# Adapting to a new version of GHC, way faster

DRAFT

Michael Peyton Jones, Ben Gamari, Simon Peyton Jones, Théophile “Hécate” Choutri

December 2021

*This document proposes a way to adapt to new GHC releases. It carries no authority; it’s just our thinking. Please help us improve it.*

*Note that this document has been superceded by* [*https://github.com/haskellfoundation/stability/pull/2/files*](https://github.com/haskellfoundation/stability/pull/2/files)

## 1. Motivation

At the moment we have a serious problem with propagating support for a new version of GHC through the Haskell ecosystem. Suppose package A depends on B which depends on C. Then the process goes like this

1. GHC version X is released.
2. The maintainer of C must wake up, fix their package, and make a Hackage release. The maintainers of A and B can do nothing at this point.
3. The maintainer of B must wake up, fix their package, and make a Hackage release. The maintainer of A can still do nothing.
4. The maintainer of A can finally fix their package and make a Hackage release.

This process is terrible in lots of ways

* It is utterly serial. The maintainer of B cannot lift a finger until the maintainer of C has not only fixed package C, but also uploaded a new release to Hackage.
* Each step has multiple serial parts: often the maintainer will merge a patch (perhaps in response to prompting) but *not* do a release, blocking further progress.
* If a maintainer is unavailable for any reason, the entire dependency tree of that package is blocked. In an ecosystem with hundreds of widely used packages, the chances of every single maintainter being available in a timely fashion are close to zero.
* It is not clear to even a willing and available maintainer *when* they need to wake up and do some work. Often it is up to a motivated individual (say the maintainer of an application that uses A) to sequentially bug the maintainers of C and B and A in sequence so they know that they are able to do something.

**The result of this is that it usually takes months or even years for support for a GHC version to percolate through the ecosystem.** For example, GHC 9.0.1 was released in February 2021, but as of December 2021 haskell-language-server still does not support it fully, largely because of difficulty updating dependencies.

We could enormously speed up the process of updating the package ecosystem for a new version of GHC if

* We could parallelise the process.
* We could allow many people to contribute to the routine updates necessary to adapt a package to changes in its dependencies.
* We could allow package maintainers to do their work when it was convenient to them, without holding up the whole train

We propose to partially improve the situation by widespread usage of a Hackage overlay (see Section 2), GHC.X.hackage, populated with non-maintainer-contributed patches for the purpose of unblocking package maintainers in adapting their library to new GHC releases.

## 2. Background

### 2.1 Additional package repositories and Hackage overlays

Cabal supports *additional* *package repositories* beyond Hackage. These can be configured in the Cabal config file, or in a cabal.project file.

Users can state which repositories they want to be *active*, and how they should be consulted. This allows additional repositories to override, or merely augment Hackage.

Overriding Hackage can be accomplished by listing the repositories in a Cabal config file, like so:

active-repositories: hackage.haskell.org, X

This will cause repository X to be consulted first for any given version of a package, but allow other versions to come from Hackage. For example, if X contains A-1 and A-2, and Hackage contains A-2 and A-3, then A-1 and A-2 will be taken from X, but A-3 can be taken from Hackage. ([Here is the documentation](https://cabal.readthedocs.io/en/3.6/cabal-project.html) for active-repositories; oddly, the list is searched last-to-first.)

We refer to this style of additional repository as a **Hackage overlay**, since it provides a way to *override* specificversions of packages which appear in Hackage, while using Hackage for everything else normally.

Additional package repositories behave well in the face of changes. Cabal will recompute the build plan whenever repositories are changed or updated; and since Cabal identifies packages by a hash which includes the hash of the source tarball, Cabal will never get confused between packages from different repositories, and it will cope gracefully even if package sources are *mutated* in the repository.

### 2.2 head.hackage

[head.hackage](https://gitlab.haskell.org/ghc/head.hackage/) is an example of a Hackage overlay. It is built from the linked source repository which consists of a set of patches to released Hackage versions of packages. head.hackage contains patches that enable packages to build with GHC head (hence the name), and is used by the GHC developers to test the upcoming GHC release.

## 3. Proposal

### 3.1 GHC.X.hackage

#### 3.1.1 The GHC.X.hackage package repository

For each released major version X of GHC, a Hackage overlay is provided called GHC.X.hackage. The intention of GHC.*X*.hackage is to override versions of packages that require changes to build with GHC *X*.

GHC.X.hackage is a *non-authoritative* set of patched packages, whose goal is primarily to unblock package maintainers. It should not be used in production, and any faults are not to be laid at the door of the package maintainers.

Since GHC.X.hackage is opt-in, users who do not use GHC.X.hackage will (obviously) not be able to use the packages which have been fixed in GHC.X.hackage, until fixed versions of those package and all their dependencies are actually released on Hackage.

Additionally, head.hackage becomes a pre-release version of GHC.X.hackage, and as such will change to behave in the same way as GHC.X.hackage.

When the release branch for GHC version *X* is created, the current state of the head.hackage repository is transitioned into GHC.*X*.hackage. A new head.hackage is created for the new GHC head.

GHC.X.hackage repositories continue to exist for old versions of GHC in perpetuity. They are not retired, although it is expected that they will be rendered redundant as proper releases are made in Hackage.

#### 3.1.2 Adding patches to GHC.X.hackage

The source of GHC.X.hackage is held in a Git repository, and accepts patches from community members *who need not be the package maintainer*. For example, the package maintainer of A might submit patches to fix B and C. This will use the same infrastructure as the existing head.hackage repository: here is an [example of a PR](https://gitlab.haskell.org/ghc/head.hackage/-/merge_requests/194) for head.hackage.

The inclusion criteria for a patch are:

1. The patched package should build with GHC X.
2. The patch should represent a patch-version level change according to the PVP, in particular it should not change the package’s API.
3. The patch should be plausibly acceptable by the upstream maintainer.

Criterion (3) is deliberately vague, but suggests that the tests should pass (although this will likely not be easy to verify given the diversity of packages), that the patch should not needlessly break compatibility with previous versions of GHC, etc. Of course, since in most cases the patch will *not* have been vetted by the maintainer, precise correctness cannot be guaranteed.

Package maintainers, or others, can update packages even *before* GHC X is released, by uploading patches to head.hackage. After the release, those patches will be part of GHC.X.hackage. The same rules for patch inclusion apply to head.hackage before it becomes GHC.X.hackage.

#### 3.1.3 Removing patches from GHC.X.hackage

Patches should not *need* to be removed from GHC.X.hackage. They will simply become obsolete as Hackage becomes more and more up-to-date.

However, in any case if a package maintainer requests the removal of a patch from GHC.X.hackage (for example, it might be outright wrong) then it should certainly be removed.

### 3.2 Versioning and releases

While GHC.X.hackage overrides particular versions of packages from Hackage, those overrides will quickly become obsolete when a new version of the package is released to Hackage.

Suppose that GHC.X.hackage contains a patched version of A-X. When A’s maintainer next releases A, they will release A-Y for Y>X. Users who use GHC.X.hackage will in most cases therefore find that cabal picks build plans with the new A-Y instead of the old (patched) A-X. However, if they have constraints that prevent this (say, because A-Y is a major version bump), then they may keep getting the GHC.X.hackage version.

It would be nice to argue that this case is unlikely: since by assumption we only allow patch-level changes into GHC.X.hackage, shouldn’t the next release of A be a patch-level release, and therefore unlikely to be forbidden by the user’s version constraints? But this requires an unrealistic picture of how maintainers work: it is more likely that the maintainer of A has some work-in-progress on A, to which they will *add* the GHC compatibility patches, to be included in the new release. So in general those changes might reach Hackage in any kind of release!

However, if users are stuck using the GHC.X.hackage versions of packages for longer than strictly necessary that is not too bad: the patches should be inoffensive, and the whole mechanism is opt-in regardless.

### 3.3 Package maintainers

Anyone submitting a patch to GHC.X.hackage should also submit a corresponding PR to the source repository of the package. (The PR will usually be identical to the GHC.X.hackage patch.) The maintainer can, *in their own time*, merge the PR and make a release to Hackage.

The only change to the work facing maintainers is that they are hopefully more likely to receive a good-quality patch fixing their compatibility problems for them.

### 3.4 Community engagement

The community should be encouraged to engage with GHC.X.hackage both before release (when it is head.hackage) and after release. One could imagine hackathons or other community events, with lists of packages to update, and celebrations when they are all done.

GHC.X.hackage should be announced as part of the GHC X release announcement.

Whenever GHC.X.hackage is mentioned, it must be clearly stated that it is strictly to be used for testing and compatibility work, since it contains patches that have not been vetted by the package maintainers.

## 4. Rationale

### 4.1 Speed

GHC.X.hackage enables significantly faster and more parallel fixing of packages.

Consider the example from the Motivation section. With GHC.X.hackage, the maintainer of A can, on the day of the GHC X release, fix both B and C in GHC.X.hackage, and use GHC.X.hackage to test and release a fix for A.

However, this speedup *only* materializes so long as the changes required in A’s dependencies are simple patch-level changes, since those are the only patches that GHC.X.hackage accepts. For everything else, we will still have to await a Hackage release. That may be problematic in practice, since many of the packages which will need major changes to support GHC X are down at the bottom of the dependency tree (e.g. because they make assumptions about GHC internals).

We will have to see in practice how much faster things get: perhaps GHC X will not be usable on day 1, but hopefully it will be usable significantly sooner than it would be otherwise.

### 4.2 Community involvement

Because GHC.X.hackage accepts patches from people who are not the package maintainers, it becomes possible for community members to get really involved. A single motivated individual can fix large swathes of the package ecosystem!

If we harness this well, we could potentially enable people to experiment with a mostly-working ecosystem via GHC.X.hackage in a few weeks, rather than months.

### 4.3 Maintainer awareness

GHC.X.hackage also partially resolves the problems of maintainers knowing when they are able to do compatibility work. The answer becomes: you can at least *try* as soon as GHC X is released.

This still isn’t ideal: many packages will be blocked on key blockers that require major changes (and hence Hackage releases), but the current proposal cannot fix that.

## 5. Implementation

Since head.hackage already exists, along with all the infrastructure to support it, we believe that it should be relatively straightforward to support multiple GHC.X.hackages, perhaps even with the same source repository (using branches).

There are a couple of roles that need to be filled on an ongoing basis

* Someone needs to maintain GHC.X.hackage. That is: vet patches for inclusion, merge them, keep the CI working, etc.
* If we want to get significant community involvement, it would be useful for someone to manage that involvement, a “community release manager” of sorts. That is: help people make contributions, drum up excitement, update the community on progress, help upstreaming patches, etc.

There’s a lot of flexibility in filling these roles. They could be

* Occupied by the same person, or different ones
* Rotating per release, or few releases
* Filled by a keen community member, or a paid professional

Currently, the GHC.X.hackage maintainer role is effectively filled by the GHC developers who maintain head.hackage. However, if the scope of the role expands, this is likely to be unsustainable. Potentially this is somewhere where the HF could provide assistance.

## 6. Alternatives and interactions

### 6.1 GHC Maintainer Preview

The [GHC Maintainer Preview proposal](https://github.com/ghc-proposals/ghc-proposals/pull/417) suggested cutting an early pre-release of new GHC versions in order to help maintainers update their packages before the “real” release. A frequent comment in the proposal discussion was that the Maintainer Previews would *not* actually be directly useful to most maintainers, because they would still be stuck waiting for their (many levels of) dependencies to update before they could do anything. It’s no good having a maintainer pre-release if maintainers can’t *use* it to fix their packages!

GHC.X.hackage alleviates this problem. Package maintainers could fix their packages and their dependencies by submitting patches to head.hackage, which would become GHC.X.hackage in due course.

### 6.2 What’s wrong with head.hackage?

The package upgrade workflow described here is occasionally done today with head.hackage. For example, [here is an attempt](https://github.com/haskell/haskell-language-server/pull/2503) to use head.hackage to pre-emptively fix up haskell-language-server for GHC 9.2.

So maybe we don’t need to do anything except

* Make one head.hackage for each version GHC version X
* Encourage more maintainers to use head.hackage
* Encourage more people to submit patches to head.hackage

But that’s essentially what the GHC.X.hackage proposal is, just with a little bit more structure, policy, and marketing.

### 6.3 Precognitive releases

The proposal as it stands does not address one of the major roadblocks to getting support out: updating packages that require major changes and hence Hackage releases.

It would be nice to do better than that. We could shoot for the following property:

**Desired Property**. An maintainer of package A can update and release to Hackage a new version of A, relying on GHC.X.hackage for A’s dependencies. *This release of A should continue to work when the maintainers of those dependencies make their releases to Hackage in due course.*

That is, we would like maintainers to be able to make *precognitive* Hackage releases based on how things *will* turn out in the future on Hackage.

A sketch of a solution could be:

1. Allow patches in GHC.X.hackage that make more than patch-level changes, and which must create *new* versions in GHC.X.hackage.
2. Encourage maintainers to make proactive Hackage releases based on the changes *and versions* in GHC.X.hackage.

For example, we might have a patch in GHC.X.hackage that fixes B-2 by creating B-2.1. The maintainer of A could then release a fixed version of A to Hackage, depending on B < 3.0, assuming that when B is finally updated on Hackage, the compatibility fixes will appear in B-2.1.

From this example it is clear that this system relies heavily on *assumptions* about how package maintainers release new versions of their packages. But this is quite problematic: as discussed in Section 3.2, package maintainers often have work-in-progress which they may want to release *with* the compatibility patches. So it is quite possible that the next release of B may be B-3, not B-2.1. At that point the maintainer of A has to do *another* release. That’s terrible - we’ve then made *more* work for the maintainer of A than they had before GHC.X.hackage existed!

It’s unclear whether a system like this can be made to work. In particular, it is not good enough for the assumptions to be *mostly* true: even if it *only sometimes* creates more work for maintainers, this is likely to be enough of a risk to put them off using it.

### 6.4 Tested with

It’s a bad user experience to compile a package with GHC X, and just get a compile error. The **tested-with** field of a cabal file helps: if present it specifies which versions of GHC this package has been tested with.

The goal should be that if A releases to Hackage a new version of package A, depending on (B < 1.6), but B has not yet released a new version to Hackage, then a naive user who is ignorant of GHC.X.hackage will get a message like

Package B has not yet been updated to GHC X

Consider using GHC.X.hackage (link)

Cabal could produce this message immediately from its solver, without compiling anything.

But perhaps B requires no updates at all to work with GHC X (this is a common case). Then this message would be over-conservative. Maybe Hackage could proactively set the tested-with field, by building the package and running its test suite? Or maybe we need two fields: a manual one and an automatic one.

# OLD TEXT

*MPJ: I’ve kept the text about versioning and package maintainers for reference, since it was by far the trickiest part of the old proposal.*

## 3. Proposal

### 3.2 Versioning

#### 3.2.1 The Desired Property of GHC.X.hackage

**Desired Property**. An maintainer of package A can update and release to Hackage a new version of A, relying on GHC.X.hackage for A’s dependencies. *This release of A should continue to work when the maintainers of those dependencies make their releases to Hackage in due course.*

Consider the example from the Motivation section. The idea is that A can make a final release without waiting for B or C. Later, when fixed versions of B and C are eventually released, A will *already* be fixed and won’t require any more changes.

The desired property sounds a bit strange: it asserts that things will continue to work after *future* package releases are made by maintainers. How can we possibly know that? Indeed, for GHC.X.hackage to work well it needs to predict the future. This relies on some assumptions about maintainer behaviour (see Section 3.3.2) and necessitates some careful thinking about versioning.

#### 3.2.2 Versioning: the simple case

The Desired Property (Section 3.2.1) relies on a precognitive assumption about versioning: we assume that the versions which appear in GHC.X.hackage will reflect those that finally appear on Hackage.

This is usually simple to achieve. For example, suppose that B’s current version is 1.5.3. Then A’s maintainer can *anticipate* that the patches for B in GHC.X.hackage will eventually be released to Hackage in a new patch version of B: 1.5.3.1. So A’s maintainer

* Offers a patch to GHC.X.hackage that fixes package B, bumping the version to B-1.5.3.1
* Makes a *final* release of A, to Hackage, with a dependency bound of B < 1.6. (Version bounds like this are standard practice, to allow B to make patch or minor releases without affecting A.)

Now when B’s maintainer eventually releases B-1.5.3.1 (or even 1.5.4) to Hackage, A will continue to build happily.

All this applies when the changes to B require only a patch or minor version bump: no semantic or API changes, just changes to adapt to a new version of GHC. If instead (for whatever reason) B’s maintainer released B-1.6, then A will *not* build and will require further work from the maintainer.

In most cases, we expect this not to be a problem. Since the changes required to support a new version of GHC are typically minor and internal (i.e. don’t change the API), the newly fixed version is likely to require only a *patch* version bump, which we can reliably predict in GHC.X.hackage (assuming that maintainers follow the PVP, see section 3.3.2).

#### 3.2.3 Versioning: the tricky case

There are a few cases where it is harder to predict what the new version is going to be:

1. The package may depend on GHC in such a way that being compatible with the new version requires an API change, and hence a new minor- or major-version.
2. The maintainer may have unreleased code changes which require a minor- or major-version bump, and will only release the compatibility patch alongside them.

It is difficult to ensure the Desired Property in this case, since if we try and guess what version to use for the fixed package in GHC.X.hackage, we might get it wrong! Continuing our example above: A’s maintainer released a new version of A with bound (B < 1.6); but if B is eventually released with version B-1.6, then A’s release won’t work with that new version of B. That means we lose the Desired Property: A’s maintainer will have to make another release with bound (B < 1.7).

For now, we assume that we will resolve such cases on a case-by-case basis, as we expect them to be rare. Typically we will need some involvement from the actual package maintainer at this point, as we do not have a way to systematically address these situations.

An alternative would be to simply not try and include such changes in GHC.X.hackage, and rather wait for the maintainer to push a fixed version to Hackage proper. So then GHC.X.hackage would contain minor-version-bumps only, and no API changes whatsoever. Especially given that we need to involve the maintainer anyway, and that we are at risk of guessing the API changes incorrectly, perhaps it is better not to try. However, GHC.X.hackage still offers some advantages in this scenario:

1. It avoids the need for the maintainer to do a Hackage release, with the effort and ceremony that might require (resolving any other issues blocking a release, updating changelogs, etc.). As discussed in the motivation, this can be a surprisingly significant roadblock!
2. It allows the rapid release of a “good enough” patch. The patch for GHC.X.hackage might have minor correctness issues (or simply not have been as rigorously tested as the maintainer would like before a proper release), but have the correct API. This is enough to unblock downstream compatibility fixes, and as such could be acceptable for GHC.X.hackage, if not for Hackage.
3. In the case of head.hackage (i.e. before GHC X is released), some maintainers may be uncomfortable making compatibility releases for unreleased versions, but be okay with including them in head.hackage.

#### 3.2.4 Multiple versions of a single package

Sometimes there are multiple versions of a popular package P *in active use*, say P-2.1, P-2.2, and P-3.0.

GHC.X.hackage can contain patches for all three versions. But perhaps the package maintainer has moved on, and is now mainly focused on P-3.0, and hence is unwilling to devote time to making new releases of P-2.1 and P-2.2 (we do not wish to assume that they will, see Section 3.3.2). In this case we lose the Desired Property, since the patched P-2.1 that GHC.X.hackage provides will never materialize on Hackage proper.

For now we leave such cases unsolved: they are going to cause trouble in any case, since the maintainer is never going to publish a fixed version of P-2.1 no matter what we do! Conceivably we could extend the Hackage trustee system so that compatibility releases of “abandoned” minor versions could be made by trustees. But this would be a major incursion into the purview of maintainers and is likely not worth it just to tackle these (hopefully rare) cases.

### 3.3 Package maintainers

#### 3.3.1 The package maintainer’s work

Anyone submitting a patch to GHC.X.hackage should also submit a corresponding PR to the source repository of the package. (The PR will usually be identical to the GHC.X.hackage patch.) The maintainer can, *in their own time*, merge the PR and make a release to Hackage.

Crucially, if we manage to attain the Desired Property, then this single Hackage release is *all* that the package maintainer needs to do to support GHC X!

The Hackage release will naturally supersede the changes in GHC.X.hackage, since GHC.X.hackage should be set up so that Hackage versions are chosen preferentially: GHC.X.hackage is a Hackage *augment* (see Section 2.1). For example, suppose the current Hackage release of package B is 1.5.3. The GHC.X.hackage repository provides a patched version B-1.5.4. *But as soon as the package maintainer releases B-1.5.4 to Hackage, Cabal will choose it, and the patched version of B-1.5.4 becomes redundant.*

Maintainers of packages that depend on GHC internals may be called upon to bless the inclusion of more significant patches into GHC.X.hackage (see point (3) in Section 3.1.2).

#### 3.3.2 Assumptions about maintainers

This proposal relies significantly on *predicting* the future behaviour of maintainers, specifically how they will respond to upstream changes in GHC. In particular, we assume the following:

1. Maintainers comply with the PVP when making releases
2. Maintainers want to have *at least one* released version of their package that is compatible with the latest GHC
3. Maintainers are willing to make the effort to release a patch or minor version of the *latest* major version of their package in the pursuit of compatibility
4. Maintainers *may* *be* willing to make the effort to release a major version of their package in the pursuit of compatibility
5. Maintainers will be comfortable with non-authoritative patches to their packages being distributed via GHC.X.hackage

We believe these are reasonable assumptions to make globally, but we have tried to keep them as minimal as possible. Note that these are *descriptive* assumptions, not *prescriptive*. We are not trying to change how people behave, but rather stating what we think needs to be true about their current behaviour for this to work.

The most troubling assumption is (5). It is quite conceivable that maintainers could take umbrage at patched versions of their packages being distributed in this way (witness the reaction that some people have to Hackage metadata revisions, which don’t even touch the package source). For this reason we believe we must take great care in the messaging around GHC.X.hackage, perhaps even choosing a different name that emphasizes its non-authoritative nature.

==========================

**Question**. The [PVP](https://pvp.haskell.org/) specifies four version numbers in fact (well any number really). Should we encourage contributors making a minor GHC-X-compatibility change to bump the fourth number, or the third? **Answer**: Probably the fourth. That is, put B-1.5.3.1 in GHC.X.hackage, because that works whether B’s author eventually chooses to release B-1.5.3.1 or B-1.5.4.

Question. Suppose I have installed B-1.5.4 from GHC.X.hackage. Now B’s maintainer makes a release of B-1.5.4 to Hackage. When Cabal looks for a dependency B, and decides that B-1.5.4 is the right version, will it notice that (a) Hackage has B-1.5.4 and (b) the installed version came from GHC.X.hackage? If not, it will use the obsolete, installed B-1.5.4, which came from GHC.X.hackage. **Answer:** I believe that both behaviors can be achieved by configuration (specifically, <https://cabal.readthedocs.io/en/3.4/cabal-project.html?highlight=repository#cfg-field-active-repositories>), although this is all very under-documented so I’ll need to consult with our Cabal experts).

**Related question**. What if someone installs B-1.5.3.1 from GHC.X.hackage, and then someone changes B-1.5.4 with a new patch to GHC.X.hackage, how can Cabal know to update the installed package. **Answer**: just like Hackage, never change the contents of a particular package in GHC.X.hackage; use B-1.5.3.2; or even B-1.5.3.1.1. But even this risks B’s maintainer releasing B-1.5.3.1, which does *not* obsolete the GHC.X.hackage version. Better answer: use timestamps; but this would require more from Cabal.

Question. Who should use GHC.X.Hackage and when? As a maintainer, do I configure CI to always use it? Do I remove it from active-repositories once it's not needed anymore? How do I remember to verify that my package works against normal hackage? As an end-user, can I configure it to be used only for certain projects?