Skylark API to the C++ toolchain - API approval

Author: hlopko@google.com
Status: Submitted
Want LGTM: skylark-cabal@
Last Updated: 2018-06-15

Summary

As elaborated in the original design doc, I'm asking for feedback for the actual Skylark API. The API proposed in this doc has been prototyped internally and I'm confident that it fulfils its goals.

Example use

```
cc_toolchain = find_cpp_toolchain(ctx)
feature_configuration = cc_common.configure_features(
    cc_toolchain = cc_toolchain,
    requested_features = ctx.features,
    unsupported_features = ctx.disabled_features +
      ['thin_lto', 'module_maps', 'use_header_modules'])
variables = cc_common.create_compile_build_variables(
    feature_configuration = feature_configuration,
    cc_toolchain = cc_toolchain,
    user_compile_flags = ctx.attr.copts)
args = cc_common.get_command_line(
    feature_configuration = feature_configuration,
    action_name = C_COMPILE_ACTION_NAME,
    variables = variables)
```

API

RED marks fields that will be removed once there are no legacy fields in the CROSSTOOL anymore.

cc_common.configure_features

Creates a feature configuration instance.

```
feature_configuration cc_common.configure_features(
  cc_toolchain,
  requested_features=[],
  unsupported_features=[])
```

cc_toolchain	CcToolchainInfo cc_toolchain for which we configure features
requested_features	sequence List of features to be enabled.
unsupported_features	sequence List of features that are unsupported by the current rule

cc_common.get_command_line

Returns Args instance with the command line generated using given feature configuration with given variables, for given action.

Since some use cases require constructing the command line in the analysis phase (and adding/moving/removing random flags), we will need to add a way to convert Args instance to a list, or two c++ get_command_line methods, one that returns Args, one that returns list of strings.

```
Args cc_common.get_command_line(
  feature_configuration,
  action_name,
  variables)
```

feature_configuration	feature_configuration

	Feature configuration to be queried.
action_name	Name of the action for which we generate command line.
variables	variables Build variables to be used for feature expansion.

cc_common.get_environment_variables

Returns environment variables to be set for given action.

```
dict cc_common.get_environment_variables(
  feature_configuration,
  action_name,
  variables)
```

feature_configuration	feature_configuration Feature configuration to be queried.
action_name	Name of the action for which we fetch environment variables.
variables	variables Build variables to be used for feature expansion.

cc_common.get_tool_for_action

Returns tool path for given action.

```
String cc_common.get_tool_for_action(
  feature_configuration,
  action_name)
```

feature_configuration	feature_configuration Feature configuration to be queried
action_name	Name of the action

cc_common.is_enabled

Returns True if given feature is enabled

```
bool cc_common.is_enabled(feature_configuration, feature_name)
```

feature_configuration	feature_configuration Feature configuration to be queried
feature_name	Name of the feature

cc_common.create_compile_variables

Returns compile build variables to be used with feature configuration.

```
variables cc_common.create_compile_build_variables(
  cc_toolchain,
  feature_configuration,
```

```
source_file=None,
output_file=None,
user_compile_flags=[],
include_directories=[],
quote_include_directories=[],
system_include_directories=[],
preprocessor_defines=[],
use_pic=False,
add_legacy_cxx_options=False)
```

cc_toolchain	CcToolchainInfo cc_toolchain for which we are creating build variables.
feature_configuration	feature_configuration Feature configuration to be queried.
source_file	Optional source file for the compilation. Please prefer passing source_file here over appending it to the end of the command line generated from cc_common.get_command_line, as then it's in the power of the toolchain author to properly specify compiler flags.
output_file	Optional output file of the compilation. Please prefer passing output_file here over appending it to the end of the command line generated from cc_common.get_command_line, as then it's in the power of the toolchain author to properly specify compiler flags.
user_compile_flags	depset Collection of additional compilation flags.

include_directories	depset Collection of include directories.
quote_include_directories	depset Collection of quote include directories.
system_include_directories	depset Collection of system include directories.
preprocessor_defines	depset Collection of preprocessor defines.
use_pic	When true the compilation will generate position independent code.
add_legacy_cxx_options	Add options coming from legacy cxx_flag crosstool fields.

cc_common.create_link_variables

Returns link build variables to be used with feature configuration.

```
variables cc_common.create_link_build_variables(
  cc_toolchain,
  library_search_directories=[],
  runtime_library_search_directories=[],
  user_link_flags=[],
  output_file=None,
  param_file=None,
```

```
def_file=None,
is_using_linker=True,
is_linking_dynamic_library=False,
must_keep_debug=False,
use_test_only_flags=False,
is_static_linking_mode=True)
```

cc_toolchain	CcToolchainInfo cc_toolchain for which we are creating build variables.
feature_configuration	feature_configuration Feature configuration to be queried.
user_link_flags	depset Collection of user provided link flags.
library_search_directories	depset Collection of include directories.
runtime_library_search_directories	depset Collection of runtime library directories.
output_file	Optional output file path.
param_file	Optional param file path.

def_file	Optional .def file path.
is_using_linker	True when using linker, False when archiver.
is_linking_dynamic_library	True when creating dynamic library, False when executable or static library.
must_keep_debug	When set to True, bazel will expose 'strip_debug_symbols' variable, which is usually used to use the linker to strip debug symbols from the output file.
use_test_only_flags	When set to True flags coming from test_only_linker_flag crosstool fields will be included
is_static_linking_mode	True when using static_linking_mode, False when using dynamic_linking_mode.

Appendix: Feature Configuration

TLDR: Feature configuration is a black box instance of a simplified SAT solver initialized from a list of features to be enabled and a list of features to be disabled. It is used to generate command lines for C++ (and ObjC) actions.

Note: We are working with spomorski@ on the proper bazel documentation as we speak.

A *feature* is anything that requires special command line flags, actions, constraints on the execution environment or changes to the dependencies in bazel, which are not enabled by

default. Features can be something as simple as allowing BUILD files to select configurations of flags, like in the case of *crosstool_annotalysis*, or include new compile actions and inputs to the compile, like in the case of *header_modules* or *ThinLTO*.

Feature selection

A feature will be enabled if and only if both Bazel/rules and the CROSSTOOL support the feature. Bazel may have arbitrary signals for when a feature is enabled. Features can have interdependencies, depend on command line flags, BUILD file settings, or similar.

Feature dependencies and relationships

Feature level constraints:

1. **requires: [‘feature1’, ‘feature2’]** Allows a feature to specify that it is only supported if a set of different features is enabled. This is used for example when a feature is only supported in certain build modes (features "opt", "dbg" or "fastbuild"). Multiple requires will be met if any of the requires is met.
2. **implies: ‘feature’** Provides the ability for one feature to imply another. An example is that a module compile implies the need for module maps. This can be supported very simply by a repeated "*implies*" string in the feature. Each of the strings must name some other feature. Enabling a feature also implicitly enables all features, well, implied by it, i.e., it functions recursively. This also provides the ability to factor common subsets of functionality out of a set of features, for example the common parts of the sanitizers.
3. **provides: ‘feature’** The second useful specification is a constraint.We allow features to indicate that they are one of several mutually exclusive alternative features with a single optional string "provides". For example, all of the sanitizers could specify ‘provides: "sanitizer"’. The only benefit this provides is nicer errors: if a user asks for two alternatives at the same time, Bazel can error, explain why, and even list the alternatives available.

Note that all of these constraints can only be applied in a positive way. The absence of features, e.g. requires: '-features', cannot be used as that would make the feature selection a

hard satisfiability problem, which is problematic to solve efficiently. In case you'd need this behavior, a common solution is to add a complementary feature, e.g. no_feature.

Feature configuration

Feature configuration is a result of the feature selection. We assume that a single feature configuration is used for all actions registered by a single target. For generating command lines feature configuration needs a Variables instance that supplies data to be used to expand templates defined in the CROSSTOOL file. It is often the case that we create a separate Variables instance for every action registered by a target.

C++ actions expand their command lines only in the execution phase, therefore they have to retain feature configuration instance and Variables instance. Memory pressure is a force strongly influencing their design.