Bundling targets by macro

- **Authors:** brandjon@bazel.build
- **Status:** Draft
- **Approver:** comius@
- **Reviewers:** tetromino@, susinmotion@, danmullow@, murali42@, lberki@
- **Created:** 2025-01-30
- **Updated:** 2025-02-01
- **Discussion:** #25168

Please read Bazel Code of Conduct before commenting.

Overview

This doc proposes an extension to the visibility system for symbolic macros. Essentially, it makes it so that if `foo` is the name of a macro instance, then passing `foo` to another macro will allow the callee to see any of `foo`'s exported targets. This is a relaxation of the current behavior, which is that the callee can see `foo`'s main target but not auxiliary targets like `foo_aux`.

Allowing this kind of access is necessary for macros whose targets are accessed indirectly by munging their main target's name. This removes a pain point from using symbolic macros, and makes migration from legacy macros more feasible.

This doc also outlines two rejected alternatives.

Problem

To illustrate the current state, suppose we have a macro, `shadow_library`, that wraps `cc_library` and that also creates a shadow graph of auxiliary targets mirroring its `deps` structure.

```
# //shadow_library/defs.bzl

def _shadow_library_impl(name, visibility, deps):
    # Main target ("foo"), exported.
    cc_library(
        name = name,
        visibility = visibility,
        deps = deps,
    )
    # Auxiliary target (“foo_aux”), also exported as a public api.
    cc_library(
        name = "%s_aux" % name,
        visibility = visibility,
        deps = ["%s_aux" % d for d in deps],
    )


shadow_library = macro(…)
```

Now suppose we have a second macro that interoperates with `shadow_library` but which adds new functionality:

```
# //extension_library/defs.bzl

load(“//shadow_library:defs.bzl”, “shadow_library”)

def _extension_library_impl(name, visibility, deps):
    # Produces “foo” and “foo_aux” as before.
    shadow_library(
        name = name,
        visibility = visibility,
        deps = deps,
    )
    # Also produces “foo_ext” and “foo_aux_ext” that depend on the deps’ main and
    # auxiliary targets, respectively.
    cc_library(
        name = “%s_ext" % name,
        visibility = visibility,
        deps = deps,
    )
    cc_library(
        name = “%s_aux_ext” % name,
        visibility = visibility,
        deps = [“%s_aux” % d for d in deps],
    )

extension_library = macro(…)
```

Finally, in the BUILD file, we have:

```
# //pkg/BUILD

load(“//shadow_library:defs.bzl”, “shadow_library”)
load(“//extension_library:defs.bzl”, “extension_library”)

# All targets are private to //pkg.
```

```
shadow_library(
    name = “A”,
)

shadow_library(
    name = “B”,
    deps = [“:A”],
)

extension_library(
    name = “C”,
    deps = [“:A”],
)
```

The two dependency edges `B -> A` and `B_aux -> A_aux` are allowed because all four targets are declared in the same place, `//shadow_library`. This even applies to the two edges `C -> A` and `C_aux -> A_aux`, despite the added intermediary macro.

The same can't be said for the edge `C_ext -> A`, since the consuming location `//extension_library` does not match `A`'s actual visibility of `{//pkg, //shadow_library}`.[1] But thanks to *visibility delegation*, this dependency is allowed anyway since the caller (`//pkg`) passed `A` in as an attribute value and the caller is itself permitted to see `A`.

Yet delegation doesn't help for the last edge, `C_aux_ext -> A_aux`. Morally, the permission to see `A_aux` should have also been delegated by passing in `A` to `extension_library`. But the visibility system doesn't recognize that A_aux is associated with A. This was always a known weakness in the design of Macro-Aware Visibility, one that we now aim to correct; we are seeking a method for *bundling* `A_aux` with `A`.

There are two important constraints on any solution we choose.

1. A bundling mechanism must uphold visibility protection. This means that either we need to validate `A_aux`'s visibility against the caller's location at analysis time, or else we need to ensure by construction that `A_aux`'s visibility is a superset of `A`'s visibility. (I say "ensure", because actually validating it isn't possible in the loading phase since package groups aren't expanded.)

2. We want our design to add no new skyframe edges. This means that we don't want to load `A`'s configured target as part of the processing of `A_aux`'s visibility check. It also means we don't want target declarations to require information from any other target declaration in a not-currently-loaded symbolic macro.

[1] A target's actual visibility is the visibility passed in, unioned with the target's declaration location.

Proposal

A new visibility atom, `//visibility:exported`, is introduced. When it is used in a `visibility` attribute of a target or macro instance, it is immediately expanded by replacing it with the visibility of the macro that the declaration appears within, or by the package's `default_visibility` if the declaration is not inside any symbolic macro. (The new atom never appears as an actual value in a `visibility` attribute.) In addition, the target or macro is conceptually marked as exported.

The new delegation semantics are that passing a macro around as a label will delegate all exported targets of the macro that the caller has visibility on. This applies transitively through exported submacros as well. By only delegating exported targets, we avoid inadvertently exposing implementation details when the caller happens to be in the same package as the macro's definition. By only delegating targets the caller can see, we avoid violating visibility in situations where the main target is visible to places that the auxiliary target is not.

<TODO: Work out exact error behavior. Is it legal if the caller can see foo_ext but not foo? Is it an error if the caller can't see either but the callee has independent visibility permission over foo_ext?>

Within a macro implementation function, the `visibility` parameter's behavior is modified to always take on the value `["//visibility:exported"]`. This ensures that the pattern of declaring exported targets with `visibility = visibility` continues to work while opting into the new export behavior. It removes the ability for macros to infer what macro called them by inspecting the visibility attribute, but this was an antipattern anyway. It should not break the vast majority of macros.

Since the value of the `visibility` attribute is now a constant, we can consider making it optional in the macro's signature, omitting it if not present as a keyword argument (not counting `**kwargs`). It could even be deprecated and removed entirely, although this would become yet another minor difference between the bodies of symbolic and legacy macros.

Although it is discouraged to use `//visibility:public` within a macro, assigning a target this visibility level will also mark it as exported for the purposes of delegation. This preserves `//visibility:public`'s position at the top of the visibility lattice.

<TODO: Will this interact poorly with tooling that introspects target visibilities via blaze query? That is, by expanding away //visibility:exported from source code in .bzl files?>

<TODO: It seemed like we were so close to a definition of delegation that didn't change depending on the caller's location. Is there a way to make the set of delegated targets constant and only check whether the caller can see the main target?>

Application to `shadow_library` example

The definitions of the two macros are unchanged. The example "just works"; the edge `C_aux_ext -> A_aux` is now permitted because the `BUILD` file passed in `A`, `A_aux` is exported from `A`, and `//pkg` can see `A_aux`.

Application to legacy macros

Targets are bundled only by appearing in a common symbolic macro. Nonetheless, targets of a legacy macro can be bundled under new names by defining `alias` targets for them underneath a symbolic macro instance. In fact, the process can be streamlined by defining a single `alias_group` symbolic macro that can be instantiated as needed.

This does require enumerating each target to be bundled. There is no way to alias an entire macro for this purpose, since bundling is a loading-phase concept and alias resolution requires the analysis phase.

Alternatives

We considered two main alternatives before this one. The first is based on fine-grained control over which targets are eligible for delegation. Ultimately, that approach did not generalize well when rules are substituted by macros. The second is a simpler form of the proposal, where delegation is maximally applied within a macro without any need for a new visibility atom. But this leaks a macro's implementation details a bit too easily.

Explicit declarations of bundling

Under this approach, a new universal attribute is defined, called `bundled_with`. Its type is a no-dep label that must refer to another target declared in the current symbolic macro instance. When the declaration of `B` says that it is ``bundled_with = A``, the effect is that 1) `B` is delegated whenever `A` is passed to a macro, and 2) `A`'s visibility is implicitly unioned to `B`'s visibility, so that `B` is guaranteed to be visible anywhere `A` is.[2] (This unioning is done in the loading phase, hence why `A` must be in the same symbolic macro as `B`.)

Bundling is transitive: If `C` has ``bundled_with = B`` then `C` can be delegated by passing either `A` or `B`. Bundling does not follow `alias` targets.

This approach is certainly powerful enough to address the `shadow_library` use case. It can also work with legacy macros since the bundling relationship is explicit rather than inferred by macro call structure. But there are several large drawbacks:

- It requires a new universal attribute name for all targets.

[2] An earlier version inverted the relationship so that `A` would declare that it ``bundles = [B]``. This was very bad for self-contained definitions and for respecting internal targets' visibility.

- It creates a distinction between bundling a macro target vs merely exporting it. Contrast this with the above proposal, where the distinction between being exported vs merely being visible to the caller is unlikely to come up in practice.

- It's unclear how to generalize `bundled_with` to macros.
    - If you set ``bundled_with = foo`` on a target, where `foo` is a macro instance, is that referencing the main target of the macro? (If so, we're possibly creating a loading-phase dependency on evaluating `foo`'s macro to get the main target's visibility.)
    - If you set `bundled_with` *on* a macro called `foo`, is that modifying `foo`'s main target / submacro?
    - If `bundled_with` is not supported on macros, then it becomes a hazard for any rule author who wants to refactor their rule into a macro.
    - What if you want a wrapping macro to extend a different bundle in the submacro than the one that contains its main target? (How would you even refer to other targets without evaluating the macro?)

The difficulties of applying this approach to macros led us to believe that it may be a mistake to let macros have more than one bundle. It just doesn't seem feasible to answer the above questions in a consistent way, while satisfying our requirements concerning loading phase dependencies and visibility safety. At the very least, it would be very confusing to explain to users.

But if there can only be one bundle per macro, then it makes sense to default into that bundle without an extra universal attribute.

Implicit bundling

The second idea was to allow a label referencing a macro's name to delegate any target transitively declared by the macro, so long as the caller could see that target. Effectively, this is the solution we proposed above but without an additional `//visibility:exported` atom.

There are several problems with this approach, but only one that differentiates this from the above proposal: Any caller that happens to be in the same package as the macro has visibility on all internal targets of the macro, and therefore delegates all those targets to the callee. This effectively defeats visibility anywhere that the macro's own package passes the main target around.

The other problems that are shared with the proposed solution are that the caller may accidentally delegate some exported targets that it didn't intend to, and that it's not always obvious which targets are and aren't delegated at a given call site.

Compatibility

Changing the value of the `visibility` parameter will break symbolic macros that introspect it in any non-trivial way.

Making the visibility parameter optional, and not passed if there is only a `**kwargs` but no separate keyword parameter, is a breaking change to any macro that expected it to be included in `**kwargs`.

Having a `//visibility:exported` atom that gets expanded when used, may break tooling and use cases that introspect the `visibility` attribute and feed its value back into target declarations.

Document History

Date	**Description**
2025-02-01	Initial version