Skylark API to the C++ toolchain

This is publicly accessible document.

Author: hlopko@google.com
Status: Implemented
Want LGTM: plf@, lberki@, ulfjack@, dslomov@
Last Updated: 2017-12-20

Goals

- Design a new API that gives access to the command line generation used by C++ rules that has access to flags emitted by feature configuration (the old API doesn't work with feature configuration, and that is blocking removal of legacy fields from CROSSTOOL)
- Remove existing API (since it's blocking platforms migration of C++ rules)
- Nice to have:
    - Crosstool author should have the ability to position custom additional flags (copts, linkopts) precisely on the generated command line
    - Crosstool author should have the final saying in what flags get passed, Skylark rules should not be able to remove arbitrary flags (very unlikely to be satisfied by this design), instead they should cooperate with CROSSTOOL author to define "features" that allow conditional behavior.

Current Problems

Currently there are 2 ways of accessing C++ Toolchain information from Skylark:

- ctx.toolchains["//tools/cpp:toolchain_type"] that is incorrect since it doesn't provide all the flags that are configured in the CROSSTOOL, namely it doesn't return anything from feature configuration. It only returns flags from legacy crosstool fields. This is preventing crosstools from being migrated to features and action_configs (b/65151735)
- ctx.fragments.cpp in addition to the problems of the above is also incorrect because it provides non-configured fields (e.g. sysroot).

The fact that neither of these APIs is correct doesn't prevent their use from custom Skylark rules both internally and externally.

There are 2 separate use cases for the API:

1) Get compiler/linker flags that would have been used by C++ rules, likely excluding any action not as simple as compile and link (e.g. C++ features such as modules, thinlto, .d file). Often times the actual compiler used is not the one configured in the crosstool, but

"something else" (e.g. CLIF). The flags are sometimes written to a file or passed down to a tool that then invokes C compiler or linker on its own (e.g. Go rules).

2) Actual C++ rules, once rewritten in the Skylark, will need to use this API to generate command lines for themselves.

API to be removed/replaced

- ar_executable
- built_in_include_directories
- c_options
- compiler
- compiler_executable
- compiler_options
- cpu
- cxx_options
- dynamic_link_options
- fully_static_link_options
- ld_executable
- libc
- link_options
- mostly_static_link_options
- nm_executable
- objcopy_executable
- objdump_executable
- preprocessor_executable
- strip_executable
- sysroot
- target_gnu_system_name
- unfiltered_compiler_options

CcToolchainProvider:

- libc
- link_options_do_not_use
- unfiltered_compiler_options_do_not_use

Proposal - Skylark API to the Feature Configuration

There's a possibility that a single API layer will not be sufficient. To support use case 2) from above, we definitely need a low level access to feature configuration:

```
Cc_toolchain_provider:
FeatureConfiguration configure_features(ctx (?), requested_features,
unsupported_features)

FeatureConfiguration:
boolean is_enabled(feature_name)
List<String> get_command_line(action_name, variables)
Map<String, String> get_environment_variables(action_name, variables)
String get_tool_for_action(action_name)
```

This is what C++ rules currently use (modulo flag filtering that I omitted as it is subject to change). This API is stable and well supported. There are 2 drawbacks:

- The API exposes action names (such as c++-compile or c++-link-executable).
- The API relies on the right build variables passed in (arguments to get_command_line and get_environment_variables). These are currently collected from multiple places (cc_toolchain, CppModel for compilation related build variables, CppLinkActionBuilder for link related build variables, and some more registered from objc rules) and their values depend on the configuration, toolchain, inputs, outputs, enabled features, and the action type being registered. In other words, on anything that is available to the native rule implementation.

Since action names are already part of the public API of Bazel (users need to put the right action names into CROSSTOOL), I expect this not to cause problems. But making all logic related to the each build variable available to Skylark would require major re-architecting of the C++ rules implementation comparable to the full rewrite of the rules to Skylark.

Assuming that most of the users of the Skylark API to the CcToolchain only need the basic command line, it is possible to provide build variables building functions on cc_common to Skylark:

```
cc_common.create_compile_build_variables(cppConfiguration, ccToolchain,
enabledFeatures, copts, usePic, *includePaths, defines,
mapOfAdditionalVariables)

cc_common.create_link_build_variables(cppConfiguration, ccToolchain,
enabledFeatures, linkopts, mapOfAdditionalVariables)
```

These functions return a Variables instance as a “black box”, and the only only thing users can do is pass this “black box” to the feature_configuration.get_command_line.

This is enough to expose for compilation:

```
USER_COMPILE_FLAGS
LEGACY_COMPILE_FLAGS
UNFILTERED_COMPILE_FLAGS
```

```
PIC
SYSROOT
INCLUDE_PATHS
QUOTE_INCLUDE_PATHS
SYSTEM_INCLUDE_PATHS
PREPROCESSOR_DEFINES
```

For linking:

```
FORCE_PIC
SYSROOT
STRIP_DEBUG_SYMBOLS
LEGACY_LINK_FLAGS*
```

*With unresolved questions for -PIE removal for linking executable binary.

This API I believe is good enough because:

- Most users don't have to deal with actual build variable names and values.
- Users are free to use CROSSTOOL to support their own custom action types (e.g. CLIF).
- There is only one API that is shared between C++ rules and other rules.
- We retain a way to affect what build variables are exposed (e.g. for migration purposes).
- Users can specify additional flags (copts, linkopts), while their positioning is still in hands of the CROSSTOOL author.
- Removal of certain flags can be controlled using features, allowing users to affect what options get emitted, and allowing crosstool owner to stay under control. However, this is only the case for well-behaving users who talk to the CROSSTOOL author. There's no way of preventing malicious user from removing flags from the result.
- If we add parameters for inputs and outputs, users of this API can stay platform independent. However I don't think this will be useful in practice, since most existing users already hardcode platform specific flags in their Skylark code. In addition, b/70042728 will add support to TreeArtifact inputs into feature configuration, and that will require changes to the API. I'm open to extending this API once Skylark API to the C++ rules is designed to also accept inputs/outputs in a compatible way.

Alternative solutions

Hide feature configuration from the API

Provide only a way to construct command line given action name and other params:

```
cc_common.get_compile_command_line(cppConfiguration, ccToolchain,
enabledFeatures, copts, usePic, *includePaths, defines, action_name)
```

```
cc_common.get_link_command_line(cppConfiguration, ccToolchain,
enabledFeatures, linkopts, action_name)
```

This approach doesn't give much on top of the proposed solution, since feature configuration will have to be exposed for C++ rules anyway.

Introduce new action types

Current action types expect certain build variables to be present. We could introduce new action names similar to cc-flags-make-variable, that will be fully specified in the crosstool without the assumption of build variables presence. These could be (names subject to change):

Generic-c-compile
Generic-c++-compile
Generic-archive
Generic-link

I'm skeptical about usefulness of this approach since users often need to tweak the generated command line and with this approach they would revert to list filtering and appending.

Introduce new Action subclass

If we insisted on crosstool owner to have full control over what flags get emitted, in other words preventing users from removing random flags from the result, we could introduce a new action subclass available to skylark, that will only construct it's command line in the execution phase from given feature_configuration and build variables. This is how C++ actions operate currently, and these actions will eventually need to be exposed to skylark, however this is still years away, and also is too big impact to the general Skylark API. Also it doesn't prevent users from removing flags in the execution phase using wrapper scripts. It also doesn't solve the problem for e.g. Go rules, that need to pass generated command line down to the third party tool in the form of environment variables or params file.