Import Versioning

DRAFT (November 20, 2017)

How should we identify and build different versions of Go packages?

This is the fundamental design choice for package management in Go, and the answer is pivotal in determining the complexity of the resulting system. The choice decides how easy or difficult package management will be to use and also to implement.

It may seem like there is an obvious answer. In fact there are at least two obvious answers, and we need to pick just one. In this post I want to work through those answers and evaluate their implications. To end the suspense, here's the answer I believe is the right one:

In Go, if an old package and a new package have the same import path, the new package must be backwards compatible with the old package.

(If you are a SemVer fan, consider the parallel with this statement: in SemVer, if an old package and a new package have the same major version, the new package must be backwards compatible with the old package. I'll return to SemVer later.)

A Dependency Story

To make the discussion concrete, consider the following story.

As an aside, this setup and the events that follow seem realistic to me given what we know of how software evolves and how important foundational libraries end up being used by a variety of projects. The similarity to reality here is critical: there's no point in basing decisions on unlikely hypotheticals. [1]

Prologue

From the perspective of a package management tool, there are Authors of code and Users of code.

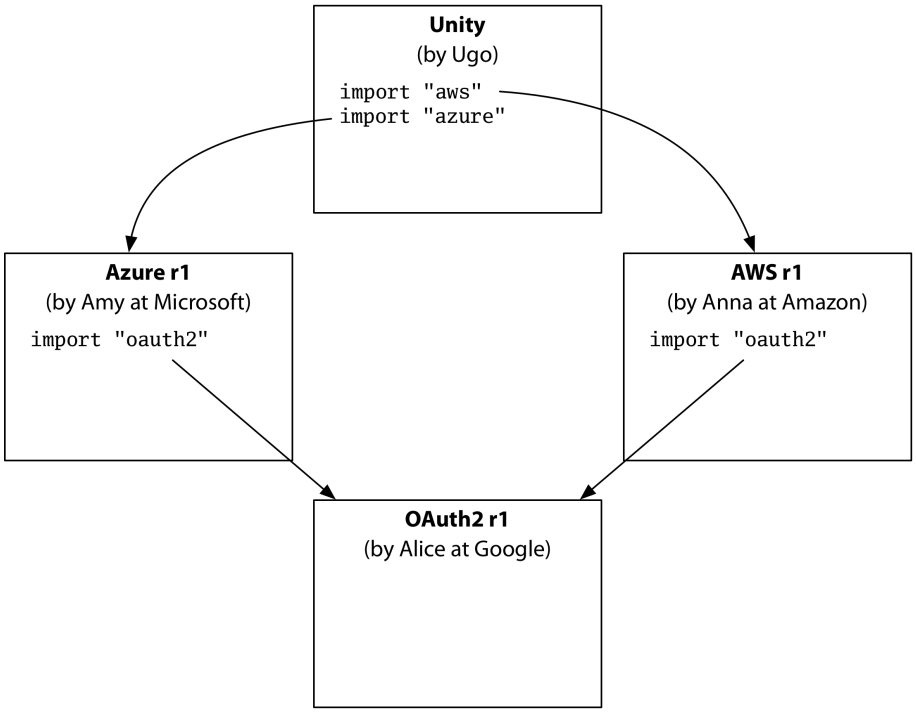
Alice, Anna, and Amy are Authors of different code packages. Alice works at Google and wrote the OAuth2 package. Amy works at Microsoft and wrote the Azure client libraries. Anna works at Amazon and wrote the AWS client libraries. Ugo is the User of all these packages. He's working on the ultimate cloud app, Unity, which uses all of those packages and others.

As the Authors, Alice, Anna, and Amy need to be able to write and release new versions of their packages, and also to specify each version's requirements for its own dependencies.

As the User, Ugo needs to be able to build Unity with these other packages; he needs control over exactly which versions are used in a particular build; and he needs to be able to update to new versions when he chooses.

There's more that our friends might expect from a package management tool, especially around discovery, testing, portability, and helpful diagnostics, of course, but those are not relevant to the story.

As our story opens, Ugo's Unity build dependencies look like:



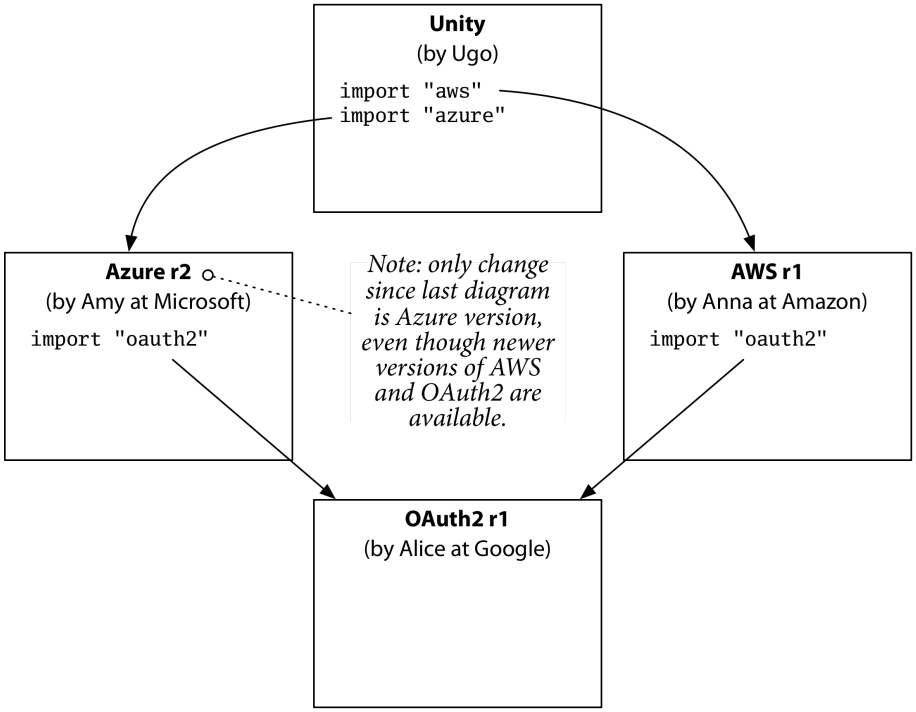
Chapter 1

At Google, Alice has been busy designing a new, simpler, easier to use API for the OAuth2 package. It can still do everything that the old package can do, but with half the API surface. She releases it as OAuth2 r2. (The 'r' here stands for revision. For now, the revision numbers don't indicate anything other than sequencing: in particular, they're not SemVer versions.)

At Microsoft, Amy is on a well-deserved long vacation, and her team decides not to make any changes related to OAuth2 r2 until she returns. The Azure package will keep using OAuth2 r1 for now.

At Amazon, Anna finds that using OAuth2 r2 will let her delete a lot of ugly code from the implementation of AWS r1, so she changes AWS to use OAuth2 r2. She fixes a few bugs along the way and issues the result as AWS r2.

Ugo gets a bug report about behavior on Azure, and he tracks it down to a bug in the Azure client libraries, which Amy already fixed and released as Azure r2 before leaving for vacation. Ugo adds a test case to Unity, confirms that it fails, asks the package management tool to update to Azure r2, confirms that the new test passes and that all his old tests still pass, and locks in the Azure update:



Chapter 2

To much fanfare, Amazon launches their new cloud offering, Amazon Zeta Functions. In preparation for the launch, Anna added Zeta support to the AWS package, which she now releases as AWS r3.

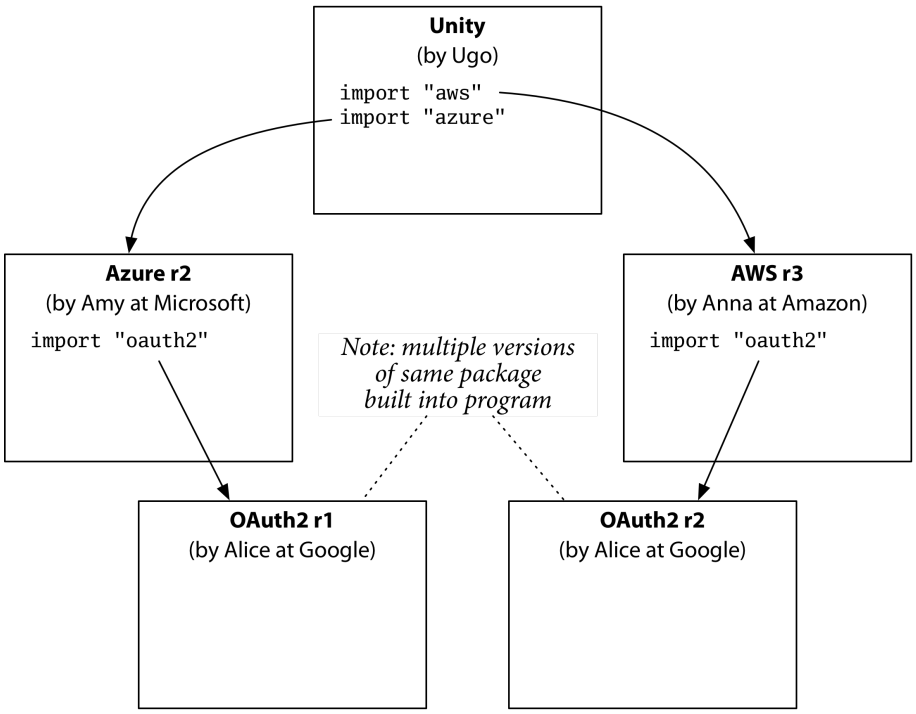
When Ugo hears about Amazon Zeta, he writes some test programs and is so excited about how well they work that he skips lunch to update Unity. Today's update does not go as well as the last one.

Ugo wants to build Unity with Zeta support using Azure r2 and AWS r3, the latest version of each. But Azure r2 needs OAuth2 r1 (not r2), while AWS r3 needs OAuth2 r2 (not r1). Classic diamond dependency, right? Ugo doesn't care what it is. He just wants to build Unity.

Worse, it doesn't appear to be anyone's fault. Alice wrote a better OAuth2 package. Amy fixed some Azure bugs and went on vacation. Anna decided AWS should use the new OAuth2 (an internal implementation detail) and later added Zeta support. Ugo wants Unity to use the latest Azure and AWS packages. It's very hard to say any of them did something wrong.

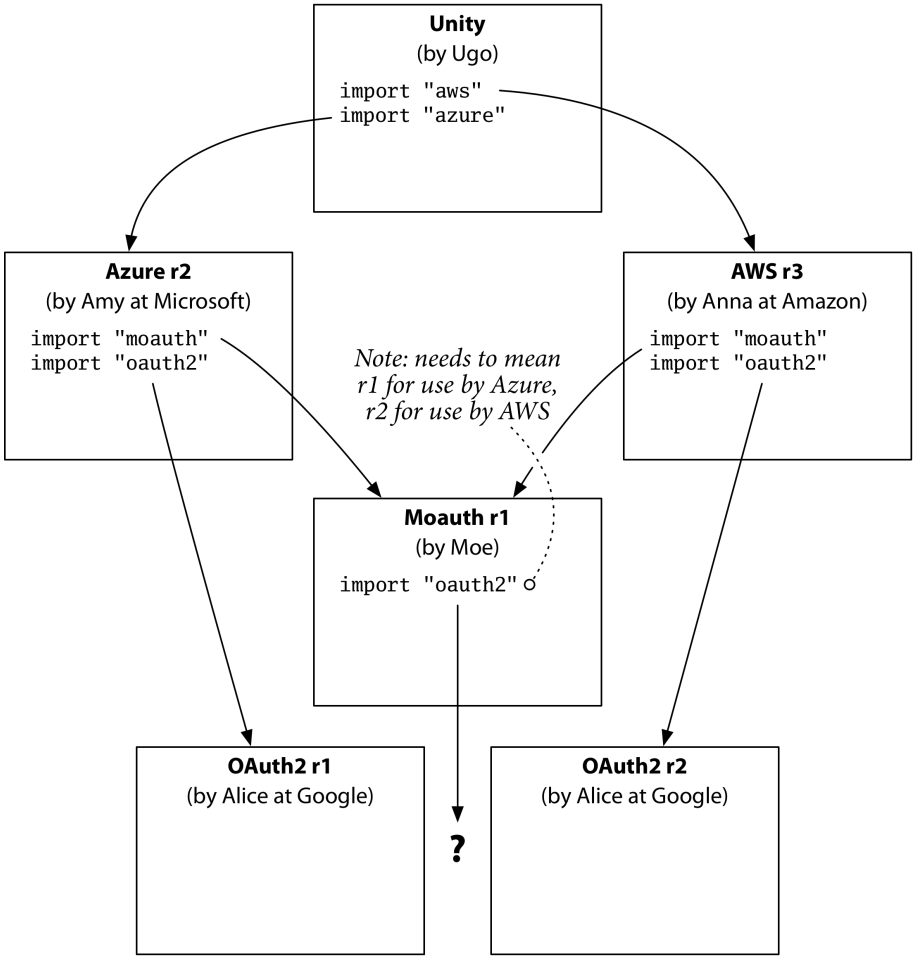
If these people aren't wrong, then maybe the package manager is. We've been assuming that there can be only one version of OAuth2 in Ugo's Unity build. Maybe that's the problem: maybe the package manager should allow different versions to be included in a single build. (This example would seem to indicate that it must.)

Ugo is still stuck, so he searches StackOverflow and finds out about the package manager's `-fmultiverse` flag, which allows multiple versions, so that his program builds as:



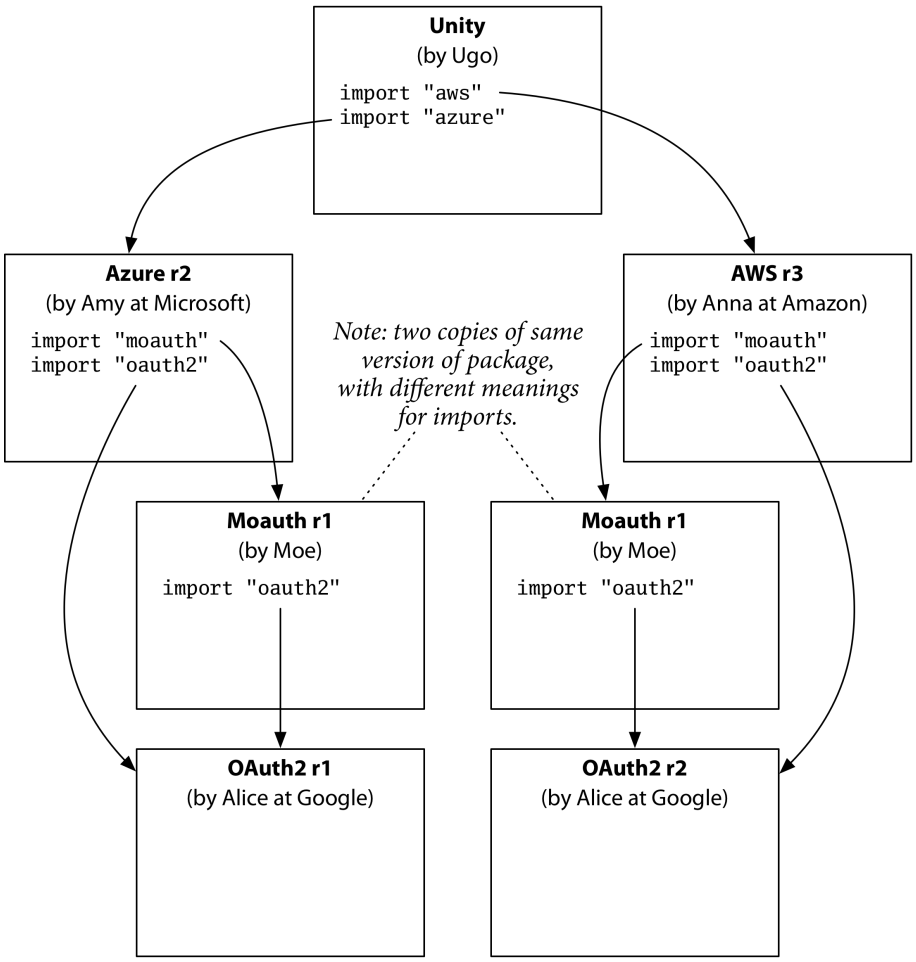
Ugo tries this. It doesn't work.

Digging further into the problem, Ugo discovers that both Azure and AWS are using a popular OAuth2 middleware library called Moauth that simplifies part of the OAuth2 processing. Moauth is not a complete API replacement: users still import OAuth2 directly, but they use Moauth to simplify some of the API calls. The details that Moauth helps with didn't change from OAuth2 r1 to r2, so Moauth r1 (the only version that exists) is compatible with either. Both Azure r2 and AWS r3 use Moauth r1. That works fine in programs using only Azure or only AWS, but Ugo's Unity build actually looks like:



Unity needs both copies of OAuth2, but then which one does Moauth import?

In order to make the build work, it would seem that we need two identical copies of Moauth: one that imports OAuth2 r1, for use by Azure, and a second that imports OAuth2 r2, for use by AWS. A quick StackOverflow check shows that the package manager has a flag for that: `-fclone`. Using this flag, Ugo's program builds as:



Ugo heads home for a late dinner.

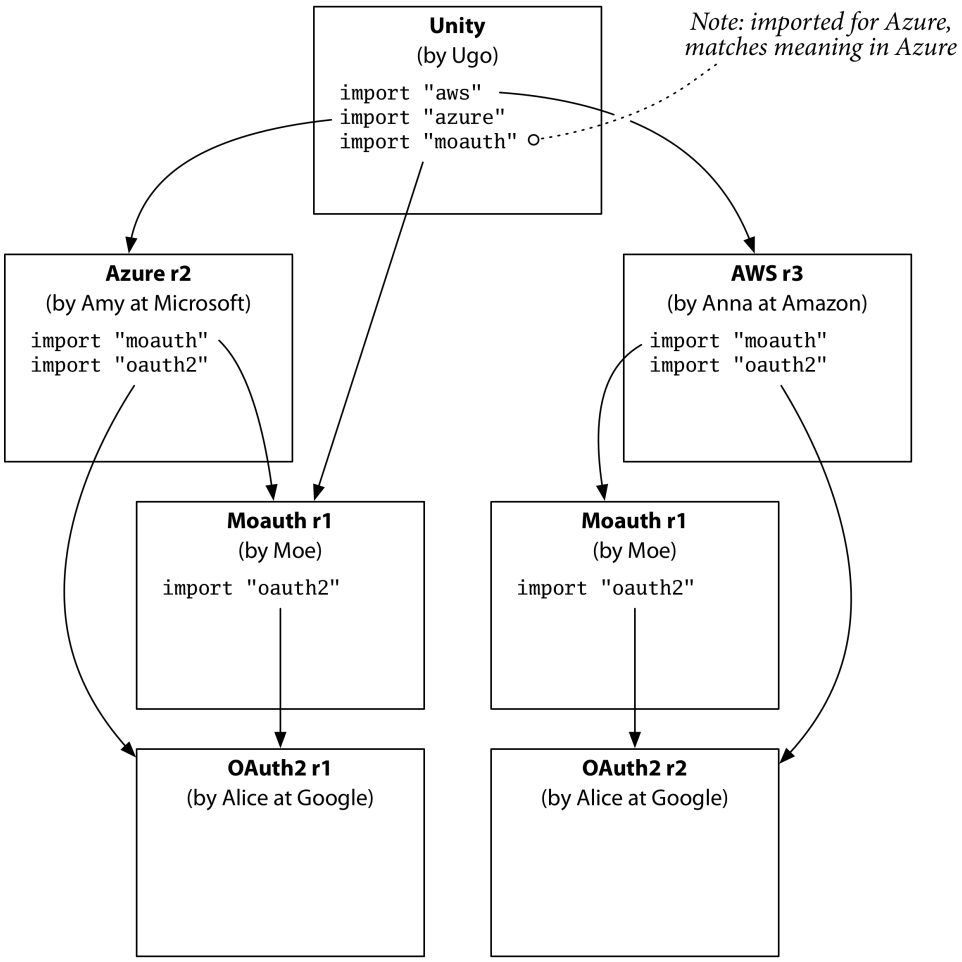
Chapter 3

Back at Microsoft, Amy has returned from vacation. She decides that Azure can keep using OAuth2 r1 for a while longer, but she realizes that it would help users to let them pass Moauth tokens directly into the Azure API. She adds this to the Azure package in a backwards-compatible way and releases Azure r3. Over at Amazon, Anna likes the Azure package's new Moauth-based API and adds a similar API to the AWS package, releasing AWS r4.

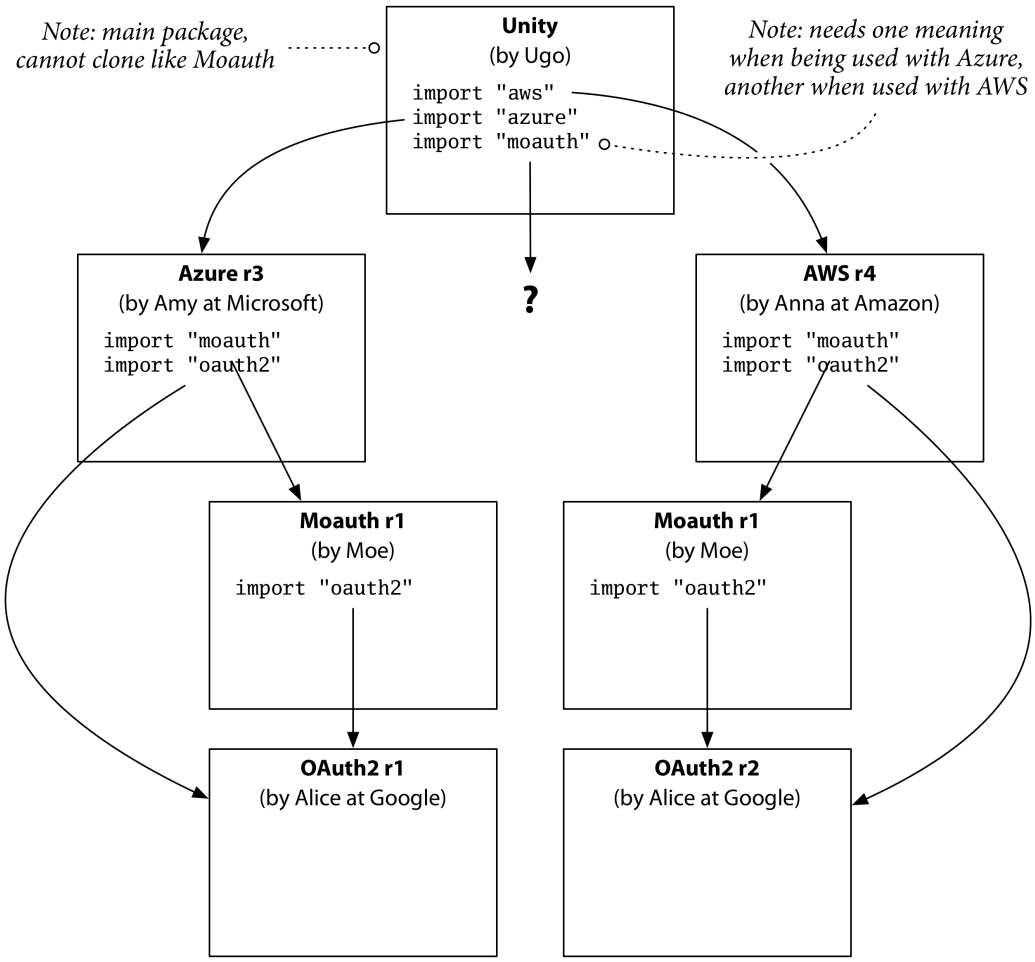
Ugo sees these changes and decides to update to the latest version of both Azure and AWS in order to use the Moauth-based APIs. This time he blocks off an afternoon. First he tentatively updates the Azure and AWS packages without modifying Unity at all. His program builds!

Excited, Ugo changes Unity to use the Moauth-based Azure API, and that builds too. When he changes Unity to also use the Moauth-based AWS API, though, the build fails. Perplexed, he reverts his use of the Moauth-based Azure API, and the build succeeds. He puts the Azure changes back, and the build fails again. Ugo returns to StackOverflow.

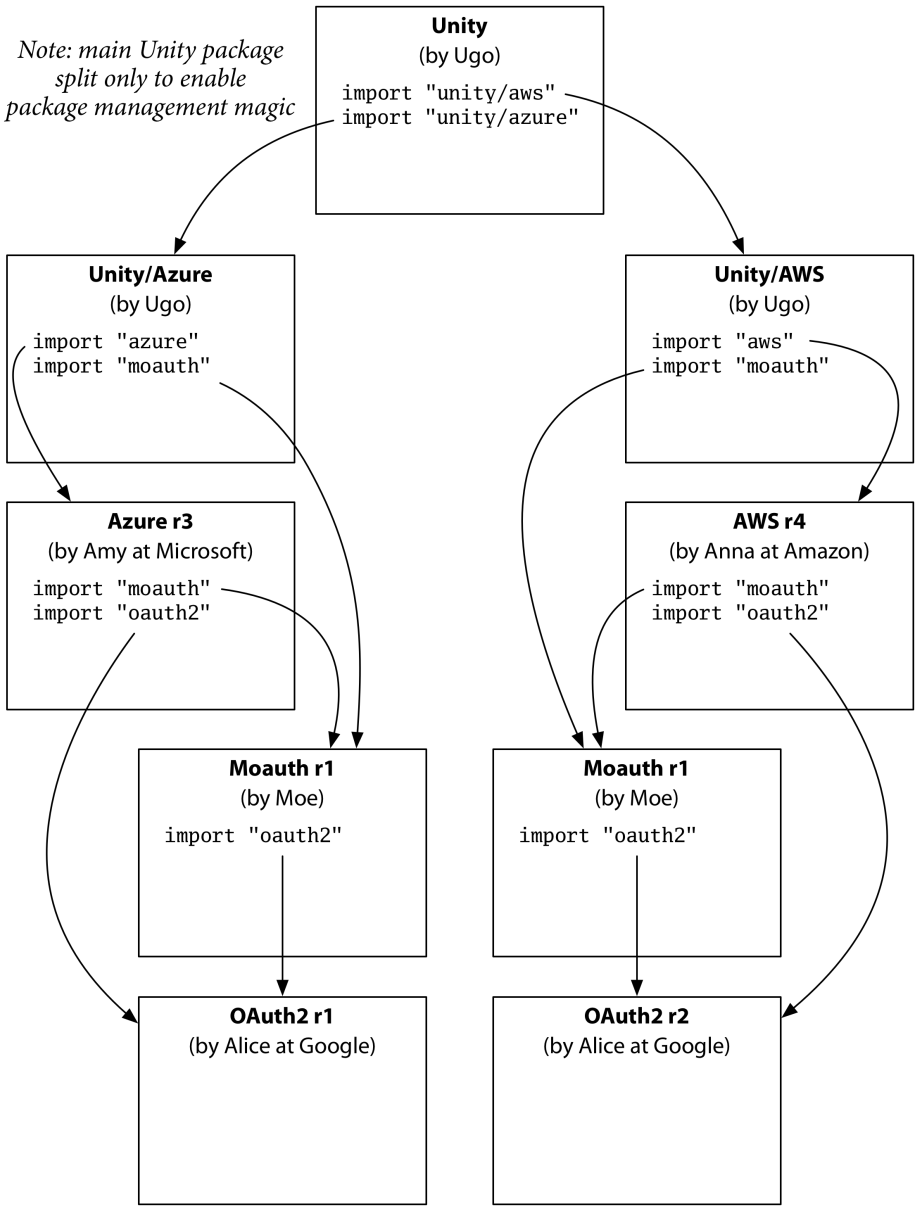
Ugo learns that when using just one Moauth-based API (in this case, Azure) with `-fmultiverse -fclone`, Unity implicitly builds as:



but when he is using both Moauth-based APIs, the single `import "moauth"` in Unity is ambiguous. Since Unity is the main package, it cannot be cloned (in contrast to Moauth itself):



A comment on StackOverflow suggests moving the Moauth import into two different packages and having Unity import them instead. Ugo tries this and, incredibly, it works:



Ugo makes it home on time. He's not terribly happy with his package manager, but he's now a big fan of StackOverflow.

A Retelling with Semantic Versioning

Let's wave a magic wand and retell the story with Semantic Versioning, assuming that the package manager uses them instead of the original story's 'r' numbers.

Here's what changes:

- OAuth2 r1 becomes OAuth2 1.0.0.
- Moauth r1 becomes Moauth 1.0.0.
- Azure r1 becomes Azure 1.0.0.
- AWS r1 becomes AWS 1.0.0.
- OAuth2 r2 becomes OAuth2 2.0.0 (partly incompatible API)
- Azure r2 becomes Azure 1.0.1 (bug fix).
- AWS r2 becomes AWS 1.0.1 (bug fix, internal change to use OAuth2 2.0.0)
- AWS r3 becomes AWS 1.1.0 (feature update: add Zeta)

- Azure r3 becomes Azure 1.1.0 (feature update: add Moauth-based APIs)
- AWS r4 becomes AWS 1.2.0 (feature update: add Moauth-based APIs)

Nothing else about the story changes. Ugo still runs into the same build problems, and he still has to turn to StackOverflow to learn about build flags and refactoring techniques just to keep Unity building. According to SemVer, though, Ugo should have had no trouble at all with any of his updates: not one of the packages that Unity imports changed its major version during the story. What went wrong?

The problem here is that the SemVer spec is really not much more than a way to choose and compare version strings. It says nothing else. In particular, it says nothing about how to handle incompatible changes after incrementing the major version number.

The most valuable part of SemVer is the encouragement to make backwards-compatible changes when possible. The FAQ correctly notes:

> "Incompatible changes should not be introduced lightly to software that has a lot of dependent code. The cost that must be incurred to upgrade can be significant. Having to bump major versions to release incompatible changes means you'll think through the impact of your changes and evaluate the cost/benefit ratio involved."

I certainly agree that "incompatible changes should not be introduced lightly." Where I think SemVer falls short is the idea that "having to bump major versions" is a step that will make you "think through the impact of your changes and evaluate the cost/benefit ratio involved." Quite the opposite: it's far too easy to read SemVer as implying that *as long as you increment the major version when you make an incompatible change, everything else will work out.*

From Alice's point of view, the OAuth2 API needed backwards-incompatible changes, and when she made them, SemVer seemed to promise it would be fine to release an incompatible OAuth2 package, provided she gave it version 2.0.0. Yet that apparently SemVer-approved change was the trigger for the cascade of problems that befell Ugo and Unity.

SemVer notation is important as a way for authors to convey expectations to users, but that's all it is. By itself, it can't be expected to solve these larger build problems. Instead, let's look at an approach that does solve the build problems. Afterward, we can consider how to fit SemVer into that approach.

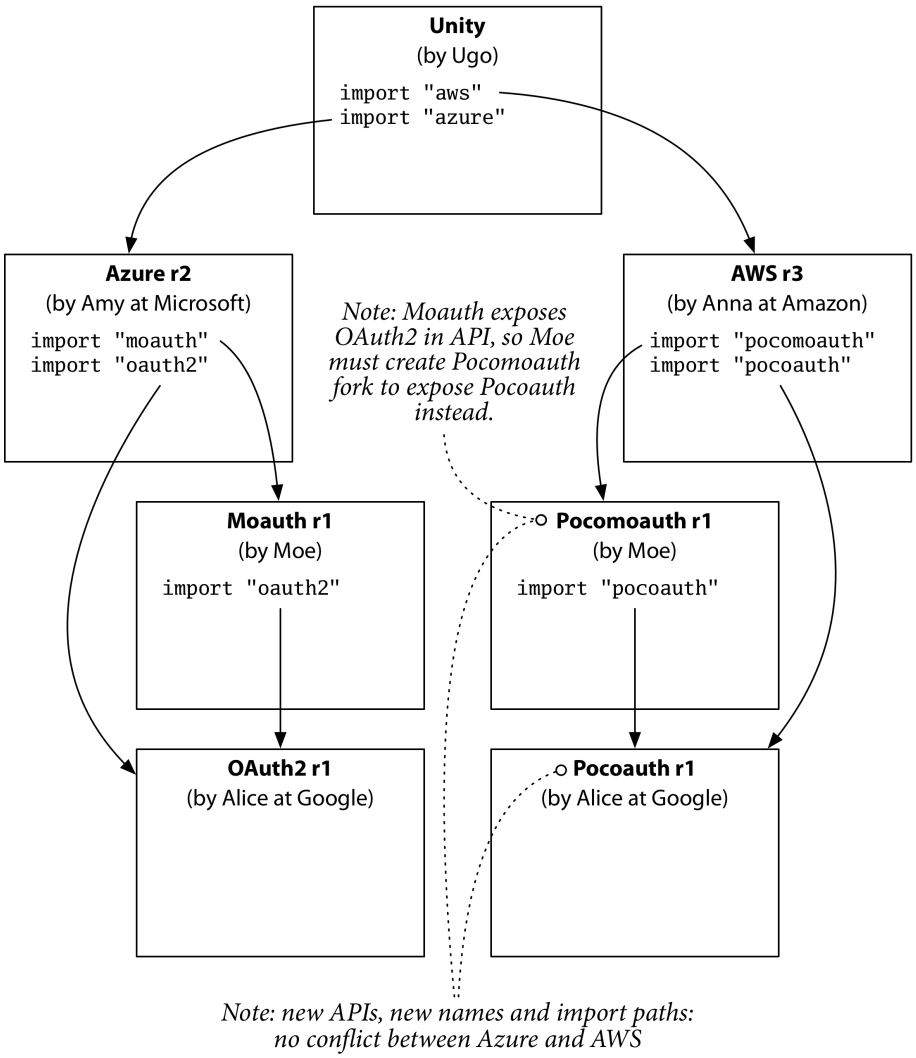
A Retelling with Import Versioning

Once again, let's retell the story, this time using the rule at the start of this post, which I will call Import Versioning:

In Go, if an old package and a new package have the same import path, the new package must be backwards compatible with the old package.

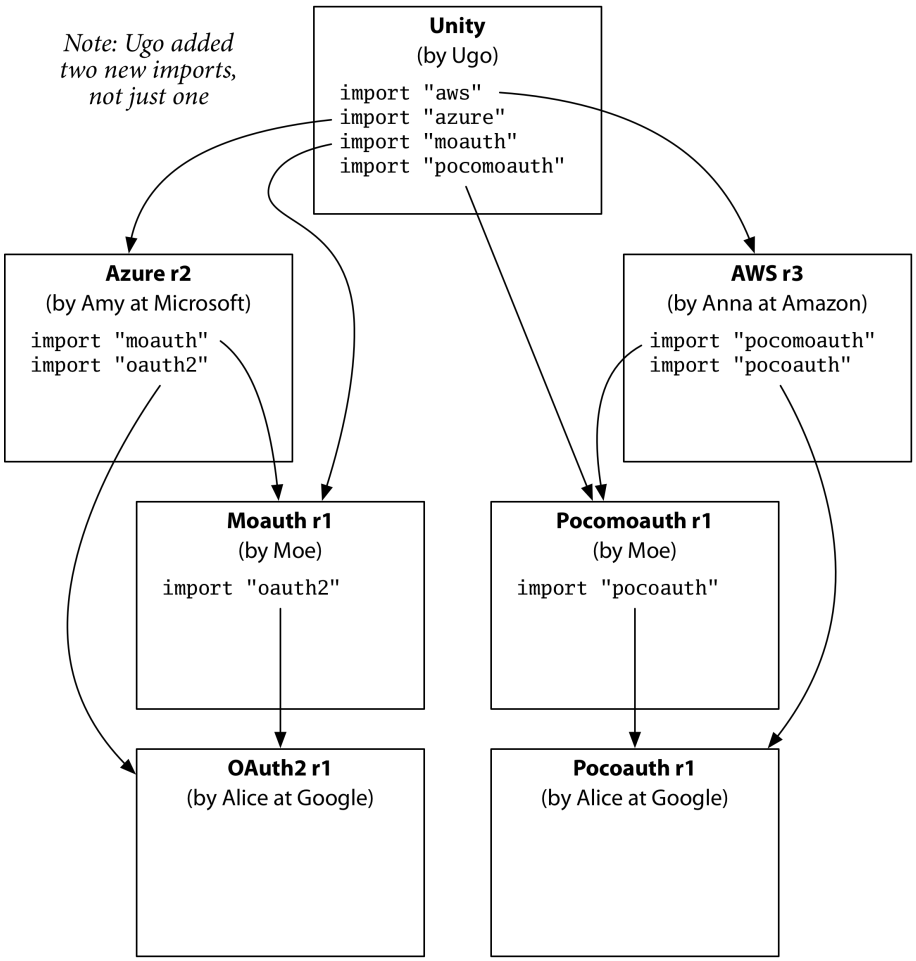
Now the plot changes are more significant. The story starts out the same way, but in Chapter 1, when Alice decides to create a new, partly incompatible OAuth2 API, she cannot use `"oauth2"` as its import path. Instead, she names the new version Pocoauth and gives it the import path `"pocoauth"`. Presented with two different OAuth2 packages, Moe (the author of Moauth) must write a second package, Moauth for Pocoauth, which he names Pocomoauth and gives the import path `"pocomoauth"`. When Anna updates the AWS package to the new OAuth2 API, she also changes the import paths in that code from `"oauth2"` to `"pocoauth"` and from `"moauth"` to `"pocomoauth"`. Then the story proceeds as before, with the release of AWS r2 and AWS r3.

In Chapter 2, when Ugo eagerly adopts Amazon Zeta, everything just works. The imports in all the packages code exactly match what needs to be built. He doesn't have to look up special flags on StackOverflow, and he's only five minutes late to lunch.



In Chapter 3, Amy adds Moauth-based APIs to Azure while Anna adds equivalent Pocomoauth-based APIs to AWS.

When Ugo decides to update both Azure and AWS, again there's no problem. His updated program builds without any special refactoring:



At the end of this version of the story, Ugo doesn't even think about his package manager. It just works; he barely notices that it's there.

In contrast to the semantic versioning translation of the story, the use of import versioning here changed two critical details. First, when Alice introduced her backwards-incompatible OAuth2 API, she had to release it as a new package (Pocoauth). Second, because Moe's wrapper package Moauth exposed the OAuth2 package's type definitions in its own API, Alice's release of a new package forced Moe's release of a new package (Pocomoauth). Ugo's final Unity build went well because Alice's and Moe's package splits created exactly the structure needed to keep clients like Unity building and running. Instead of Ugo and users like him needing incomplete package manager complexity like `-fmultiverse` `-fclone` aided by extraneous refactorings, the Import Versioning rule pushes a small amount of additional work onto package authors, and all users benefit.

There is certainly a cost to needing to introduce a new name for each backwards-incompatible API change, but as the SemVer FAQ says, that cost should encourage authors to more clearly consider the impact of such changes and whether they are truly necessary. And in the case of Import Versioning, the cost pays for significant benefits to users.

An advantage of Import Versioning here is that package names and import paths are well-understood concepts for Go developers. If you tell an author that making a backwards-incompatible change requires creating a different package with a different import path, then—without any special knowledge of versioning—she can reason through the implications on client packages: clients are going to need to change their own imports one at a time; Moauth is not going to work with the new package; and so on.

Able to predict the effects on users more clearly, authors might well make different, better decisions about their changes. Alice might look for way to introduce the new, cleaner API into the original OAuth2 package alongside the existing APIs, to avoid a package split. Moe might look more carefully at whether he can use interfaces to make Moauth support both OAuth2 and Pocoauth, avoiding a new Pocomoauth package. Amy might decide it's worth updating to Pocoauth and Pocomoauth instead of exposing the fact that the Azure

APIs use outdated OAuth2 and Moauth packages. Anna might have tried to make the AWS APIs allow either Moauth or Pocomoauth, to make it easier for Azure users to switch.

In contrast, the implications of a SemVer "major version bump" are far less clear and do not exert the same kind of design pressure on authors.

Semantic Import Versioning

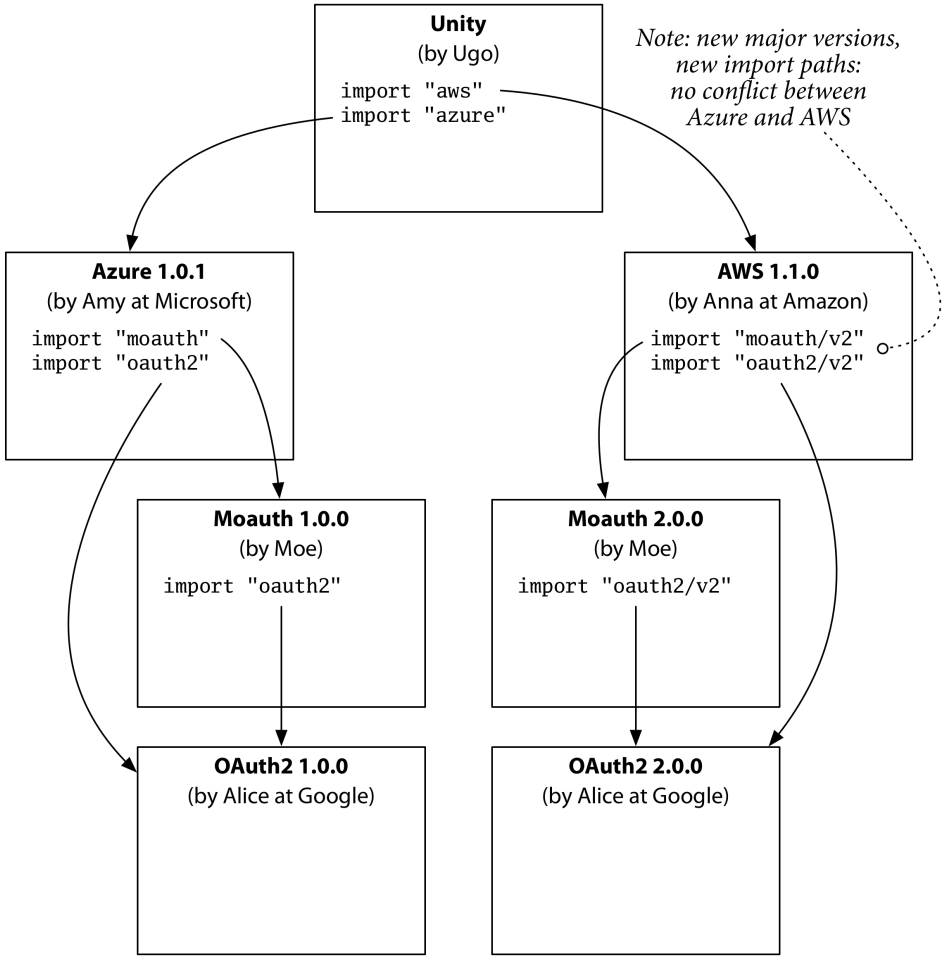
The previous section showed how the Import Versioning rule leads to simple, predictable builds during updates. The best objection to Import Versioning is that being required to choose a new name at every backwards-incompatible change is difficult and unhelpful to users: given the choice between OAuth2 and Pocoauth, which should Amy use? Without further investigation, there's no way to know. In contrast, Semantic Versioning makes this easy: OAuth2 2.0.0 is clearly the intended replacement for OAuth2 1.0.0.

Here are the two rules we've seen:

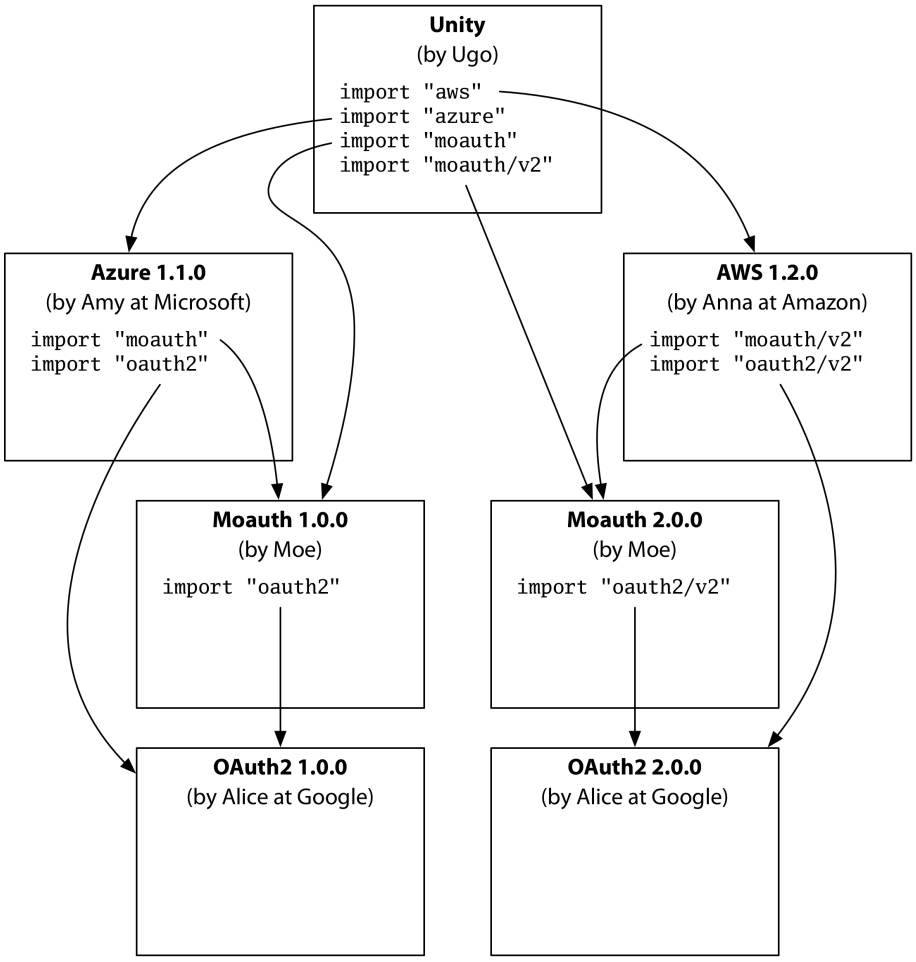
- Semantic Versioning: when making a backwards-incompatible change to a package, its major version number must change.
- Import Versioning: when making a backwards-incompatible change to a package, its import path must change.

To add the version-comparison benefits of Semantic Versioning with the predictable builds of Import Versioning, we must therefore include the major version number in the import path.

With Semantic Import Versioning, instead of needing to invent a cute but unrelated new name like Pocoauth, Alice can call her new API OAuth2 2.0.0, with the new import path `"oauth2/v2"`. The same for Moe: Moauth 2.0.0 (imported as `"moauth/v2"`) can be the helper package for OAuth2 2.0.0, just as Moauth 1.0.0 was the helper package for OAuth2 1.0.0. When Ugo adds Zeta support in Chapter 2, his build looks like:



Because "moauth" and "moauth/v2" are simply different packages, it is perfectly clear to Ugo what he needs to do to use "moauth" with Azure and "moauth/v2" with AWS: import both.



For compatibility with existing Go usage and as a small encouragement not to make backwards-incompatible API changes, I am assuming here that major version 1 is always omitted from import paths: `import "moauth"`, not "moauth/v1". That would also imply import paths like "moauth/v0.1.2" for pre-v1 changes. (Because SemVer defines no compatibility expectations for major version 0, the full three-part number would be necessary, with all the obvious implications about multiple copies of a package being used in a single build.)

Discussion

Twenty years ago, Rob Pike taught me the rule of thumb that when you change a function's behavior, you also change its name. We were modifying the internals of a Plan 9 C library, and changing the name forced the compiler to show us every call to the old function, ensuring that we adjusted all of them. This rule of thumb is even more important in today's world of distributed version control: changing the name makes sure that a merge of concurrently-written code expecting the old semantics does not silently get the new semantics instead. Replacing a function like this works only in situations like modifying package internals, when it's feasible to find and fix all existing uses. For exported APIs, it's usually much better to leave the old name and old behavior intact and only add a new name and behavior.

In the early days of "`go get`", when people asked about making backwards-incompatible changes, our response—based on intution derived from years of experience with these kinds of software changes—was to give the Import Versioning rule, but without a clear explanation why this approach was better than not putting the major version in the import paths. The motivation for this blog post is to show, using a clear, believable example, why following the rule is so important.

This post has focused on making sure that software still builds during updates, but there are added benefits to treating different major versions of a package as different packages. One is that doing so makes it trivial for the v2 API to be written as a wrapper of the v1

implementation, or vice versa. This lets them share the code and, with appropriate design choices and perhaps use of type aliases, might even allow clients using v1 and v2 to interoperate. Allowing different major versions to coexist in a single build also makes it possible to update clients gradually, which is critical in large code bases.

When discussing the possibility of linking two different major versions of a package into a single program, the question always arises: how can an author indicate that this is not allowed? For example, obviously there must be at most one copy of a package that listens on a specific network port or that provides a coordination point like the `http.DefaultServeMux`. If different major versions are treated as different packages, then the answer is clear: if a package must not be duplicated, do not introduce a new major version, or else find a way for the new major version to avoid the problematic duplication. For example, if Go introduced a net/http v2 package, we could allow programs to mix use of v1 and v2 as long as the handler registration routines in one major version forwarded to the other, so that there would effectively be only one `http.DefaultServeMux`.

Import Versioning may also resolve a key technical problem in defining automatic API updates. Before Go 1, we relied heavily on `go fix`, which users ran after updating to a new Go release and finding their programs no longer compiled. Updating code that doesn't compile makes it impossible to use most of our program analysis tools, which require that their inputs are valid programs. Also, we've wondered how to allow authors of packages outside the Go standard library to supply "fixes" specific to their own API updates. The ability to name and work with multiple incompatible versions of a package in a single program suggests a possible solution: if a v1 API function can be implemented as a wrapper around the v2 API, the wrapper implementation can double as the fix specification. For example, suppose v1 of an API has functions `EnableFoo` and `DisableFoo` and v2 replaces the pair with a single `SetFoo(enabled bool)`. After v2 is released, v1 can be implemented as a wrapper around v2:

```
package p // v1

import v2 "p/v2"

func EnableFoo() {
    //go:fix
    v2.SetFoo(true)
}

func DisableFoo() {
    //go:fix
    v2.SetFoo(false)
}
```

The special `//go:fix` comments would indicate to `go fix` that the wrapper body that follows should be inlined into the call site. Then running `go fix` would rewrite calls to v1 `EnableFoo` to v2 `SetFoo(true)`. The rewrite is easily specified and type-checked, since it is plain Go code. Even better, the rewrite is clearly safe: v1 `EnableFoo` is *already* calling v2 `SetFoo(true)`, so rewriting the call site plainly does not change the meaning of the program.

It is plausible that `go fix` might use symbolic execution to fix even the reverse API change, from a v1 with `SetFoo` to a v2 with `EnableFoo` and `DisableFoo`. The v1 `SetFoo` implementation could read:

```
package q // v1

import v2 "q/v2"

func SetFoo(enabled bool) {
    if enabled {
        //go:fix
        v2.EnableFoo()
    } else {
        //go:fix
        v2.DisableFoo()
    }
}
```

and then `go fix` would update `SetFoo(true)` to `EnableFoo()` and `SetFoo(false)` to `DisableFoo()`. This kind of fix would even apply to API updates within a single major version. For example, v1 could be deprecating (but keeping) `SetFoo` and introducing `EnableFoo` and `DisableFoo`. The same kind of fix would help callers move away from the deprecated API.

These examples demonstrate the power of having durable, immutable names attached to specific code behavior. We must simply follow the rule that when you make a change to something, you also change its name.

Last year, Rich Hickey made the point in his "Spec-ulation" talk that this approach of only adding new names and behaviors, never removing old names or redefining their meanings, is exactly what functional programming encourages with respect to individual variables or data structures. The functional approach brings benefits in clarity and predictability in small-scale programming, and the benefits are even larger when applied, as in the Import Versioning rule, to whole APIs: dependency hell is really just mutability hell writ large. That's just one small observation in the talk; the whole thing is worth watching.

I expect that the Import Versioning rule will be used in the final package management support in the go command itself. Go's official experiment for package management, *dep*, does not implement the Import Versioning rule yet, but I believe it should.

Acknowledgements

TODO.

[1] My favorite proof of this is given by Leslie Valiant in his landmark paper, "A Theory of the Learnable." Toward the end of the paper he observes that an implication of the theory he has established (for which he later won a Turing award) is that "[the] behavior of [a] program for unnatural inputs has no relevance. Hence thought experiments and logical arguments involving unnatural hypothetical situations may be meaningless activities."