Unified Configuration Transitions

Greg Estren (gregce@google.com)
Bazel configurability team
Visibility: PUBLIC OUTSIDE GOOGLE
Reviewers: mstaib@google.com, jcater@google.com
Status: *May 24, 2018:* **IMPLEMENTED**

Summary

Bazel's configuration abstractions are unnecessarily complex due to legacy history. This document proposes a refactoring that removes that complexity.

Background

In Bazel, a **configuration transition** is when the environment a rule builds under changes. For example, when a binary builds with `--cpu=k8` and depends on a library that builds with `--cpu=ppc`, the library consumes a *transition* that changes the *cpu* setting.

Before dynamic configurations, Bazel's configuration support was limited and hard to change. Dynamic configurations vastly simplified the configuration model. But legacy abstractions from the old days resulted in an overly complex API that doesn't faithfully reflect this simplicity.

As of September 15, 2017, Bazel had five core abstractions for configuration transitions: Attribute.Transition, Attribute.SplitTransition, Attribute.ConfigurationTransition, CppRuleClasses.LipoTransition, and PatchTransition.

As of January 30, 2018, this has been reduced to three: ConfigurationTransition, SplitTransition, and PatchTransition.

This document proposes a redefinition of the remaining transitions aimed at providing the cleanest possible API and eliminating all remaining technical debt from pre-dynamic configuration Bazel.

Today

The dynamic configuration model essentially means that any transition transforms a set of BuildOptions into one or more output BuildOptions.

ConfigurationTransition is an empty interface. PatchTransition transforms a set of BuildOptions into one output. SplitTransition transforms a set of BuildOptions into one or more outputs.

Despite this commonality, SplitTransition and PatchTransition provide distinct and incompatible interfaces. This results in forked APIs, e.g.:

```
      Attribute.Builder.cfg(SplitTransition configTransition) {...}
      Attribute.Builder.cfg(ConfigurationTransition configTransition) {...}
      Attribute.Builder.cfg(SplitTransitionProvider splitTransitionProvider) {...}
```

and a brittle, overly complicated implementation, e.g.:

```
public static List<BuildOptions> applyTransition(BuildOptions fromOptions,
      ConfigurationTransition transition) {
    ...
    if (transition instanceof PatchTransition) {
      result = ImmutableList.of(((PatchTransition) transition).apply(fromOptions);
    } else if (transition instanceof SplitTransition) {
      List<BuildOptions> toOptions =
          ((SplitTransition) transition).split(fromOptions);
      if (toOptions.isEmpty()) {
        result = ImmutableList.of(fromOptions);
      } else {
        result = toOptions;
      }
    } else {
      throw new IllegalStateException(
          String.format("unsupported config transition type: %s",
              transition.getClass().getName()));
    }
}
```

, the co-existence of ComposingSplitTransition and ComposingPatchTransition, and so on.

Proposal

We redefine ConfigurationTransition to expose the core pattern of dynamic configurations: a transition transforms a set of BuildOptions into one or more output options:

```
  public interface ConfigurationTransition {
    List<BuildOptions> apply(BuildOptions fromOptions);
    /** Speed bump to discourage users from implementing this directly: */
    String reasonForCoreOverride();
  }
```

We retain PatchTransition and SplitTransition as distinct, easy-to-implement interfaces:

```
@FunctionalInterface
  public interface PatchTransition extends ConfigurationTransition {
    BuildOptions patch(BuildOptions fromOptions);
```

```
    @Override
    default List<BuildOptions> apply(BuildOptions fromOptions) {
      return ImmutableList.of(patch(fromOptions));
    }

    @Override
    default String reasonForCoreOverride() {
      "This is a core conceptual transition"
    }
  }
```

```
@FunctionalInterface
  public interface SplitTransition extends ConfigurationTransition {
    List<BuildOptions> split(BuildOptions fromOptions);

    @Override
    default List<BuildOptions> apply(BuildOptions fromOptions) {
      return split(fromOptions);
    }

    @Override
    default String reasonForCoreOverride() {
      "This is a core conceptual transition"
    }
  }
```

Users are encouraged to write transitions on top of PatchTransition or SplitTransition vs. ConfigurationTransition to encourage considering the consequences of each type. API methods and implementations consume ConfigurationTransition to automatically apply correct logic.

Analysis

This still creates more abstractions than necessary: strictly speaking we could make ConfigurationTransition the same as SplitTransition and have users manually return `ImmutableList.of(transformedOptions)` when they want simple patches.

But now this abstraction is intentional. It honors the reality that most transitions are likely to be patches by providing the simplest interface for that use case. It also makes users explicitly declare their intention to split configurations, which is a useful speed bump given the potential cost of that choice

`reasonForCoreOverride` disincentivizes users from directly implementing ConfigurationTransition. Making it an abstract class with a package-private constructor would accomplish this more strongly, but that precludes defining transitions with lambdas.

This proposal also limits the generality of ConfigurationTransition: in theory a transition doesn't have to apply the "transform some existing build options" model. But since that model is so core to how dynamic configurations work, further abstracting "transition" creates generality without purpose. If further generality is ever needed it's easy enough to add later.

Unchosen Proposal

In the dark old days, it was often deeply unclear whether consuming code should take a Transition, ConfigurationTransition, PatchTransition, or SplitTransition as input. Having just three transitions is better, but still doesn't eliminate the confusion.

An alternate paradigm is to let transition writers write any kind of transition they want, then force-reduce them to a single type before the configuration API takes them:

```
public final class ConfigurationTransition {
    public interface Splitter { List<BuildOptions> split(BuildOptions options); }
    public interface Patcher { BuildOptions patch(BuildOptions options); }

    private final Splitter internalSplitter;
    private ConfigurationTransition(Splitter splitter) {
      this.internalSplitter = splitter;
    }

    public static ConfigurationTransition patchTransition(Patcher patcher) {
      return new ConfigurationTransition(
        options -> ImmutableList.of(patcher.patch(options)));
    }

    public static ConfigurationTransition splitTransition(Splitter splitter) {
      return new ConfigurationTransition(options -> splitter.split(options));
    }

    public final List<BuildOptions> apply(BuildOptions options) {
      return internalSplitter.split(options);
    }
  }
```

Configuration creators can then write transitions as follows:

```
ConfigurationTransition simplePatcher = ConfigurationTransition.patchTransition(
      options -> { return options; });
  ConfigurationTransition simpleSplitter = ConfigurationTransition.splitTransition(
      options -> { return ImmutableList.of(options); });
```

Analysis

This more faithfully maps transitions to their basic essence: it gives users higher level distinctions that matter to them but strips these distinctions in the implementation layer. This facilitates a clearer and simpler implementation design.

But it also incurs complexity in the user layer:

```
ConfigurationTransition simplePatcher = ConfigurationTransition.patchTransition(
      options -> { return options; });
```

is not as simple as

```
PatchTransition simplePatcher = options -> { return options; };
```

This also complicates legitimate uses of inheritance, such as transitions that compose other transitions.

The takeaway is that this penalizes rule writers for the benefit of core Bazel developers. This isn't a worthwhile tradeoff, especially given the minor consequences to implementation design under either proposal.

Implementation Sequence

1. Rename PatchTransition.apply to PatchTransition.patch to de-clash with ConfigurationTransition's version **DONE**
2. Add new interface methods to ConfigurationTransition, PatchTransition, SplitTransition **DONE**
3. Merge implementation code that unnecessarily branches patches and splits **DONE**
4. Merge RuleTransitionFactory and SplitTransitionProvider into RuleDependentConfigurationTransition (which can directly implement the transition interface) **CANCELLED** (the interfaces aren't sufficiently similar)
5. Merge ComposingPatchTransition, ComposingSplitTransition, ComposingRuleTransitionFactory **DONE** (except ComposingRuleTransitionFactory)
6. Merge configuration APIs that are overloaded with patches and splits (e.g. Attribute.Builder#cfg) **DONE**