Bazel Constraint Equality

Author: schmitt@google.com
Reviewers:

- jcater@google.com (LGTM)
Last updated: February 12th, 2019 [shelved]
Status: **reviewed, implementation deferred until needed**

Platform definitions are a new mechanism in Bazel that replaces (often language-specific) flags that describe target and execution environments (for example --cpu, --host_cpu, --crosstool_top, --javabase, ...). They are instrumental in new functionality like allowing Bazel to choose an execution platform per action and performing toolchain selection. However their introduction to repositories building with Bazel could cause significant disruption of existing workflows and conflicts between imported repositories.

The main challenge we expect once platforms are in use is **definition fragmentation**, not of platforms themselves but of constraint settings and their values. Where previously there were informal agreements on the validity and meaning of values for flags like --cpu[1], these meanings will now be explicitly encoded in repositories in the form of constraints. While the Bazel team plans to provide a central repository to store constraints, we expect many repositories to define their own, particularly for specialised values. This will eventually lead to some repository importing two others, each of which defines a constraint A and B respectively such that A and B are semantically equivalent. Now if a platform with A is used, toolchains and select() statements referring to B will not match and vice versa - even though they should.

This document proposes a solution to constraint definition fragmentation which aims to be least invasive and onerous.

Background

Platform

A *platform* in Bazel is defined in a BUILD file and is a collection of *constraint values* (which in turn come from *constraint setting* enumerations). Here's a brief example, find a lot more information in the platform documentation and design doc:

[1] Which have caused significant headaches when there was fragmentation, such as in the case of macOS.

```
constraint_setting(name = 'java')

constraint_value(
    name = 'java7',
    constraint_setting = ':java')

constraint_value(
    name = 'java8',
    constraint_setting = ':java')

platform(
    name = 'linux_x86',
    constraint_values = [
      '@bazel_tools//platforms:linux',
      '@bazel_tools//platforms:x86_64',
      ':java8',
    ])
```

Platforms are propagated in Bazel's configuration and used to determine for each build target what environment it targets and what environment its toolchain(s) can be executed on. Multi-architecture use cases are not represented as a single platform in the configuration but a collection of multiple platforms.

Toolchain

A *toolchain rule* is a rule (native or Starlark) emitting a provider which describes a set of tools and their configurations such that they can be used with actions. A toolchain indicates that it is compatible with certain target and execution platforms by referencing their constraint values. Toolchains are described in the toolchain documentation.

Proposal

To simplify definition and implementation of this proposal, as well as understanding of the resulting code in users repository it is limited to solving the problem of **constraint value** fragmentation only. Constraint setting fragmentation, while theoretically possible should be rare. If a compelling use case comes up the method proposed can be extended to cover constraint settings as well.

Constraint Value Equality

In cases where a repository owner ends up with two distinct (in terms of Bazel path) but semantically equivalent constraint values we allow them to explicitly define that equality in their WORKSPACE file using a constraint_value_alias(from, to) rule:

```
constraint_value_alias("@repoA//some/path:longdash", "@repoB//config/package:em")
```

This will cause Bazel to match any reference to "@repoA//some/path:longdash" if "@repoB//some/path:em" is present in the platform and vice versa. So a select() branch will match a platform containing "@repoA//some/path:longdash" regardless of whether it is keyed on "@repoA//some/path:longdash" or "@repoB//some/path:em".

Transitive Aliases

More generally (and to support scenarios where many repositories are imported in parallel or repository A imports repository B which imports repository C and so forth[2]) more than two constraint values can map to the same semantic meaning. In such cases multiple constraint_value_alias instances can refer to the same constraint value, effectively creating equivalence classes with more than two members. Bazel will evaluate all constraint_value_alias statements across all recursively imported WORSPACE files and compile the set of global constraint value equivalence classes.

Note that repeated aliases and "reverse" aliases are perfectly fine and treated as no-ops (they don't modify the equivalence class).

For example:

```
# //:WORKSPACE
constraint_value_alias("@repoA//some/path:robert", "@repoB//config:robert")
constraint_value_alias("@repoB//config:robert", "//my/definitions:bob")

# @repoB//:WORKSPACE
constraint_value_alias("//config:robert", "@repoD//names:rob")
```

Results in all of "@repoA//some/path:robert", "@repoB//config:robert", "//my/definitions:bob" and "@repoD//names:rob" being in the same equivalence class (i.e. if one of them is in a platform then any of them match that platform).

Branch Collisions

Using constraint aliases it is possible to change the behavior of constraint matching in select() statements and toolchain selection that have multiple branches matching a single constraint value equivalence class. In these cases Bazel will behave just as if the same constraint value had been used in multiple branches, with extra error information as necessary. For example:

---

[2] Actually, recursive WORKSPACE evaluation doesn't exist yet - but it is being designed and implemented and should support this functionality as outlined here.

```
# //:WORKSPACE
constraint_value_alias(
    "@repoA//some/path:longdash",
    "@repoB//config/package:em"
)

# //foo/BUILD
config_setting(
    name = "longdash",
    constraint_values = [ "@repoA//some/path:longdash" ])
config_setting(
    name = "em",
    constraint_values = [ "@repoA//some/path:em" ])


some_rule(
    name = "foo",
    srcs = select({
        ":longdash": [longdash.txt],
        ":em": [em.txt]
    }))
```

When //foo:foo is analysed Bazel will raise an error noting that two branches of the select statement have semantically equivalent keys, with a pointer to where the equivalence was defined.

Implementation

This is a draft, especially considering recursive WORKSPACE evaluation doesn't exist yet.

The new constraint_value_alias starlark function is registered in the WorkspaceFactory. It functions very similarly to the existing register_toolchains function in that it causes alias registrations to be attached to the //external package which can be retrieved by its SkyKey. Just like for toolchain resolution the data is then queried by the ConfiguredTargetFunction and attached to the RuleContext for availability in rule processing.

The aliases can be used to construct platform() instances, thus any comparison with a platform's constraint values would take the aliases into account; this covers both select() branch comparisons and toolchain resolution. At this point we can also enforce that all aliases in the same equivalence class share the same constraint setting.

Aliases are stored as the labels mapped (not targets, to save memory) and the location where the mapping was defined (kept to allow better error messages and debugging):

```
class ConstraintAliases {
  ConstraintAliases(Map<Label, Label> constraintAliases) { ... }

  /**
   * Returns the canonical label that the given label resolves to or the given
   * label if it was not registered as an alias.
   */
  Label resolve(Label constraintValue) {
    if (constraintAliases.contains(constraintValue)) {
      return constraintAliases.get(constraintValue);
    }
    return constraintValue;
  }

  static class Builder {
    Set<Set<Label>> equivalences = new HashSet<>();

    // TODO: Also store original mapping location for debugging.
    void addAlias(Label from, Label to) {
      ImmutableSet.Builder<Label> newEquivalence = ImmutableSet.builder()
          .add(from)
          .add(to);
      for (equivalence : equivalences) {
        if (equivalence.contains(from) || equivalence.contains(to)) {
          newEquivalence.addAll(equivalence);
          equivalences.remove(equivalence);
        }
      }
      equivalences.add(newEquivalence.build());
    }

    ConstraintAliases build() {
      ImmutableMap.Builder<Label, Label> mappings = ImmutableMap.builder();
      for (equivalence : equivalences) {
        Label canonical = equivalence.stream()
            .min(l1, l2 -> l1.toString().compareTo(l2.toString()))
            .get();
        for (Label label : equivalence) {
          mappings.put(label, canonical);
        }
      }
      return new ConstraintAliases(mappings.build());
    }
```

```
  }
}
```

In the select() implementation, when detecting conflicting branches the error message is extended by alias information in case this played a role in causing the conflict.

Rollout Plan

The new functionality can be added to Bazel without any guards as it doesn't modify existing behavior.