[RFC] Unified Constraint Rule with Defaults

Unified Constraint Rule

A configuration is a collection of constraints. There are two components of a constraint, constraint setting (the key) and constraint value (value for a key). Below is an example of an OS constraint with linux, macos, and windows as values.

```python
# cfg//BUCK

constraint_setting(name = "os")  # this is constraint key

# These are constraint values for the key
constraint_value(
  name = "linux",
  constraint_setting = ":os",
)

constraint_value(
  name = "macos",
  constraint_setting = ":os",
)

constraint_value(
  name = "windows",
  constraint_setting = ":os",
)
```

There are two ergonomic problems with the way that constraint values are currently defined.

1. As shown in the previous example, defining a single constraint is rather verbose.
2. It's not possible for a user to see all the possible values of a constraint setting. For example, it’s possible to define an OS constraint value in a different BUCK file from the OS constraint setting.

Instead, consider the following syntax for OS.

```python
# cfg//BUCK
```

```
constraint(
  name = "os",
  values = [
    "linux",
    "macos",
    "windows",
  ],
  # other fields...
)
```

This is much more concise, and it's immediately clear to a reader that `linux`, `macos`, and `windows` are values for `os` constraint.

The `constraint` rule creates exactly one target with label `cfg//:os`, where `cfg//:os` is the constraint key. Values are referenced via subtargets, ex. `cfg//:os[linux]` and `cfg//:os[macos]`. These can be used in selects, for example,

Python
```
deps = select({
  "DEFAULT": [],
  "cfg//:os[linux]": [":linux-only-dep"],
})
```

They can also be used in modifiers or platforms.

Python
```
# PACKAGE file

set_cfg_modifiers(cfg_modifiers = [
  "cfg//:os[linux]",
])

# BUCK file
platform(
  name = "linux",
  constraint_values = [
    "cfg//:os[linux]",
```

```
  ],
)
```

Values must be strings.

Aliases can be used to alias subtarget values to targets for backwards compatibility with existing syntax where values are targets.

```python
# This is old syntax
constraint_value(
  name = "linux",
  constraint_setting = ":os",
)

# This is equivalent target that uses new syntax.
configuration_alias(
  name = "linux",
  actual = ":os[linux]",
)

# In some other BUCK file
deps = select({
  "DEFAULT": [],
  "cfg//:linux": [":linux-only-dep"],
})
```

By default, constraints defined using `constraint` cannot have constraint values defined outside of the `constraint` rule. However, it is possible to relax this limitation by setting `allow_external_definitions_of_constraint_values = True`. This is useful for extending values outside of where the constraint is defined. For example, we may have some default constraints defined in prelude but those can be extended either internally or in individual open-source projects.

```python
# cfg//BUCK
constraint(
  name = "os",
```

```
  values = [
    "linux",
    "macos",
    "windows",
  ],
  allow_external_definitions_of_constraint_values = True
  # other fields ...
)

constraint_value(
  name = "powerpc",
  constraint_setting = ":os",
)

### The following constraint_value would be an error
constraint(
  name = "enable_incremental_compilation",
  values = [
    "true",
    "false",
  ],
  # other fields ...
)

constraint_value(
  name = "maybe_incremental_compilation",
  constraint_setting = ":enable_incremental_compilation",
)
```

We plan to migrate all `constraint_setting` to new `constraint` rule once buck1 is EOL.

`default` attribute

Suppose we have the following constraint defined for sanitizer.

Python

```
constraint(
  name = "sanitizer",
  values = [
    "asan",
    "tsan",
```

```
    "msan",
    "none",
  ],
)
```

`cfg//:sanitizer[none]` is used to represent that no sanitizer is present. This is useful for disabling sanitizer on command line, for example by specifying `--modifier cfg//:sanitizer[none]`. However, a configuration having `cfg//:sanitizer[none]` and a configuration without sanitizer constraint set would be considered different configurations and have different configuration hashes even though they are equivalent in behavior.

The unified constraint rule enables a way to define defaults for constraints such that a configuration with default constraint value is equivalent to a configuration with the constraint unset.

Python
```
constraint(
    name = "sanitizer",
    values = [
        "asan",
        "tsan",
        "msan",
        "none",
    ],
    default = "none",
)
```

In this example, `none` is the default value for `:sanitizer` constraint. When a configuration does not have `:sanitizer` set, it automatically defaults to `none`, and the following select will resolve to `None`.

Python
```
sanitizer_name = select({
    ":sanitizer[none]": None,
    ":sanitizer[asan]": "ASAN",
    ":sanitizer[msan]": "MSAN",
    ":sanitizer[tsan]": "TSAN",
```

```
})
```

The select from the above example is guaranteed to resolve and never hit an unresolved select error.

As mentioned previously, a configuration with `cfg//:sanitizer[none]` will be identical to a configuration without sanitizer constraint set.

The default must be a value listed in `values` attribute or `None`. If it is `None`, then there is no default value. `default` will be a required field, so `None` must be explicitly specified if there is no default.

The default can be set to select on other constraint values. For example, below is an example of optimization level constraint based on build mode constraint.

Python
```
constraint(
    name = "cxx_optimization",
    values = [
        "o0",
        "o1",
        "o2",
        "o3",
    ],
    default = select({
        "DEFAULT": None,
        ":build_mode[dev]": "o0",
        ":build_mode[opt]": "o3",
    }),
)
```

This means that a select on optimization level can directly select on the cxx optimization level constraints rather than high-level constraints like build mode.

Currently, a select on cxx optimization levels looks something like this.

Python
```
cxx_flags += select({
```

```
    ":build_mode[opt]": "-O3",
    "DEFAULT": "-O0",
})
```

This is undesirable because the select is selecting on constraints that are too high level. For example, there are no ways to tweak the build with a different optimization level on the command line. With defaults, these can be expressed as

```python
cxx_flags += select({
    ":cxx_optimization[o0]": "-O0",
    ":cxx_optimization[o1]": "-O1",
    ":cxx_optimization[o2]": "-O2",
    ":cxx_optimization[o3]": "-O3",
})
```

Decoupling how the cxx optimization constraint from the high level build mode constraint means there's more flexibility in how constraint values are used. As an example, you can choose to always use `-O3` for python builds even in dev mode. Additionally, you may be able to use something like `--modifier cfg//compiler:cxx_optimzation[o3]` on the cli to tweak with optimization levels.

The `default` attribute enables a wide variety of configurations to be defined without constraint values being explicitly set in platforms or modifiers. This is particularly beneficial for open-source environments. Imagine that you have an OSS project that is looking into adopting buck2. Currently, that project may have to define its own set of constraints, selects, and toolchains. With defaults, these projects can immediately use a set of default configurations exported from buck2 without defining a single platform or modifier.

Changes to ConfigurationInfo API in transitions

In transitions, constraints are directly accessed in the `constraints` attribute of `ConfigurationInfo` as a dictionary of constraint settings to values. To support defaults, `constraints` will become an opaque `ConstraintsInfo` object that supports `new`, `get`, `insert`, and `pop` methods but not methods that can see constraint keys without knowing them like `iter`. In practice, we don't expect this to limit what people typically do in transitions.

Link to google doc: [RFC] Unified Constraint Rule with Defaults