Diamond Splitting for External Repositories

PUBLIC DOCUMENT
Author: dslomov@google.com
Status: Approved, being implemented
Last updated: 2018-03-07
Reviewers: aehlig, Skylark Team
Prototype implementation

In the current WORKSPACE file design, the namespace of external workspaces (names occurring before @) is uniform across all referenced workspaces. In other words,

- assume the main project P refers to two external repositories A and B
- A refers to a external (to itself) workspace D via @D//<...> labels in its BUILD files
- B refers to a external (to itself) workspace D via @D//<...> labels in its BUILD files
- project P needs to depend on unique external workspace D that satisfies requirements of A and B

Effectively, names that occur after @ in labels are forced to be *universally unique:* all projects need to agree on a naming scheme. Our guidance has been to use reverse domain names with dots replaced by underscores (such as "io_bazel", back when Bazel's website was "bazel.io") as unique ids. This is painful and problematic for users (people might change domains, projects start as hobbyist projects with no real domain name etc). Moreover, this enforces "single version policy": if A and B in the above example do depend on the same D, but on different incompatible versions, there is no way to reconcile these dependencies.

This document proposes a way to redirect the @*name* references *within a specific repository* to a different repository.

Proposal

The key functionality we need to make available is for the user to declare: "within repository a, @d means @x".

Proposed Solution

We equip all repository rules with a "string_string_dict"-typed attribute, `assignments`:

```
local_repository(name = "a",
                 ...
                 assignments = { "d" : "x", ... })
```

All assignments pertaining to the repository are listed in its declaration. No later reassignments are allowed. When any BUILD or .bzl file is loaded from repository 'a', all labels of the form '@d//<...>' are interpreted as '@x//<...>.

Interaction with the future "recursive workspaces" proposal

In the future, we want to allow loading dependent projects' WORKSPACE files automatically (design doc). Under this proposal, renaming has a nice semantics: if a dependent project adds a dependency that is renamed, bazel will load it under a new name. For example:

main WORKSPACE:

```
local_repository(name = "a", ...
                 recursive = True,
                 assignments = { "d" : "x" })
```

a's WORKSPACE:

```
local_repository(name = "d", ...)
```

When a's WORKSPACE is recursively loaded, "d", when it is introduced, is named "x", not "d" (And if "x" is already defined in main WORKSPACE, a new "x" is *not* introduced).

Interaction with "deps.bzl" pattern

In the absence of built-in support for "recursive workspaces", many Bazel projects adopt "deps.bzl" pattern: the repository reassignments need to be injected into the repository rules injected by `<something>_dependencies()`:

```
git_repository(name = "rules_go",
               assignments = { "org_golang_x_text" : "foobar" }
               ...)
load("@rules_go//:deps.bzl", "go_dependencies")
go_dependencies()
```

This simulates the "recursive workspaces" behavior. To support renaming, .._dependencies function will need to accept renaming map as a parameter and use it when adding dependencies:

```
def realname(name, assignments):
    return assignments[name] if name in assignments else name
def go_dependencies(assignments = {})
    _maybe(git_repositiory,
           name = realname("org_golang_x_text", assignments),
           assignments = assignments # pass the renames
           remote = "https://github.com/golang/text",
           ...
    )
    ...
```

where _maybe is a function to only introduce a repository if it is not already introduced:

```
def _maybe(repo_rule, name, **kwargs):
```

```
if name not in native.existing_rules():
  repo_rule(name=name, **kwargs)
```

(Rejected) Alternative

We introduce a new top-level function in WORKSPACE file, assign:

```
assign(within = "@a", local = "@d", to = "@x")
```

After this declaration, when loading BUILD files within repository "a", all "@d" references will be interpreted as "@x". Assignments accumulate over the entirety of WORKSPACE file(s) (including --override_repositiory and so on) and apply to BUILD files as they are loaded.

The interesting question arises w.r.t. load statements though. Consider:

WORKSPACE file:

```
local_repository(name = "a", ...)
assign(within = "@a", local = "@d", to = "@x")
load("@a//:defs.bzl", "foo")
```

defs.bzl within a:

```
load("@d//utils:utils.bzl", "some_util")
```

Depending on the location of "assign" before or after `load("@a//:defs.bzl",...) the load inside defs.bzl will have different semantic.

The "assign" also supports the "deps.bzl" pattern that repos use to get "poor-man's recursive repositories": the dependencies brought in via `<something>_dependencies()` call can be fixed by its caller..

Reasons for rejection:

1. Funky side-effects
2. Interaction with recursive workspace loading is tricky
3. deps.bzl pattern is easily fixable in line with the future recursive workspaces solution

Implementation

Under the proposal, for every target there exist an unique label that identifies it. If we declare the assignment of d to x with a there is no future redeclaration of "x": any labels within repository "a" that have "@d" prefix, should be, at loading time, remapped to "@x". The remapping can happen very early (at loading time).

Implementation-wise, the key issue we face is that Label.parseAbsolute(String) is no longer a viable method. To interpret a string as a label, the caller needs to provide a context: repository name mappings that are applicable for conversion. The Label class always represent a label in

form that does not require further remapping. Prototype implementation has further details (it currently implements the "Rejected Alternative").