`selectable` rule

Our sanitizer constraint today looks something like this (syntax is from the new unified constraint rule).

```python
# prelude//constraints/BUCK

constraint(
  name = "sanitizer",
  values = [
    "asan",
    "tsan",
    "ubsan",
    "asan-ubsan",
    "more-asan-ubsan",
    "afl",
    "afl-asan",
    "nosan",
  ],
  default = "nosan",
)
```

As one can see, while most sanitizers are standalone, some sanitizers like ASAN can be used in combination with other sanitizers and thus show up in multiple values. This leads to a problem that it's difficult to correctly write a select on ASAN that accounts for all cases. Essentially, you have to write something as follows.

```python
asan_flags = select({
  "DEFAULT": [],
  "prelude//constraints:sanitizer[asan]": ["-Zsanitizer=address"],
  "prelude//constraints:sanitizer[asan-ubsan]": ["-Zsanitizer=address"],
  "prelude//constraints:sanitizer[more-asan-ubsan]"["-Zsanitizer=address"],
  "prelude//constraints:sanitizer[afl-asan]": ["-Zsanitizer=address"],
}),
```

This is unergonomic to write and rather impossible for an average user to remember. Indeed, if you search the codebase, you will find that most selects just select internally for `ovr_config//build_mode:asan` (equivalent to `prelude//constraints:sanitizer[asan]` in this example) without selecting for all the other sanitizer variations that also have ASAN enabled, and that can lead to correctness issues.

For example, missing select on all ASAN variations was the cause of buck2 being not runnable in fbcode dev mode (D61327904). The following proposal introduces a `selectable` rule to fix this type of issue.

A `selectable` rule can be used to define a target that goes in the key of a select. For the above ASAN example, you can define a selectable for ASAN that looks something like this.

```python
# prelude//constraints/sanitizer/BUCK

selectable(
  name = "asan",
  value = select({
    "DEFAULT": False,
    "prelude//constraints:sanitizer[asan]": True,
    "prelude//constraints:sanitizer[asan-ubsan]": True,
    "prelude//constraints:sanitizer[more-asan-ubsan]": True,
    "prelude//constraints:sanitizer[afl-asan]": True,
  }),
)
```

Now, users can easily select on ASAN by selecting on `prelude//constraints/sanitizer:asan` instead of `prelude//constraints:sanitizer[asan]`.

```python
asan_flags = select({
  "DEFAULT": [],
  "prelude//constraints:sanitizer[asan]": ["-Zsanitizer=address"],
})
```

This is semantically equivalent to the previous `asan_flags` example written above.

Selectables can be nested, so a selectable can be used within a `select` statement for other selectables.

```python
# prelude//constraints/sanitizer/BUCK

selectable(
  name = "asan_or_ubsan",
  value = select({
```

```
    ":asan": True,
    "prelude//constraints:sanitizer[ubsan]": True,
    "DEFAULT": False,
  })
```

The `selectable` rule can be viewed as a more powerful version of `config_setting`. Whereas `config_setting` can only express ANDs, `selectable` can express ORs and NOTs via select. Once implemented, we will roll it out to all existing `config_setting`s and kill `config_setting`.

Simplifying Selects

One UX issue with the `selectable` rule is that it can be much more verbose to write a `selectable` than writing a `config_setting` of the same semantics, especially for a long list of constraints. I think the best way to fix this if needed is to allow a tuple of constraints to express an AND in select keys. For example, the following two selects would be semantically equivalent, but `select2` with tuple in key is significantly less verbose than `select1.`

Python

```
select1 = select({
  "DEFAULT": False,
  "prelude//:os[linux]": select({
    "DEFAULT": False,
    "prelude//:cpu[x86_64]": select({
      "DEFAULT": False,
      "prelude//:lto[thin]": True,
    }),
  }),
})

select2 = select({
  "DEFAULT": False,
  (
    "prelude//:os[linux]",
    "prelude//:cpu[x86_64]",
    "prelude//:lto[thin]",
  ): True,
})
```

This does mean select dictionaries no longer coerce directly into JSON, but that's easily fixable as we can always convert the dictionary into something that is JSON-coercible like list of tuples.

Select Resolution

One problem with implementing `selectable` in buck2 is how to handle the `select` resolution refinement process that currently exists. There are a few ways I imagine this can be done. We can choose to change the select resolution process to something more user intuitive like first-match rather than refinement (although that does require us to stop sorting dictionary literals in starlark). Alternatively, we can count what unique constraints are transitively matched when evaluating the selectable and use that for refinement.

`select_deps`

It is often the case that a constraint may need some additional `selectable` targets defined to be used ergonomically in the repo. However, these selectables often mostly look the same for all constraint values and defining one selectable per constraint value can be quite verbose. `select_deps` is a way to make `constraint` and `constraint_value` targets define selectables that are different from the default.

For example, consider our internal fbcode platform constraint, which would look something like this in the new constraint rule.

```
Python
constraint(
  name = "linux-clang",
  values = [
    "12",
    "15",
    "17",
  ],
  default = "15",
)
```

These constraints are only relevant when the OS is linux. So if I want to set a `compiler_version` flag on the fbcode build info target, that should select on `linux-clang` only if our OS matches linux. There are many places in the repo that would want to select on the clang version, so it would be useful to define additional `selectable` targets for this that would select on linux before selecting on linux-clang so that users don't need to remember to do this every time when they use a select. However, not only is this verbose, but the author would also need to remember to additionally specify a `select` on linux on the `default` attribute of linux-clang. Otherwise a select on clang-15 would always match even when the OS is not linux. Overall, this would look something like the following.

Python

```
constraint(
  name = "linux-clang",
  values = [
    "12",
    "15",
    "17",
  ],
  default = select({
    "DEFAULT": None,
    "prelude//:os[linux]": "15",
  }),
)

selectable(
  name = "linux-clang12",
  value = select({
    "DEFAULT": False,
    ("prelude//:os[linux]", ":clang[12]"): True,
  }),
)

selectable(
  name = "linux-clang15",
  value = select({
    "DEFAULT": False,
    ("prelude//:os[linux]", ":clang[15]"): True,
  }),
)

selectable(
  name = "linux-clang12",
  value = select({
    "DEFAULT": False,
    ("prelude//:os[linux]", ":clang[17]"): True,
  }),
)
```

`select_deps` is a way to introduce additional targets to evaluate on a select in addition to the constraint value itself such that this complexity can be avoided. For example, the above can be concisely expressed as follows.

Python

```
constraint(
  name = "linux-clang",
```

```
  values = [
    "12",
    "15",
    "17",
  ],
  default = "15",
  select_deps = ["prelude//:os[linux]"],
)
```

Now whenever Buck selects on any `linux-clang` subtarget like `:linux-clang[12]`, it will also by default select on `prelude//:os[linux]`. No additional `selectable` targets need to be defined. When `linux-clang` is a constraint in the platform, a select on `:linux-clang[15]` will only match the platform if `prelude//:os[linux]` is satisfied and will not match otherwise.

`select_deps` can be set to a list of constraint values/selectables. When a list is specified, the list is evaluated as an “AND” of all values, so the `select_deps` are only satisfied if each individual entry in `select_deps` is satisfied. Additionally, `select`s can be used in `selectable_deps` with True/False as values to express “OR”s and “NOT”s inline. Any `False` value would indicate that the entire `select_deps` fails to match. The following would be a valid `select_deps` attribute.

Python

```
select_deps = [
  "prelude//constraints/sanitizer:asan",
] + select({
  "DEFAULT": [False],
  # Note this is technically a trivial select with the above
  # definition of `linux-clang" constraint. It's just for demo.
  ("prelude//:os[linux]", "prelude//:linux-clang[17]"): [],
  ("prelude//:os[windows]", "prelude//:windows-clang[17]"): [],
})
```

Additional Notes

For `selectable`, I had an additional idea to add safeguards to prevent `prelude//constraints:sanitizer[asan]` from being selected on directly by a confused user when `prelude//constraints/sanitizer:asan` is the preferred “selectable” to use.

To support this, I had in mind a feature to disable `prelude//constraints:sanitizer[asan]` from being selected outside of `selectable` rule. We can add a `generate_default_selectable` field on the `constraint` rule that defaults to True such that every constraint value of that constraint can be used in a select by default. When set to False, however, it will no longer allow these constraint values to be selected by default. I decided to abandon it because it didn't seem super useful given the existence of `selectable` and `select_deps`, particularly for examples I had in mind like sanitizers. However, it would be fairly easy to implement if some use case does come up for it.

Link to google doc: [RFC] selectable rule