Compatibility with dependencies

Tobias Stolzmann

July 14, 2018

1 Motivation

I've followed the discussions around backward-compatibility, minimal `rustc` versions and LTS releases for quite some time now. I believe, it lacks a strong theoretical background. Therefore, I've decided to do some math.

2 Notation and definitions

In order to go ahead, we need to fix some notation and terminology.

2.1 Crates and versions

$A, B, C, \ldots$ will denote crates like `rustc` and `serde`.

$v_A, v'_A, v_B, \ldots$ will denote versions like `1.27.1` and `1.0.70`. Versions are totally ordered and can thus be compared.

$A : v_A, B : v_B, \ldots$ will denote crates in a specific version (*versioned crates*) like `rustc:1.27.1` and `serde:1.0.70`. We will identify versioned crates with sets of public features.

2.2 Compatibility

A versioned crate $A : v_A$ is *compatible* with another crate $B : v_B$, iff

$$A : v_A \supseteq B : v_B \tag{1}$$

That means, every feature provided by $B : v_B$ is also provided by $A : v_A$.

A versioned crate $A : v_A$ is called *backward-compatible* with $B : v_B$, iff $A : v_A$ is compatible with $B : v_B$ and $v_A \geq v_B$.

A versioned crate $A : v_A$ is called *forward-compatible* with $B : v_B$, iff $A : v_A$ is compatible with $B : v_B$ and $v_A \leq v_B$.

Backward-compatibility allows feature addition while forward-compatibility allows feature removal.

2.3 Dependency

$A : v_A \to B : v_B$ denotes a (public) dependency. It means, that crate A in version v_A depends on crate B in version v_B. Public means, that $A : v_A$ re-exports at least some of the features provided by $B : v_B$. (To make things easier, we won't deal with private dependencies for now.)

3 Semantic versioning

A *semantic version* as described in [?] is a 3-tuple `major.minor.patch` that make up a version. Semantic versioning provides some compatibility guaranties:

Let A be a crate with two semantic versions v_A and v'_A with $v_A \geq v'_A$.

Given that $major(v_A) = major(v'_A)$, semantic versioning requires that $A : v_A$ is backward-compatible with $A : v'_A$.

Given that $major(v_A) = major(v'_A)$ and $minor(v_A) = minor(v'_A)$, semantic versioning requires that $A : v'_A$ is also forward-compatible with $A : v_A$. That essentially means that $A : v_A = A : v'_A$.

Exeptions exist, especially for $major(v) = 0$. We won't tackle them for now.

4 Compatibility with dependency

If a crate needs to be compatible, this imposes some requirements on the public dependencies.

Let $A : v_A$ be compatible with $A : v'_A$. Let also $A : v'_A \to B : v'_B$. In that case, we need that

1. $A : v_A \to B : v_A$ and
2. $B : v_B$ is compatible with $B : v'_B$.

5 Compatibility with dependency in case of semantic versions

This section is limited to minor version changes, since

1. We can say nothing about major version changes since they to not guarantee any compatibility.
2. Patch version changes are trivial, since they do not allow feature changes at all.

Let A be a crate with two semantic versions v_A and v'_A with $v_A \geq v'_A$ and $major(v_A) = major(v'_A)$. Let $A : v'_A \to B : v'_B$. We may conclude that $A : v_A$ is backward-compatible with $A : v'_A$ and thus we need

1. $A : v_A \to B : v_A$ and
2. $B : v_B$ is compatible with $B : v'_B$.

If v_B and v'_B are semantic versions with $v_B \geq v'_B$ and $major(v_B) = major(v'_B)$, the second requirement is automatically satisfied since $B : v_B$ needs to be backward-compatible with $B : v'_B$.

Since $B : v'_B$ is backward-compatible with itself and any version v_B with $v_B \leq \langle major(v'_B + 1), 0, 0\rangle$ is backward-compatible with $B : v'_B$, we may write

$$A : v_A \to B : [v'_B, \langle major(v'_B + 1), 0, 0\rangle)$$

instead of both requirements above. The notation means that $A : v_A$ must depend on B in any version between v'_B (inclusive) and $\langle major(v'_B + 1), 0, 0\rangle$ (exclusive). The crate author may decide which version is appropriate.

6 The crux with minimal versions

Let's take the example from section 5 even further.

First, we assume that there is no possibility to have B in more than one version. Second, we say there is a crate $C : v'_C$ with $C : v'_C \to B : v'_B$ which is **forward-**compatible with $C : v_C$.

We may conclude that there is no possibility to use both $A : v_A$ and $C : v_C$ at the same time other that stick to $B : v'_B$. Hence, in order to maximize the versions of A and C, we need to minimize the version of B.

In that case, the implications for A are dramatic since we may conclude that

$$A : [v'_A, \langle major(v'_A + 1), 0, 0\rangle) \to B : v'_B$$

which means that $B : v'_B$ is a dependency for A until we increase the major version of A.

In order to conclude this section, I would like to indroduce the general syntax

$$A : [\check{v}_A, \hat{v}_A) \to B : [\check{v}_B, \hat{v}_B)$$

which means that each version v_A of A with $\check{v}_A \leq v_A < \hat{v}_A$ needs to depend on any version v_B of B with $\check{v}_B \leq v_B < \hat{v}_B$. Other types of interval boundaries work accordingly.

7 A real work example

`serde:1.0.70` depends on `rustc:1.13` or more. Formally, we may write `serde:1.0.70 -> rustc:[1.13, +inf)`. From the math done above, we can derive that any version of `serde` greater than or equal to `1.0.70` needs to depend on `rustc` between `1.13` (inclusive) and `2` (exclusive). Formally, we may write `serde:[1.0.70, 2) -> rustc:[1.13, 2)`. Please note, that our math does also limit the maximum version of `rustc` for `serde:1.0.70` which seems plausible but was not initally required by us.

Let's now introduce a pseudo-crate called `system:1` which represents a user's system configuration. `system:1` depends on `rustc:1.27.1`. `system:1` should be forward-compatible with future version of `system`, since the user does not want to update the configuration each time she updates her software. In that case, we derive `serde:[1.0.70, 2) -> rustc:[1.13, 1.27.1)` if `serde` wants to stay compatible with that particular user.

8 Conclusion

I've introduced a formal definition for compatibility and dependency between crates.

I've analyzed the formal implications of combining compatibility, dependency and semantic versioning with these results:

1. Future crate versions that maintain backward-compatibility require dependencies to persist with the possibility to update.

2. The combination of backward- and forward-compatibility introduces upper and lower version boundaries for dependencies. Since forward-compatibility constrains naturally come from the users' system configurations, crate authors need to stick to a (minimal) dependency version if once selected.

I am aware that my theorization comes in support to the so-called "convervative approach" which suggests that a crate author may never change a dependency without changing the major version. Two remarks:

1. This does not hold for (private) dependencies that do not contribute to the API of a crate. Therefore, I suggest to add a destinction between public and private dependencies to `cargo`.

2. I have ideas in the pipe that may justify updating public dependencies without changing the major version. These ideas advertise the addition of meta-versions like `stable` that translate to a specific version at some time t' while simultaneously announces that it might translate to a greater version at some time t with $t' < t$. I believe, these ideas provide a elegant solution to most of the use cases of LTS versions and thus being a strong alternative. I hope, I will be able to write them down soon.