .tofu.json and .tf.json files, and everything else can do in data. In practice both sets will be merged together into the contents of the library, but this separation is consistent with the bazel approach to sources and runtime data being declared separately.

All rules require deps to be non-file targets for consistency with broader bazel conventions.

Note: The Tofu Library rule builds a reusable component for use as a dependency in other targets, and the underlying data structures are optimised for performance by minimising the artefact contents and relying on providers for the . The Tofu Export rule builds a reusable component for deployment to external package managers, and the underlying data strutures

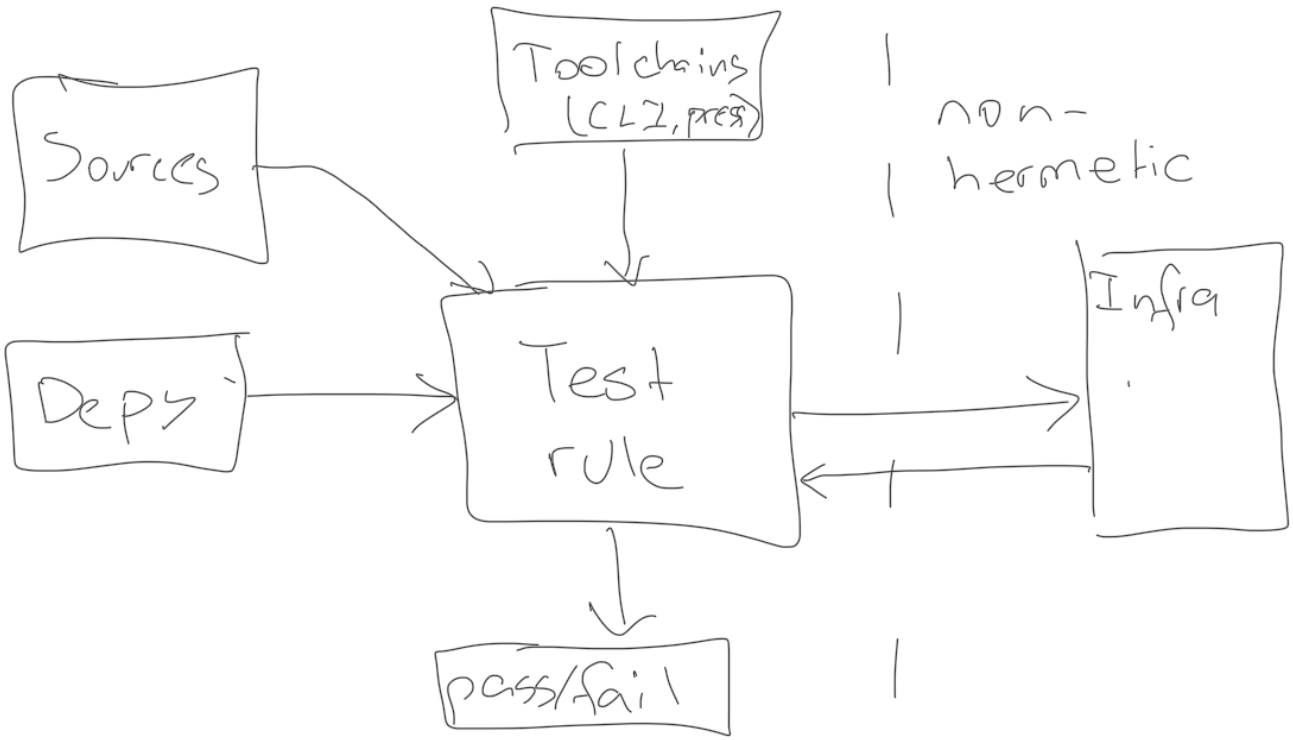
Binary Rule

Create executable tofu programs that can be run to provision infrastructure.

```
tofu_binary(
  name,
  deps,
) -> BakedTofu
```

Test Rule

The test rule (tofu_instrumentation_test) does not have a direct Java analogue because it is an instrumented test not a unit test. This limitation emerges from the ecosystem which does not have a concept of local testing and is entirely reliant on provisioning real infrastructure to determine program correctness. The rule simplifies testing by accepting the library under test as a dependency and using the specified sources to setup the text fixtures (e.g. set variables, perform checks, etc). It works well when libraries follow general engineering best practices such as avoiding global/static data so tests con specify their behavior with input variables.



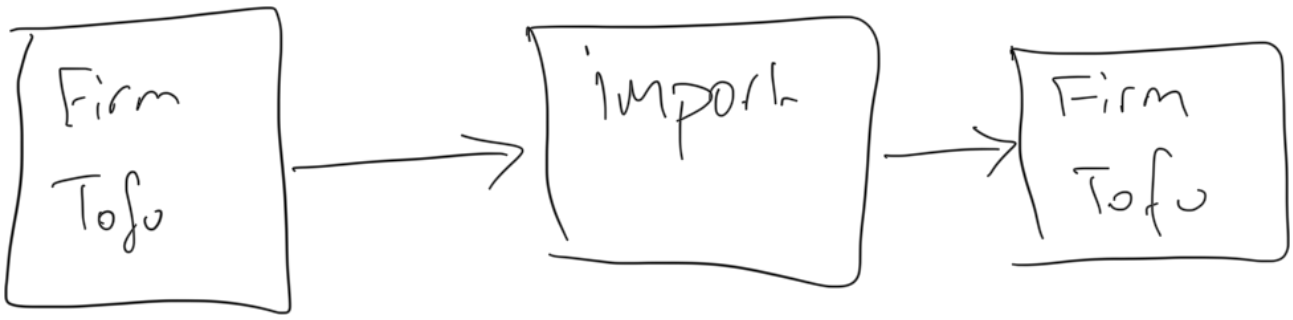

```
tofu_test(
  name,
  deps,
  srcs,
  testfile,
)
```

Uses the Tofu CLI test command internally.

Since the test interacts with processes outside Bazel (infrastructure) it is non-hermetic. Bazel must be made aware of this with the `no-cache` tag. Furthermore, the `requires-network` must be included to allow the test access to the network (for connecting to remote infra). Together these tags configure Bazel to allow network access and re-run the test whenever requested (as opposed to returning the cached result).

Import Rule

The import rule (tofu_import) is analogous to java_import. It accepts a FirmTofu file and makes it available to the build graph. It’s functionally the simplest of all the build rules and is effectively just a pass through to integrate pre-compiled archives.



Export Rule

The export rule (tofu_export) is analogous to java_export. It accepts sources and dependencies, and compiles them into a composable library object (FirmTofu). Unlike the library rule, the output includes all transitive dependencies, and is well suited to release to external package managers.

Export modules for publication.

```
tofu_export(
  name,
  deps,
  exclude_external_modules,
) -> FirmTofu with external deps potentially removed
```

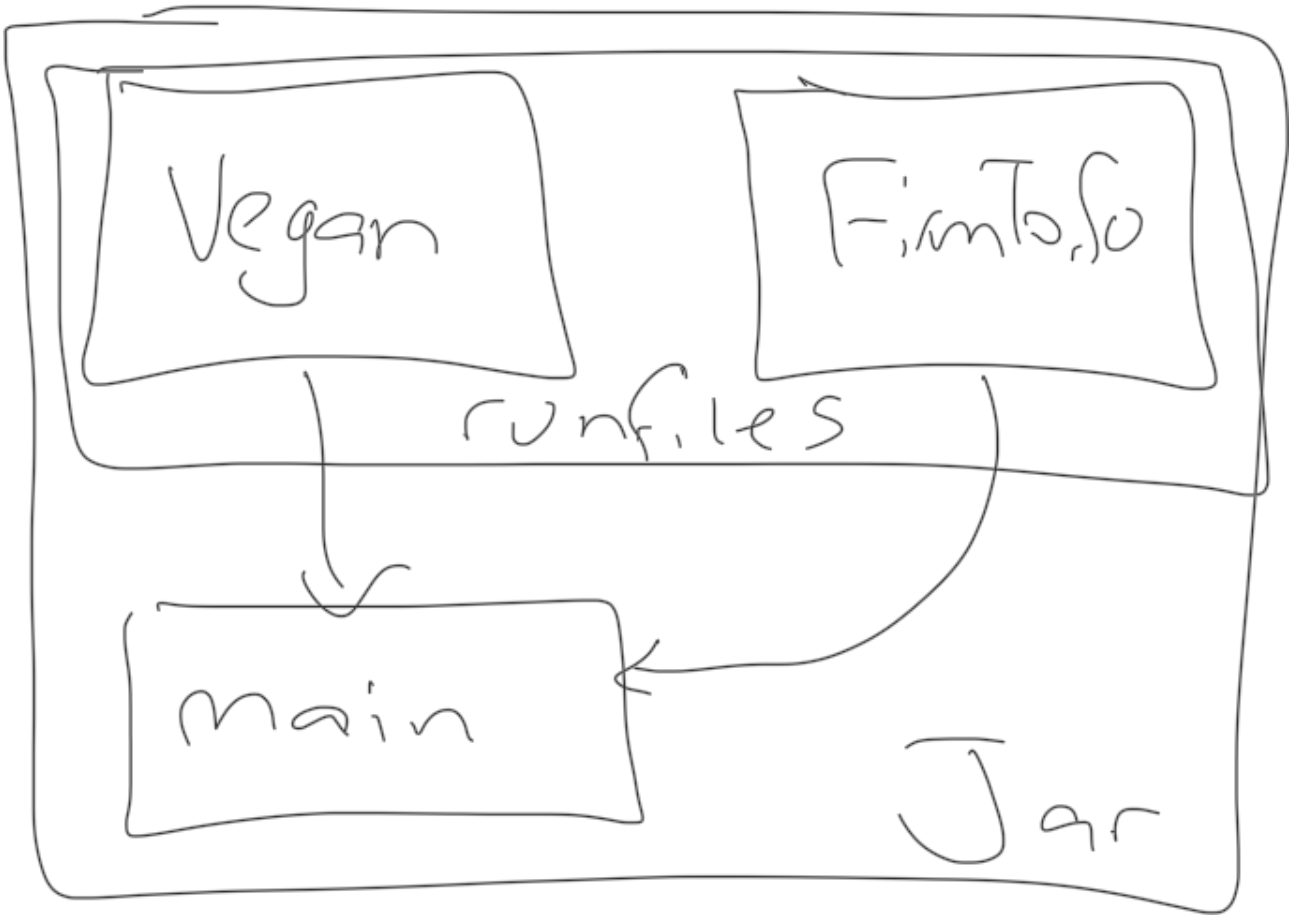
export accepts a list of metadata replacements to transform internal modules to their external form, for example, replacing bazel/foo/bar with an equivalent remote module, repository.opentofu.org/foo/bar.

Export accepts a boolean to indicate whether external modules/provider content should be removed

Wrapper Macro

The ruleset provides the BakedTofu executable format, which is effectively a Java JAR containing the Vegan runtime, a FirmTofu archive, and a main program to link them together. It exists to simplify execution at runtime so the end user can reason about provisioning as running a single executable. It's not single program but rather a class of programs, as it comes form the Binary Rule and each instance bundles different data.

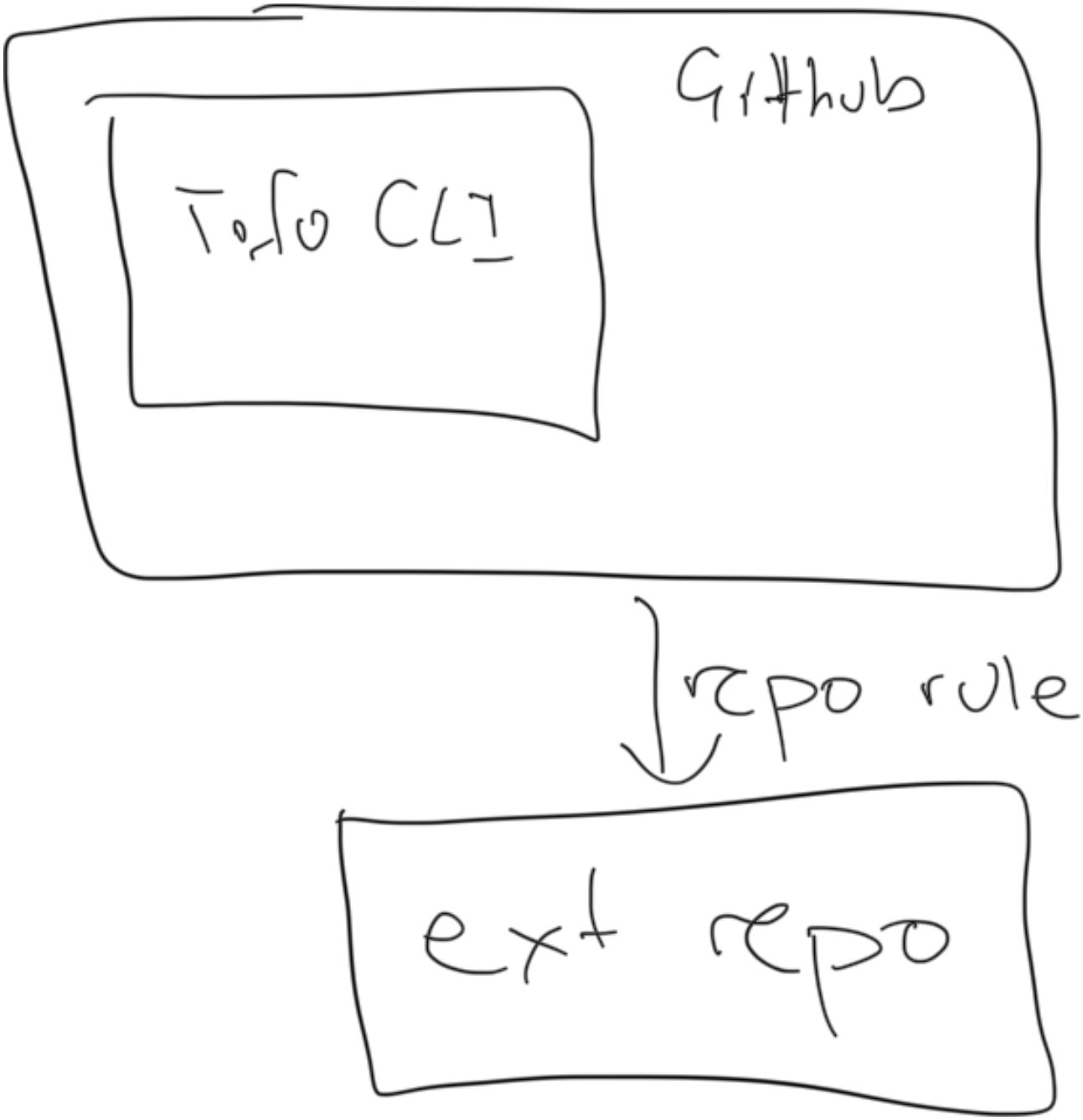
The binary rule produces a FirmTofu archive that is not executable on its own, and is conceptually equivalent to a JAR with a main class. It can only be executed by passing it to the vegan runtime on the target host, and does not directly bundle the runtime. A convenience macro is provided to simplify execution as a one step process, by creating a native executable that bundles the vegan runtime, the archive, a JRE (for vegan), and all the logic necessary to unpack the contents and pass the archive to vegan. It was written in Go for native support.



BakedTofu executables are produced by the tofu_binary build macro (introduced below).

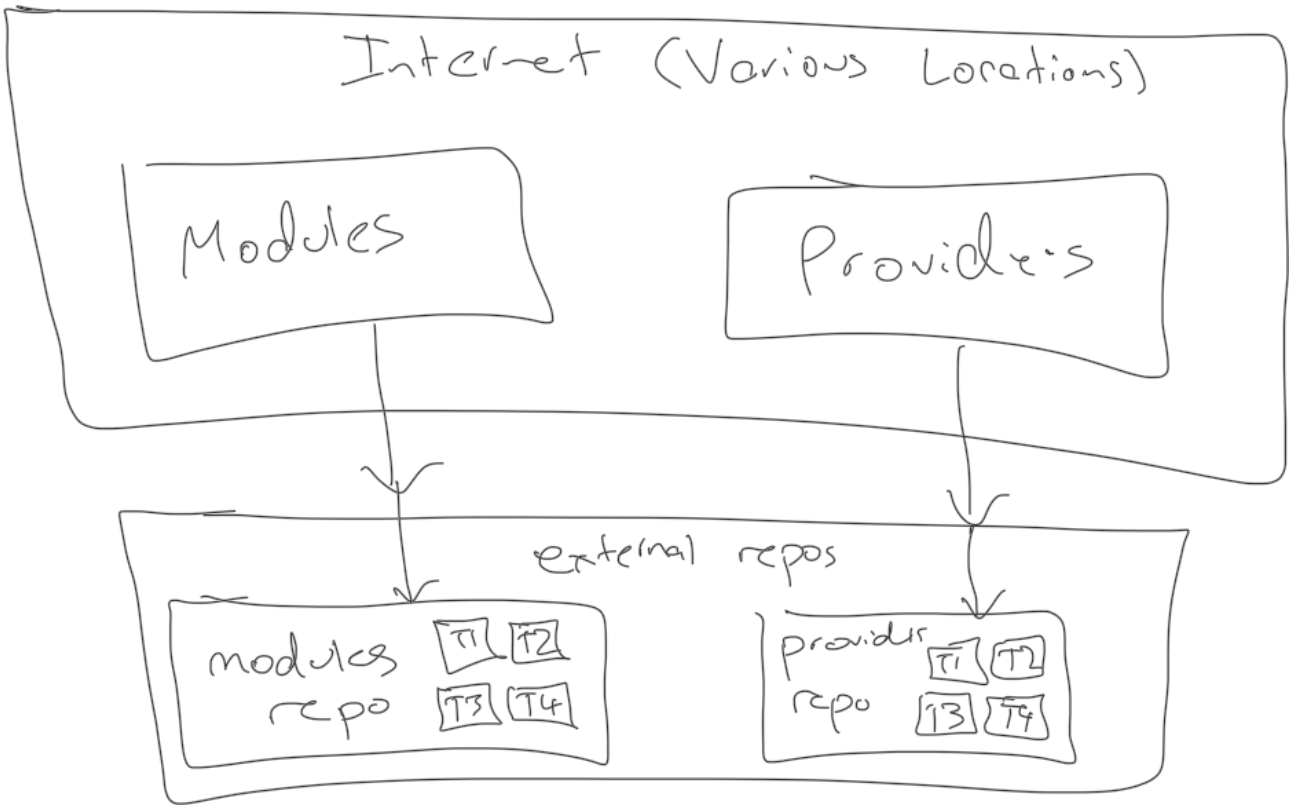
Tofu CLI Repository Rule

The ruleset provides the Tofu CLI repository rule to make the Tofu CLI available to the build. It accepts the address to download from and performs the download using the standard ctx.download operation. It exposes the CLI as an executable file target, and handles the platform resolution steps using the select function.



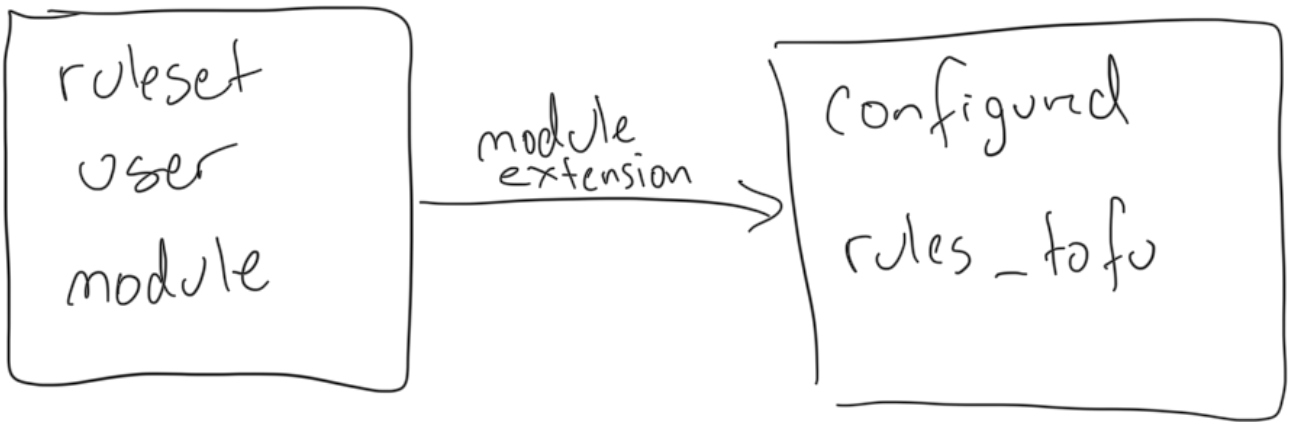
Tofu Dependency Repository Rule

The ruleset provides the tofu dependency repository rule to make external modules and providers available in the build. It accepts a flat list of modules and providers, downloads them using Soybean, and creates targets to make them accessible. In a previous design iteration the rule was divided into two (one for modules one for providers) but they were merged to simplify generation tasks.



Module Extension

The ruleset provides a single module extension for integration with the ruleset user's module. It exists to collect configuration information and orchestrate external dependency resolution via the build rules.



The module extension has two responsibilities:

1. Downloading the Tofu CLI and registering the Tofu CLI toolchain.
2. Downloading the module and provider dependencies into external repositories.

Operation 1 is straightforward. The module extension provides a tag to specify which Tofu CLI to use as an HTTP address pointing to a download location. It is optional and may be elided to specify the latest version from GitHub. It's an error to set it repeatedly though and doing so will cause the build to fail. In the module extension itself, the Tofu CLI Repository Rule is used to perform the download, and a toolchain is registered using the standard toolchain API. Overall this makes the Tofu CLI available to build rules without dependence on the host system.

Operation 2 is more complex because it involves transitive dependency resolution and version conflict resolution. While providers themselves are pre-compiled binaries and thus have no transitive dependencies, modules may depend on both modules and providers, so given a list of modules and providers, the transitive closure may include other modules and other providers. Furthermore, as modules are not globally coordinated in any way, multiple versions of the same module may exist in the transitive closure, and a strategy is required for handling them.

In order to be intentional about versioning and reduce the possibility of dependency hell, the ruleset follows an approach similar to Bzlmod versioning, whereby it resolves a single version for each dependency by default, but allows ruleset users to specify version overrides as required. Reusing that approach for Tofu module/provider resolution ensures a single-version strategy is the norm without locking ruleset users into dependency hell when versions require fine tuning. Implementation wise this means the module extension automatically upgrades every module to use the latest version in the dependency graph, but allows ruleset users to specify specific dependencies to not upgrade.

For example, consider dependency A which has versions 1.1. and 1.2 in the resolved transitive closure. Most modules which depend on A may work with version 1.2, but one specific dependency, dependency B, might only work with version 1.1. In that case, the user must specify that dependency B, but only dependency B, can use version 1.1 of dependency A, while all other versions continue to use version 1.2. This ensures the majority of the graph is using the latest version and positions overrides as a workaround not an ideal solution.

An important details is the handling of transitive dependencies of the overridden module. While the override process allows a specific module to retain a dependency on a specific version of another module/provider, the retained module may still depend on other modules, and those versions will be upgraded. In order to specify a version override for the entire transitive closure of a module, a version override must be set for every transitive dependency. This prevents the top-level configuration from becoming cumbersome and keeps version overrides focused on the specific module in question.

For operation 2, the module extension provides four tags:

1. A tag to specify first order modules dependencies.
2. A tag to specify first order provider dependencies.
3. A tag to specify modules version overrides.

4. A tag to specify provider version overrides.

All tags are optional and can be elided to specify no dependencies/overrides. Tags 1 and 2 accept a dictionary where the keys are HTTP addresses pointing to download locations and the values are version strings. Tags 3 and 4 accept two parameters, the first to specify the target for the version override and the second to specify the dependency to retain (i.e. which has a dependency to keep, and which dependency to keep).

The module extension resolves the transitive closure of all first order dependencies, applies upgrading with respect to version overrides, then creates a flat list of all modules and providers. The lists are then passed to the repository rules so they can be downloaded and exposed as targets. In order to keep the transitive dependencies isolated from the first order dependencies, first order dependencies and transitive dependencies are stored in different repositories, and the transitive dependencies repository is only visible to the first order repository. This prevents ruleset users from accidentally depending on the targets for transitive dependencies in their build targets.

Since version resolution happens in the module extension it will be evaluated as soon as any target in a build execution depends on a single external module. This is not ideal for performance, but cannot be avoided, as the entire transitive closure is required for version resolution.

The alternative of allowing multiple versions by default was considered, but it does not align with the Bazel philosophy or general engineering best practices, as it creates a significant opportunity for dependency hell and does not scale to large repositories. For these reasons the single-version approach was selected instead.

The module extension and repository rules presently depend on the Soybean program for downloading external dependencies, but two alternatives were considered. First, the Tofu CLI could have been invoked directly in the extension and rules, and second, the CTX.download function could have been used to perform the download using Bazel's built in logic. Both eliminate the need for the Soybean program entirely, and while that may reduce typing, it prevents rigorous unit testing of the download functionality, and in the case of option two, would require reimplementing complex ecosystem-specific authentication/fetching logic in Starlark. For maintainability, Soybean was used. In summary, the module extension and repository rules use Soybean internally to avoid reinventing the wheel.

Implementation

All components and processes are complex enough to warrant further detail, except for the Import Rule and the Tofu CLI Repository Rule which are straightforward and require virtually no design work.

FirmTofu

TofuPress

TofuPress is a compiler for Tofu. It produces FirmTofu archives from module/provider sources and checks their contents for correctness. It was written in Java for cross-platform support and to avoid coupling it with Bazel (thereby allowing standalone use). It provides four core operations:

1. Compilation, which creates archives from module/provider sources.
2. Merging, which is turning multiple archives into one equivalent archive.
3. Reduction, which is removing modules/providers from an existing archive.

FirmTofu works with files on the local system and does not perform fetching of remote dependencies, as that function is delegated to Soybean. As part of its function as a compiler, it performs For example, when compiling

Note: Archives are treated as immutable, meaning operations which appear to modify an archive actually create new archives and leave the existing archive unchanged. This avoids unintentional data-loss and is simplifies implementation.

Compilation

Compilation creates a FirmTofu archive from native Tofu sources and existing archives. It accepts the following as inputs simultaneously:

1. A root module.
2. A list of non-root modules.
3. A list of providers.
4. A list of other archives.

The compiled artefact is effectively the combined set of all modules/providers from all inputs, but since there can only be one root module in a program, compilation will fail if there are multiple root modules in the combined set. The user is responsible for ensuring that only one root module is supplied.

The program expects all inputs to be provided as local file paths and it does not perform any remote resolution of dependencies (that is handled by Soybean). Source information for externally resolved dependencies must be passed into the compiler with each module so validation can link them together and verify program integrity. If a module is referenced in another but is not present in the combined set (either due to being elided or because source information is missing) then compilation will fail. Source information can be supplied using textproto files, protobuf binary wire format files, and JSON, all based on the module/provider source messages defined in FirmTofu section. Note: It's valid to pass

in module/provider source information without contents, it will simply create a metadata entry in content addressable space without associated content. This is useful when creating library archives that must reference but not contain their external modules/provides.

When processing the root module, it can be used as a validation root or a runtime root, as determined by the no-package-root flag. When the root is treated as a validation root, it is temporarily compiled into the archive, used for validation, then removed. When the root is treated as a runtime root, it remains packaged into the archive and is never removed. This allows validation of library archives which do not include a runtime root, and is necessary because Tofu modules cannot be validated without a root. This is a hard limitation of how the Tofu language works. Internally validation uses the Vegan runtime to unpack the archive before running the Tofu CLI validation function on the result. This ensures the program is both statically valid, passes compilation, and is unpacked correctly. Note: Code is never run as part of this process as that could inadvertently affect production.

When resolving module paths, recursive directories are included by default, and a module will be created in the archive for each directory. This can be disabled by passing in a no-recursion flag. Recursion is the default because Tofu programs outside Bazel are usually structured as deeply nested directory trees, and it would be inconvenient to enumerate every directory.

When resolving provider paths, compilation expects one of three formats in the directory structure, and tries each until one succeeds. First it tries the native Tofu packed structure, then it tries the native Tofu unpacked structure, then it tries a flat list structure (effectively the same structure that FirmTofu uses internally). If the provider directory cannot be parsed into any of these formats, compilation fails with an invalid directory structure argument.

The actual compilation process involves two major steps. First the validation archive is constructed, then the runtime archive is constructed (by removing the validation root if necessary). The first stage involves allocating a temporary working directory, transforming the code into a FirmTofu structure, then zipping the directory into an actual FirmTofu file. The second stage is effectively a reduction operation and is covered below in the reduction section.

The first stage is characterised by the transformations it must apply to the source files. After the process every module and every provider will exist in content addressable storage with a metadata file

The first stage involves the following operations:

1. For each provider, format its contents using the FirmTofu structure (i.e. flat list of binaries named $os_$arch) and insert it into content addressable storage. Next create a metadata file with its source information and insert it into content addressable storeage. At this point it will contain only source information, no references.
2. For each module, insert its contents into content addressable storage, and create a metadata file with the known information. At this point it will contain some but not all of the references, as some are yet to be discovered. As each module is processed, update the references for

A few other design notes:

– The compilation operation can be performed on-disk or in-memory, and the program accepts a configuration value to set this. The former is necessary for archives which are large enough to overload memory and must be progressively constructed on disk before being zipped. This divergence affects the underlying implementation but does not affect general compilation and there result is the same form both paths.

– If two inputs contain the same module/provider (by content address), the compiler can check they have the exact same content. In theory they always should due to content addressing, but in practice it might be necessary to check. For performance this is disable by default and can be enabled by passing the check-hashes-on-merge or check-hashes flags (the former checks all contents).

Merging

Merging archives is the process of creating one archive that represents the combined set of multiple other archives. It's equivalent to a compilation operation with archive inputs only (i.e. no root module, no modules, and no providers). An error is raised during merging if:

1. The archives have different versions.
2. The archives have multiple root modules (collectively).

This operation is effectively a convenience wrapper to simplify usage.

Reduction

Reducing archives is the process of removing content from an archive to create a simpler (i.e. reduced) version.

Supplementary

1. Verify Content
2. Validate Content

Operation 1 scans through the content in an archive and confirms the hash for each entry is correct. It requires recomputing the hash for every entry, which could be inconvenient depending on the size of the archive and the resources of the host, so it is provided as an optional operation, and not an integral part of the core operations.

Operation 2 recompiles the contents of an archive to verify it's still compiled correctly. It works by decompiling the program using the embedded vegan program then compiling them again. It's offered as a supplemental operation to check archives which have been manually modified outside TofuPress or may have become corrupted.

Details on how the algorithm will work.

1. The input is the module source address to fetch.
2. The output is a firm tofu archive that contains the requested module and all its dependent modules in a way that can be unpacked to the disk at runtime.

At runtime, each top-level module will be unpacked to a parallel directory, and its contents will be reassembled exactly, except for module references, which will be rewritten to point at directories with

absolute paths (possible since the output directory is known during unpacking). This unpacked format requires the archive format to store both the directory structure and the references between modules so the tree can be reconstructed for each top-level directory and the references can be updated to absolute paths.

Algorithm for unpacking

Assume going with the previously designed archive format. Each directory is stored in parallel, named by its content address, and a metadata file for each records the directory structure and reference structure (with bidirectional links). Directory names are not part of the content hash and are stored in metadata. Reassembly is effectively a recursive graph traversal operation where contents are unpacked, and if an item needs to be unpacked somewhere that hasn't been unpacked yet, recursively unpack it first. Exact details are tbd, but it should work. Given an item the details are:

1. If its a top-level item, create a top-level directory, and unpack it there. The directory name is arbitrary, but must be recorded in memory so references can be poined to it (i.e. record a content address to directory map). If its not a top level item, unpack its parents (recursive).
2. Unpack its children, storing each content address in the directory map.
3. Iterate over the entire tree and replace all module references (content addresses) with directory references from the map.

The result transforms the flat content space into a directory hierarchy where each external module is in a separate top-level directory with all its contents reassembled exactly, and references are all updated to use absolute references.

Soybean

TODO jack update this so they output FirmTofu archives for simplicity

Soybean handles downloading Tofu modules and providers at build time so they can be used in the Bazel build graph. In essence, it's the bridge between Bazel's model of build-time external dependencies and Tofu's model of runtime dependencies. It was written in Java to support cross-platform build/execution, functions as a standalone executable, and can be used outside Bazel.

Soybean exposes the following functions:

1. Download module.
    1. Inputs:
        1. Module (spec). The fully-qualified address of the module to download.

        2. Output (path). The location to write the downloaded files to (directory)
    2. Outputs: A directory containing the downloaded module sources, including transitive module dependencies, but not provider binaries.
2. Download provider.
    1. Inputs:
        1. Provider (spec): The fully-qualified address of the provider to download.
        2. Output (path). The location to write the downloaded provider files to.
    2. Outputs: A directory containing the downloaded provider sources for all architectures/OSes.

Internally soybean uses the Tofu CLI to perform the download. It generates a basic tofu file that uses the required provider/module, then calls the CLI to perform the download. After it has downloaded the dependencies, it uses TofuPress to create a FirmTofu and outputs it.

TODO more implementation details

Supplementary

The following supplementary operations are provided:

1. Export the manifest to a human readable format.
2. Lookup a module from source information.
3. Lookup a module from content address.
4. Lookup a provider from source information.
5. Lookup a provider from content address.
6. Lookup the submodules of a module.
7. Lookup the supermodules of a module.
8. Lookup the providers of a module.
9. Lookup the modules of a provider.
10. Export the FirmTofu archive.

Operation 1 accepts a flag specifying JSON or XML and exports the manifest in that format. It exists because reading the manifest is useful during debugging but the protobuf-based manifest is not human readable.

Operation 2 accepts the source information of a module and returns its content address, and operation 3 does the opposite. Operations 4 and 5 are equivalent operations for providers. Operations 6, 7 and 8 accept the content address of a module and return the content addresses of its submodules, supermodules, and providers (respectively). Operation 9 accepts the content address of a provider and returns the content addresses of the providers which use it. These operations exist to allow graph traversal which can be useful during debugging.

Operation 10 accepts the location to write the archive to and exports it for direct use. This operation exists to support debugging.

Side note: Need to support patching in bazel. Add a way to write patches in soybean. P3.