C++ rules skylark migration plan

Author: hlopko@google.com
Status: Being implemented
Last Updated: 2018-06-20

Goals

- C++ rules are released independently from “core” blaze/bazel.
- Blaze/Bazel binary is smaller since the tools required by the rules are no longer embedded in the binary.
- C++ rules are owned by experts (C++ language platforms team) with clear separation of responsibilities (Skylark rule api as the dividing line).
- We proved Skylark powerful enough (“if we can write rules for C++, we can write rules for anything”).
- Correctness/hermeticity of C++ rules is implied by Skylark language and API design.
- We want Skylark to be the common language for rule definition and to share the common APIs with other build systems so the Skylark rules can be reused. If we are successful, having C++ rules written in Skylark will enable other build systems to use them.

Key Results

- We removed native rule code and we also removed supporting code (e.g. TransitiveInfoProvider currently has 2 different apis, one used by native rules, and one exposed to Skylark). The Bazel codebase is healthier.
- We implemented C++ sandwich. That is a requirement for other rules to be migrated to Skylark or integrated with C++. This is useful because:
    - Internal and external go rules could be unified into single Skylark rules.
    - Open source CLIF rules could be implemented (this is important for Tensorflow who right now has to support their own CLIF Skylark rules with limited capabilities), internal clif rules could be rewritten to Skylark.
    - Objc is currently heavily integrated into C++ rules, cc_toolchain, crosstool, and especially C++ actions, having sandwich expressive enough is a requirement for rewriting them in Skylark.
    - Android/NativeDeps and Python/Swig cannot be rewritten without C++ sandwich.
    - All existing Skylark rules could integrate with C++. This will enable all existing Skylark rules to accept native C++ dependencies, or to embed programs in different languages into C++ code.

Risks

Google3 and Hyrum's law

It is very very risky to try to implement this in isolation and then replace native C++ rules in one go. Due to the number of users we have, and their varying requirements (and creativity) a lot of the unintended C++ rules behavior is depended upon. Even with best effort incremental approach there is a big risk of google3 substantially slowing down the progress to the extent that we would have to abandon the effort.

C++ rules support

Non negligible amount of work is required to support C++ rules and to keep google3 going.

Performance

C++ rules are memory-use sensitive. I think we have good understanding of the memory-critical components so I don't consider the memory use to be a big risk. But their CPU requirements are unclear. It's expected that the CPU use will regress with rules written in Skylark, the question is how much can we regress before it becomes a problem.

Commitment of related parties

The blaze/bazel team. Assuming the current promised headcount for this work (<=1.9) and multiyear expected duration there is a very big risk of new priorities (feature work, performance) coming up slowing the progress down or stopping the progress completely. Anecdata: there were multiple attempts to simplify C++ rules implementation, but none of them was fully completed (because of the complexity, lack of commitment, and other priorities). In the last team meeting we identified unfinished projects as one of the main problems of blaze/bazel.
The Skylark team. C++ rules will require changes to existing Skylark APIs and creation of new APIs. Skylark team must be willing to accept that the API surface will be bigger and more complex. So far none of the existing Skylark rules were used at the scale comparable to the C++ rules. Having C++ rules in Skylark will require performance work, that will make APIs more complex.
Other rules teams. Teams that currently hook their rules into java implementation of C++ rules (go, swig, clif, java, python, objc) need to migrate themselves away once C++ sandwich is implemented. We cannot remove C++ rules code before these rules are using Skylark API exclusively.

The language platform team. As the team that would inherit the Skylark rules, might decide that the complexity and performance requirements outweighs the benefits of Skylark. Also the transfer of ownership must be planned and staffed.

Mitigation

- Incremental transition, no distractions (ideally feature freeze).
- Strong support from the higher leadership so we can push back on feature requests.
- Dedicated support from the Skylark team.
- Dedicated staffing in the LPT team.
- Commitment from “other” rules owners to migrate over to the Skylark API.

Technical challenges

Differences between internal and external rules

We either need to unify internal and external rules, or we need a mechanism to support both from Skylark. Viability of either is not clear.

List of different behaviors for internal C++ rules, external C++ rules, and C++ rules used for ObjC follows.
Include scanning is not available externally. It’s unclear whether we want to open source our include scanner, but since it has custom hints specific to google3, swig, for this discussion I assume we don’t want to open source it as part of the migration effort. This is only relevant when we decide we need to rewrite C++ actions in Skylark. Then we would need “Mutable Actions” Skylark API.
ObjC coupling. Different dependencies/inputs are added to the actions created for objc and for pure C++. This needs to be taken into account when designing C++ Sandwich.
SWIG support is not available externally. Ideally we would design the C++ Sandwich in a way that enables SWIG rules to be rewritten easily.
Hdrs_check is not available externally (or in other words it’s implicitly strict for bazel, but can be loose/warn internally). Either we migrate google3 to ‘strict’ and remove this logic (likely too costly), or expose ‘loose’/’warn’ to Bazel.
Fake C++ rules are not available externally. We identified only 115 instances in google3. We need to investigate removal of these rules, or replacing them with Skylark rules.

Skylark API

C++ rules infect multiple areas of blaze/bazel codebase, many of which have no Skylark API and this API needs to be created:

- Include scanning circumnavigates Skyframe by updating action inputs after the execution phase. Dotd pruning, ‘show includes’ pruning for MSVC, Header Thinning for ObjC are all variations on the common theme, but there is no universal abstraction (yet).
- cc_toolchain is currently implemented in java, but if we want to get rid of internal interfaces, we need to migrate its behavior to Skylark as well.
- C++ rules use their own action implementations, not SpawnActions. Only SpawnAction (TODO verify) are accessible from Skylark. We either need to make SpawnAction more powerful and migrate C++ actions to it, or expose C++ specific actions to Skylark.

Performance

C++ rules are already under pressure to decrease their memory consumption. Some internal projects OOM as soon as we regress memory retained by C++ rules by 5% (TODO Can we get a real number? This is a guess). 5% roughly translates to 40 new fields in C++ actions, less if actions retain more objects. Anecdata: My cls were rolled back because they regressed 0.2% on //gws:gws (12mb on 6gb heap).
Current Skylark actions use either plain list of strings or CustomCommandLine for command lines. C++ rules use feature configuration and build variables to construct the command lines. This is another memory-sensitive area that cannot regress and CustomCommandLine in its current form is not powerful enough. There was (was because the current ordering of projects is unclear before reorg is finished) planned experimental work for Q4/2017 to see if we can replace entire C++ command line generation with a skylark function.
Current mechanism to pass data to the feature configuration is build variables. As a memory optimization these support lazy evaluation/expansion. How to implement this in Skylark is unclear.

Nested sets

Ongoing memory optimization work will possibly require more nested set implementations supporting filter/map operations and exposing them to Skylark (we don’t use nested sets for object files, but we should).

Blaze/Bazel Options, features

C++ rules make use of ~ 60 options that would need to be somehow accessible to Skylark. If we agree that decoupling release process of C++ rules and Blaze/Bazel is valuable, we also need to enable Skylark C++ rules to define custom options. There is a mechanism called Features that could be used, their problem is however that they are only boolean-like (feature is present or is not).

The Plan

I plan to have a more detailed plan :) Once I understand the key results and investigate the challenges further.
List of projects follows, with order not set in stone.

Sandwich

This covers exposing C++ provider(s) to Skylark and opening up the C++ actions API. Exposing actions and at the same time having solid API will be tricky because of include scanning, and feature configuration/build variables (if they were not removed by the cc_toolchain in skylark step). Currently actions require CppSemantics instance and it's unclear how to access it in Skylark (and if we should at all).
We need to make sure all existing rules that depend on C++ can migrate to sandwich (Python - swig, Proto, Java, Clif, Go, Objc, Android - native deps, transitively J2ObjC). Sandwich for ObjC will be especially challenging, since their implementation is heavily intertwined into every place of C++ rules implementation. There is a possibility that we won't be able to clearly separate C++ actions from ObjC, and the result will be that the C++ action API will carry a lot of baggage from ObjC.
Uneducated estimation is that this will take 1.5 engineer-years (Java sandwich took 9 engineer-months, and my crystal ball says C++ sandwich will take at least twice as much) not including the objc migration.

cc_toolchain to Skylark

Rewriting current cc toolchain in Skylark will enable easy access to the toolchain attributes from sandwich. Command line construction, currently done using Feature configuration and build variables will be (if experiments show it's a better solution) replaced by skylark function. This will simplify C++ action api. Side benefit is easier toolchain specification for all our internal and external users. Of course google3 needs to be migrated to the new implementation. This subproject is performance sensitive.
Uneducated estimation is that this will take 6 engineer-months.

Remove behavior from CppConfiguration

IIUC configurations cannot be created in Skylark, therefore we need to migrate the logic from configurations into cc_toolchain. This sounds feasible and not that complicated. The only slowdown there is our existing skylark interface to cppConfiguration and to ccToolchain will require google3 migration.

Rewrite “above the actions”

We have quite few rules already:

- cc_binary
- cc_fake_binary
- cc_inc_library
- cc_library
- cc_test
- cc_embed_data
- cc_proto_profile_provider
- cc_proto_library

Unless we change our priorities, we plan to add:

- cc_import
- cc_shared_library
- cc_static_library

Cc_import, cc_shared_library, cc_static_library, cc_proto_library don’t have srcs, so they don’t emit compile actions. Cc_import and cc_proto_library only emit properly initialized providers. cc_*_library rules register linking actions, and have diamond detection logic.
Remaining rules are complex and emit multiple compilation and linking actions depending on how the universe is aligned. This is the part where most of the complexity of rules sits. This is the place where differences between internal/external/objc rules are the most painful. This is where cpu and memory optimizations will be required. This is also where we will need to ensure backwards compatibility and possibly we’ll have to do google3 migrations.
The possible way forward is to start with Skylark wrappers around native rules that only delegate to the java implementation. Then incrementally migrate slices of behavior into skylark.
Uneducated estimation is 2 engineer-years.

Migrate C++ actions

TODO: do we need to do this at all? Can’t we live with existing actions exposed to Skylark?
Spawn action needs to be extended to support the requirements of C++ actions. Their memory usage cannot be substantially bigger than it is currently.
Uneducated estimation is 1 engineer-year.