[SBC] Parameterized Transitions API

go/sbc-transitions
Author: juliexxia@google.com
Last updated: Dec 07, 2018 - added 'The Final Proposal'
Reviewers: gregce, cparsons
Status: LGTM'd and being implemented
Extension of Starlark Build Configuration
https://github.com/bazelbuild/bazel/issues/5574

This is a publicly readable document

Objective

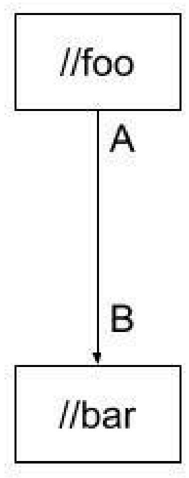
A major end goal of the Starlark Build Configuration effort is to make configurability features no longer a blocker for migrating native rules to starlark. For the purpose of this document, we will only discuss transition features that must be exposed to starlark for native rule migration. The goals of this doc are:

1. Agree on a powerful but simple transition API that supports all attribute-parameterized transition features used by native rules today
2. *[Optional]* Simplify the native transition API by combining features that are similar

Background

Native Transition Factories

For more details, see Appendix A: Native Configuration Calls



SplitTransitionProvider - `SplitTransition apply(AttributeMap attributeMap)`

- Attributes: //foo, **configured**
- Attached at: A

RuleTransitionFactory - `PatchTransition buildTransitionFor(Rule rule)`

- Attributes: //bar, unconfigured (because just receives Rule object)
- Attached at: B
- Can't be split transition
- Currently gets entire rule but in practice only reads attributes* so we should be able to simplify to only take an attribute map.

*Exception for config feature flag transitions (which will remain native)

Current Attribute-Parameterized Transition Design

Transitions are defined using the starlark `transition` function. This includes defining an implementation function for the transition:

myapp/transitions.bzl:

```
def _compile_for_android_transition(settings, attr):
  return {
    "//myapp:my_flag": settings["//myapp:my_flag"] + "_android",
    "//tools/cpp:crosstool_top": settings["//tools/android:android_crosstool_top"]
  }

compile_for_android_transition = transition(
  implementation = _compile_for_android_transition_impl,
  inputs = ["//tools/android:android_crosstool_top",  "//myapp:my_flag"],
  outputs = ["//tools/cpp:crosstool_top",  "//myapp:my_flag"]
)
```

Once defined, transitions are attached either onto an attribute or a rule itself using the `cfg` module. For more details, see the original design.

General summary of the state of world re: transitions in the original proposal

- We have two implementation function signatures for defining new starlark transitions; one for regular transitions and one for attribute-parameterized transitions. These can both be attached either directly to a rule (rule class transition) or to an attribute (attribute transition):
  ```
  def _my_transition_impl(settings)
  def _my_transition_impl(settings, attrs)
  ```
- For attribute-parameterized transitions, the values of the attribute map depend on where the transition is attached.
  - If the transition is a rule class transition (attached directly to a rule), the attribute values are unconfigured
  - If the transition is an attribute transition (attached to an attribute), the attribute values are configured
- Rule class transitions can only be 1->1 transitions while attribute transitions can be either 1->1 transitions or split transitions.

Final Proposal

Differences from the original proposal:

- The implementation function for Starlark transition creation will **always** take two parameters: `settings` and `attrs`. Non-parameterized transitions will just not use the `attrs` param.
- The `attrs` parameter, a map of attribute names -> values, will **always** contain **configured** values i.e. values with selects resolved.

The main advantage this proposal has over the other proposals in this document is the unified transition implementation function signature. The `attrs` parameter will always be present and will always contain configured values.

In order to make the second change work, we need to introduce restrictions for starlark rules that use parameterized rule class transitions. Parameterized rule class transitions cannot depend on attribute values that proceed to depend on configuration (i.e. through a select) since this would introduce a configuration->attribute->configuration dependency cycle.

To deal with this, we'll throw runtime errors for transition implementation functions that create these dependency cycles. The specific case we're interested in is when (1) a build setting* is **declared written** by a transition at any point** on the incoming edge to a rule-transitioned target, (2) an incoming rule transition transition reads an attribute of the target, and (3) that target configures the same attribute using the same build setting. If this happens, we'll throw an error when we try to access that attribute in the transition implementation function. This all gets a bit hairy, see the examples below for a more concrete example.

*for V1, disallow any build settings that set other build settings to prevent having to do more cycle checking

** “at any point” refers to via an outgoing edge transition or an incoming rule transition which can both happen on the same edge. We compose transitions that exist on the same edge.

A simple example - a parameterized rule class transition

//my_app/rules.bzl

```
# transition reads the "bool" attr
_transition_impl(settings, attr):
    if attr.bool:
        return { "//my_flag": "foo" }
    else:
        return { "//my_flag": "bar" }

# declaring that we will write to //myflag
my_transition = transition(_transition_impl, inputs = [], outputs = ["//my_flag"])

_rule_impl(ctx):
  …

my_rule = rule(
    implementation = _rule_impl,
    cfg = my_transition,
    attrs = {
        "bool": attr.bool()
    }
)
```

//my_app/BUILD

```
load("//my_app:rules.bzl", "my_rule")

config_setting(
  name = "my_flag_setting",
  values = { "//my_flag": “foo”)
)

# throws an error because bool selects on a flag that may be changed by
my_transition above (and my_transition reads bool)
my_rule(
  name = "my_rule",
  bool = select({
    ":my_flag_setting": True,
    "//conditions:default": False
  })
)
```

A more complex example - a composed transition

In this example, an outgoing edge transition that reads the `bool` attribute interferes with a select on the depended on target.

//my_app/rules.bzl

```
_rule_impl(ctx):
    …

# we write //my_flag
_attr_transition_impl(settings, attr):
    return { "//my_flag": "foo" }

attr_transition = transition(
    _attr_transition_impl,
    inputs = [],
    outputs = ["//my_flag"]
)

# we apply this transition on an outgoing edge
parent_rule = rule(
    implementation = _rule_impl,
    attrs = {
        "deps": attr.label_list(cfg = attr_transition),
    },
)

_rule_transition_impl(settings, attr):
    if attr.bool:
        return { "//my_other_flag": "foo" }
    else:
        return { "//my_other_flag": "bar" }

rule_transition = transition(
    _rule_transition_impl,
    inputs = [],
    outputs = ["//my_other_flag"]
)

# we have a rule class transition on the incoming edge that reads attr.bool
```

```
dep_rule = rule(
    implementation = _rule_impl,
    cfg = rule_transition,
    attrs = {
        "srcs": attr.string_list()
    }
)
```

//my_app/BUILD

```
load("//my_app:rules.bzl", "parent_rule", "dep_rule")

parent_rule(
    name = "parent",
    deps = [":dep"],
)

# throws an error because bool selects on a flag that may be changed by
attr_transition above -> rule_transition gets parent_rule's value of bool not
dep_rule's value.
dep_rule(
    name = "dep",
    bool = select({
        ":my_flag_setting": True,
        "//conditions:default": False
    })
)

config_setting(
    name = "my_flag_setting",
    values = { "//my_flag": "foo")
)
```

Potential Designs

As seen in the original design, the transition implementation method has two possible signatures:

```
  def _my_transition_impl(settings)
  def _my_transition_impl(settings, attrs)
```

Here we examine how the second signature could be expanded to account for the different types of transition factories detailed above.

(I) Explicitly declare transitions as attribute-parameterized

Add a parameter, `applied_to`, to transition creation that signals where the transition will be attached and therefore what kind of attributes the transition receives. The value of this attribute can be either `“target”` or `“dep”`.

It is an error to write an implementation function with a `attrs` parameter without the `applied_to` attribute set.

RuleTransitionFactory -> `target` attrs provided

```
# attrs = unconfigured attrs of target applied to
# Cannot return a dict of dicts (i.e. be a split transition)
def _my_transition_impl(setting, attrs):
  …

# must be applied as a rule class transition, applying elsewhere throws an error
my_transition = transition(
  impl = _my_transition_impl,
  applied_to = "target"
  inputs = ["//tools/android:android_crosstool_top",  "//myapp:my_flag"],
  outputs = ["//tools/cpp:crosstool_top",  "//myapp:my_flag"]
)
```

If the `factory` attribute is set to `“target”`, it is an error for the implementation function to return a dict of dicts.

SplitTransitionProvider -> `dep` attrs provided

```
# attr = configured attrs of target applied to
def _my_transition_impl(setting, attrs):
  …

# must be applied on a dependency edge
my_transition = transition(
  impl = _my_transition_impl,
  applied_to = "dep"
  inputs = ["//tools/android:android_crosstool_top",  "//myapp:my_flag"],
  outputs = ["//tools/cpp:crosstool_top",  "//myapp:my_flag"]
)
```

Pros:

- Forces rule writers to be explicit about what kind of parameterized transition they’re trying to write

- Explicitly differentiates the two types of parameterized transitions

Cons:

- Odd to have a parameter mean different things based on an attr that is set in a different function
- Makes initializing transitions more verbose
- Doesn't clarify difference between configured and unconfigured attr values

(Ia) Explicitly declare transitions as attribute-parameterized + transition implementation function signature change [added Nov 12]

Turn the positional unnamed `attrs` parameter of the transition implementation function into a keyword argument. If this transition is declared a `rule`-parameterized transition, the keyword parameter is `rule_attrs`. If the transition is declared a `dep`-parameterized transition, the keyword parameter is `dep_attrs`.

```
# attrs = unconfigured attrs of target applied to
# Cannot return a dict of dicts (i.e. be a split transition)
def _my_transition_impl(settings, rule_attrs):
  …

# must be applied as a rule class transition, applying elsewhere throws an error
my_rule_transition = transition(
  impl = _my_transition_impl
  applied_to = "target"
  inputs = ["//tools/android:android_crosstool_top",  "//myapp:my_flag"],
  outputs = ["//tools/cpp:crosstool_top",  "//myapp:my_flag"]
)

# attr = configured attrs of target applied to
def _my_transition_impl(setting, dep_attrs):
  …

# must be applied on a dependency edge
my_dep_transition = transition(
  impl = _my_transition_impl
  applied_to = "dep"
  inputs = ["//tools/android:android_crosstool_top",  "//myapp:my_flag"],
  outputs = ["//tools/cpp:crosstool_top",  "//myapp:my_flag"]
)
```

Pros:

- Makes implementation functions also explicitly typed as one kind of parameterized transition

Cons:

- different looking signature from other implementation functions

(II) Infer type of transition factory based on attachment site

If the transition is applied directly to a rule class, `attrs` is the unconfigured attributes of the target applied to. If the transition is applied on a dependency edge, `attrs` is the configured attributes of the rule initiating the transition.

```
# If this transition is applied to a rule, attr == unconfigured attrs of target
applied to
# if this transition is applied to a dependency, attr == configured attrs of target
applied to
def _my_transition_impl(setting, attrs)
```

Pros:

- No added complexity to transition function

Cons:

- Difference between factories very hidden -> probably easy to do the wrong thing as a rule writer
- Doesn’t clarify difference between configured and unconfigured attr values

Probably Bad Ideas

(III) Always have two `attrs` parameters, but only have one be non-null

Slightly less confusing than inferring attrs’ contents but still “black magic” element of not clearly knowing which one will be non-null. Also strange to always have a parameter be null.

```
# If this transition is applied to a rule class,
#    rule_attrs == unconfigured attrs of target applied to
#    dep_attrs == null
# if this transition is applied to a dependency,
#    rule_attrs == null
#    dep_attrs == configured attrs of target initializing
def _my_transition_impl(settings, rule_attrs, dep_attrs)
```

(IV) transition_ctx

```
# transition_ctx holding setting as well as attributes
def _my_transition_impl(transition_ctx)
```