Moving LLVM Projects to GitHub

Table of Contents

Introduction

This is a proposal to move our current revision control system from our own hosted Subversion to GitHub. Below are the financial and technical arguments as to why we are proposing such a move and how people (and validation infrastructure) will continue to work with a Git-based LLVM.

There will be a survey pointing at this document which we'll use to gauge the community's reaction and, if we collectively decide to move, the time-frame. Be sure to make your view count.

Additionally, we will discuss this during a BoF at the next US LLVM Developer meeting (http://llvm.org/devmtg/2016-11/).

What This Proposal is *Not* About

Changing the development policy.

This proposal relates only to moving the hosting of our source-code repository from SVN hosted on our own servers to Git hosted on GitHub. We are not proposing using GitHub's issue tracker, pull-requests, or code-review.

Contributers will continue to earn commit access on demand under the Developer Policy, except that that a GitHub account will be required instead of SVN username/password-hash.

Why Git, and Why GitHub?

Why Move At All?

This discussion began because we currently host our own Subversion server and Git mirror on a voluntary basis. The LLVM Foundation sponsors the server and provides limited support, but there is only so much it can do.

Volunteers are not sysadmins themselves, but compiler engineers that happen to know a thing or two about hosting servers. We also don't have 24/7 support, and we sometimes wake up to see that continuous integration is broken because the SVN server is either down or unresponsive.

We should take advantage of one of the services out there (GitHub, GitLab, and BitBucket, among others) that offer better service (24/7 stability, disk space, Git server, code browsing, forking facilities, etc) for free.

Why Git?

Many new coders nowadays start with Git, and a lot of people have never used SVN, CVS, or anything else. Websites like GitHub have changed the landscape of open source contributions, reducing the cost of first contribution and fostering collaboration.

Git is also the version control many LLVM developers use. Despite the sources being stored in a SVN server, these developers are already using Git through the Git-SVN integration.

Git allows you to:

- Commit, squash, merge, and fork locally without touching the remote server.
- Maintain local branches, enabling multiple threads of development.
- Collaborate on these branches (e.g. through your own fork of llvm on GitHub).
- Inspect the repository history (blame, log, bisect) without Internet access.
- Maintain remote forks and branches on Git hosting services and integrate back to the main repository.

In addition, because Git seems to be replacing many OSS projects' version control systems, there are many tools that are built over Git. Future tooling may support Git first (if not only).

Why GitHub?

GitHub, like GitLab and BitBucket, provides free code hosting for open source projects. Any of these could replace the code-hosting infrastructure that we have today.

These services also have a dedicated team to monitor, migrate, improve and distribute the contents of the repositories depending on region and load.

GitHub has one important advantage over GitLab and BitBucket: it offers read-write **SVN** access to the repository (https://github.com/blog/626-announcing-svn-support). This would enable people to continue working post-migration as though our code were still canonically in an SVN repository.

In addition, there are already multiple LLVM mirrors on GitHub, indicating that part of our community has already settled there.

On Managing Revision Numbers with Git

The current SVN repository hosts all the LLVM sub-projects alongside each other. A single revision number (e.g. r123456) thus identifies a consistent version of all LLVM sub-projects.

Git does not use sequential integer revision number but instead uses a hash to identify each commit. (Linus mentioned that the lack of such revision number is "the only real design mistake" in Git [TorvaldRevNum].)

The loss of a sequential integer revision number has been a sticking point in past discussions about Git:

- "The 'branch' I most care about is mainline, and losing the ability to say 'fixed in r1234' (with some sort of monotonically increasing number) would be a tragic loss." [LattnerRevNum]
- "I like those results sorted by time and the chronology should be obvious, but timestamps are incredibly cumbersome and make it difficult to verify that a given checkout matches a given set of results." [TrickRevNum]
- "There is still the major regression with unreadable version numbers. Given the amount of Bugzilla traffic with 'Fixed in...', that's a non-trivial issue." [JSonnRevNum]
- "Sequential IDs are important for LNT and llvmlab bisection tool." [MatthewsRevNum].

However, Git can emulate this increasing revision number: *git rev-list –count <commit-hash>*. This identifier is unique only within a single branch, but this means the tuple *(num, branch-name)* uniquely identifies a commit.

We can thus use this revision number to ensure that e.g. *clang -v* reports a user-friendly revision number (e.g. *master-12345* or *4.0-5321*), addressing the objections raised above with respect to this aspect of Git.

What About Branches and Merges?

In contrast to SVN, Git makes branching easy. Git's commit history is represented as a DAG, a departure from SVN's linear history. However, we propose to mandate making merge commits illegal in our canonical Git repository.

Unfortunately, GitHub does not support server side hooks to enforce such a policy. We must rely on the community to avoid pushing merge commits.

GitHub offers a feature called *Status Checks*: a branch protected by *status checks* requires commits to be whitelisted before the push can happen. We could supply a pre-push hook on the client side that would run and check the history, before whitelisting the commit being pushed [statuschecks]. However this solution would be somewhat fragile (how do you update a script installed on every developer machine?) and prevents SVN access to the repository.

What About Commit Emails?

We will need a new bot to send emails for each commit. This proposal leaves the email format unchanged besides the commit URL.

Straw Man Migration Plan

Step #1 : Before The Move

1. Update docs to mention the move, so people are aware of what is going on.
2. Set up a read-only version of the GitHub project, mirroring our current SVN repository.
3. Add the required bots to implement the commit emails, as well as the umbrella repository update (if the multirepo is selected) or the read-only Git views for the sub-projects (if the monorepo is selected).

Step #2 : Git Move

4. Update the buildbots to pick up updates and commits from the GitHub repository. Not all bots have to migrate at this point, but it'll help provide infrastructure testing.
5. Update Phabricator to pick up commits from the GitHub repository.
6. LNT and llvmlab have to be updated: they rely on unique monotonically increasing integer across branch [MatthewsRevNum].
7. Instruct downstream integrators to pick up commits from the GitHub repository.
8. Review and prepare an update for the LLVM documentation.

Until this point nothing has changed for developers, it will just boil down to a lot of work for buildbot and other infrastructure owners.

The migration will pause here until all dependencies have cleared, and all problems have been solved.

Step #3: Write Access Move

9. Collect developers' GitHub account information, and add them to the project.
10. Switch the SVN repository to read-only and allow pushes to the GitHub repository.
11. Update the documentation.
12. Mirror Git to SVN.

Step #4 : Post Move

13. Archive the SVN repository.
14. Update links on the LLVM website pointing to viewvc/klaus/phab etc. to point to GitHub instead.

One or Multiple Repositories?

There are two major variants for how to structure our Git repository: The "multirepo" and the "monorepo".

Multirepo Variant

This variant recommends moving each LLVM sub-project to a separate Git repository. This mimics the existing official read-only Git repositories (e.g., http://llvm.org/git/compiler-rt.git), and creates new canonical repositories for each sub-project.

This will allow the individual sub-projects to remain distinct: a developer interested only in compiler-rt can checkout only this repository, build it, and work in isolation of the other sub-projects.

A key need is to be able to check out multiple projects (i.e. lldb+clang+llvm or clang+llvm+libcxx for example) at a specific revision.

A tuple of revisions (one entry per repository) accurately describes the state across the sub-projects. For example, a given version of clang would be *<LLVM-12345, clang-5432, libcxx-123, etc.>*.

Umbrella Repository

To make this more convenient, a separate *umbrella* repository will be provided. This repository will be used for the sole purpose of understanding the sequence in which commits were pushed to the different repositories and to provide a single revision number.

This umbrella repository will be read-only and continuously updated to record the above tuple. The proposed form to record this is to use Git [submodules], possibly along with a set of scripts to help check out a specific revision of the LLVM distribution.

A regular LLVM developer does not need to interact with the umbrella repository – the individual repositories can be checked out independently – but you would need to use the umbrella repository to bisect multiple sub-projects at the same time, or to check-out old revisions of LLVM with another sub-project at a consistent state.

This umbrella repository will be updated automatically by a bot (running on notice from a webhook on every push, and periodically) on a per commit basis: a single commit in the umbrella repository would match a single commit in a sub-project.

Living Downstream

Downstream SVN users can use the read/write SVN bridges with the following caveats:

- Be prepared for a one-time change to the upstream revision numbers.
- The upstream sub-project revision numbers will no longer be in sync.

Downstream Git users can continue without any major changes, with the minor change of upstreaming using `git push` instead of `git svn dcommit`.

Git users also have the option of adopting an umbrella repository downstream. The tooling for the upstream umbrella can easily be reused for downstream needs, incorporating extra sub-projects and branching in parallel with sub-project branches.

Multirepo Preview

As a preview (disclaimer: this rough prototype, not polished and not representative of the final solution), you can look at the following:

- Repository: https://github.com/llvm-beanz/llvm-submodules
- Update bot: http://beanz-bot.com:8180/jenkins/job/submodule-update/

Concerns

- Because GitHub does not allow server-side hooks, and because there is no "push timestamp" in Git, the umbrella repository sequence isn't totally exact: commits from different repositories pushed around the same time can appear in different orders. However, we don't expect it to be the common case or to cause serious issues in practice.
- You can't have a single cross-projects commit that would update both LLVM and other sub-projects (something that can be achieved now). It would be possible to establish a protocol whereby users add a special token to their commit messages that causes the umbrella repo's updater bot to group all of them into a single revision.
- Another option is to group commits that were pushed closely enough together in the umbrella repository. This has the advantage of allowing cross-project commits, and is less sensitive to mis-ordering commits. However, this has the potential to group unrelated commits together, especially if the bot goes down and needs to catch up.
- This variant relies on heavier tooling. But the current prototype shows that it is not out-of-reach.
- Submodules don't have a good reputation / are complicating the command line. However, in the proposed setup, a regular developer will seldom interact with submodules directly, and certainly never update them.
- Refactoring across projects is not friendly: taking some functions from clang to make it part of a utility in libSupport wouldn't carry the history of the code in the llvm repo, preventing recursively applying `git blame` for instance. However, this is not very different than the current state.

Workflows

- Checkout/Clone a Single Project, without Commit Access.

- Checkout/Clone a Single Project, with Commit Access.
- Checkout/Clone Multiple Projects, with Commit Access.
- Commit an API Change in LLVM and Update the Sub-projects.
- Branching/Stashing/Updating for Local Development or Experiments.
- Bisecting.

Monorepo Variant

This variant recommends moving all LLVM sub-projects to a single Git repository, similar to https://github.com/llvm-project/llvm-project. This would mimic an export of the current SVN repository, with each sub-project having its own top-level directory. Not all sub-projects are used for building toolchains. In practice, www/ and test-suite/ will probably stay out of the monorepo.

Putting all sub-projects in a single checkout makes cross-project refactoring naturally simple:

- New sub-projects can be trivially split out for better reuse and/or layering (e.g., to allow libSupport and/or LIT to be used by runtimes without adding a dependency on LLVM).
- Changing an API in LLVM and upgrading the sub-projects will always be done in a single commit, designing away a common source of temporary build breakage.
- Moving code across sub-project (during refactoring for instance) in a single commit enables accurate `git blame` when tracking code change history.
- Tooling based on `git grep` works natively across sub-projects, allowing to easier find refactoring opportunities across projects (for example reusing a datastructure initially in LLDB by moving it into libSupport).
- Having all the sources present encourages maintaining the other sub-projects when changing API.

Finally, the monorepo maintains the property of the existing SVN repository that the sub-projects move synchronously, and a single revision number (or commit hash) identifies the state of the development across all projects.

Building a single sub-project

Nobody will be forced to build unnecessary projects. The exact structure is TBD, but making it trivial to configure builds for a single sub-project (or a subset of sub-projects) is a hard requirement.

As an example, it could look like the following:

```
mkdir build && cd build
# Configure only LLVM (default)
cmake path/to/monorepo
# Configure LLVM and lld
cmake path/to/monorepo -DLLVM_ENABLE_PROJECTS=lld
# Configure LLVM and clang
cmake path/to/monorepo -DLLVM_ENABLE_PROJECTS=clang
```

Read/write sub-project mirrors

With the Monorepo, the existing single-subproject mirrors (e.g. http://llvm.org/git/compiler-rt.git) with git-svn read-write access would continue to be maintained: developers would continue to be able to use the existing single-subproject git repositories as they do today, with *no changes to workflow*. Everything (git fetch, git svn dcommit, etc.) could continue to work identically to how it works today.

The monorepo can be set-up such that the SVN revision number matches the SVN revision in the GitHub SVN-bridge.

Living Downstream

Downstream SVN users can use the read/write SVN bridge. The SVN revision number can be preserved in the monorepo, minimizing the impact.

Downstream Git users can continue without any major changes, by using the git-svn mirrors on top of the SVN bridge.

Git users can also work upstream with monorepo even if their downstream fork has split repositories. They can apply patches in the appropriate subdirectories of the monorepo using, e.g., `git am –directory=...`, or plain `diff` and `patch`.

Alternatively, Git users can migrate their own fork to the monorepo. As a demonstration, we've migrated the "CHERI" fork to the monorepo in two ways:

- Using a script that rewrites history (including merges) so that it looks like the fork always lived in the monorepo [LebarCHERI]. The upside of this is when you check out an old revision, you get a copy of all llvm sub-projects at a consistent revision. (For instance, if it's a clang fork, when you check out an old revision you'll get a consistent version of llvm proper.) The downside is that this changes the fork's commit hashes.
- Merging the fork into the monorepo [AminiCHERI]. This preserves the fork's commit hashes, but when you check out an old commit you only get the one sub-project.

Monorepo Preview

As a preview (disclaimer: this rough prototype, not polished and not representative of the final solution), you can look at the following:

- Full Repository: https://github.com/joker-eph/llvm-project
- Single sub-project view with *SVN write access* to the full repo: https://github.com/joker-eph/compiler-rt

Concerns

- Using the monolithic repository may add overhead for those contributing to a standalone sub-project, particularly on runtimes like libcxx and compiler-rt that don't rely on LLVM; currently, a fresh clone of libcxx is only 15MB (vs. 1GB for the monorepo), and the commit rate of LLVM may cause more frequent `git push` collisions when upstreaming. Affected contributors can continue to use the SVN bridge or the single-subproject Git mirrors with git-svn for read-write. Note that this is not a concern for downstream consumers that don't need commit access.
- Preservation of the existing read/write SVN-based workflows relies on the GitHub SVN bridge, which is an extra dependency. Maintaining this locks us into GitHub and could restrict future workflow changes.

Workflows

- Checkout/Clone a Single Project, without Commit Access.

- Checkout/Clone a Single Project, with Commit Access.
- Checkout/Clone Multiple Projects, with Commit Access.
- Commit an API Change in LLVM and Update the Sub-projects.
- Branching/Stashing/Updating for Local Development or Experiments.
- Bisecting.

Multi/Mono Hybrid Variant

This variant recommends moving only the LLVM sub-projects that are *rev-locked* to LLVM into a monorepo (clang, lld, lldb, ...), following the multirepo proposal for the rest. While neither variant recommends combining sub-projects like www/ and test-suite/ (which are completely standalone), this goes further and keeps sub-projects like libcxx and compiler-rt in their own distinct repositories.

Concerns

- This has most disadvantages of multirepo and monorepo, without bringing many of the advantages.
- Downstream have to upgrade to the monorepo structure, but only partially. So they will keep the infrastructure to integrate the other separate sub-projects.
- All projects that use LIT for testing are effectively rev-locked to LLVM. Furthermore, some runtimes (like compiler-rt) are rev-locked with Clang. It's not clear where to draw the lines.

Workflow Before/After

This section goes through a few examples of workflows, intended to illustrate how end-users or developers would interact with the repository for various use-cases.

Checkout/Clone a Single Project, without Commit Access

Except the URL, nothing changes. The possibilities today are:

```
svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm
# or with Git
git clone http://llvm.org/git/llvm.git
```

After the move to GitHub, you would do either:

```
git clone https://github.com/llvm-project/llvm.git
# or using the GitHub svn native bridge
svn co https://github.com/llvm-project/llvm/trunk
```

The above works for both the monorepo and the multirepo, as we'll maintain the existing read-only views of the individual sub-projects.

Checkout/Clone a Single Project, with Commit Access

Currently

```
# direct SVN checkout
svn co https://user@llvm.org/svn/llvm-project/llvm/trunk llvm
# or using the read-only Git view, with git-svn
```

```
git clone http://llvm.org/git/llvm.git
cd llvm
git svn init https://llvm.org/svn/llvm-project/llvm/trunk --username=<username>
git config svn-remote.svn.fetch :refs/remotes/origin/master
git svn rebase -l  # -l avoids fetching ahead of the git mirror.
```

Commits are performed using `svn commit` or with the sequence `git commit` and `git svn dcommit`.

Multirepo Variant

With the multirepo variant, nothing changes but the URL, and commits can be performed using `svn commit` or `git commit` and `git push`:

```
git clone https://github.com/llvm/llvm.git llvm
# or using the GitHub svn native bridge
svn co https://github.com/llvm/llvm/trunk/ llvm
```

Monorepo Variant

With the monorepo variant, there are a few options, depending on your constraints. First, you could just clone the full repository:

```
git clone https://github.com/llvm/llvm-projects.git llvm
# or using the GitHub svn native bridge
svn co https://github.com/llvm/llvm-projects/trunk/ llvm
```

At this point you have every sub-project (llvm, clang, lld, lldb, ...), which doesn't imply you have to build all of them. You can still build only compiler-rt for instance. In this way it's not different from someone who would check out all the projects with SVN today.

You can commit as normal using `git commit` and `git push` or `svn commit`, and read the history for a single project (`git log libcxx` for example).

Secondly, there are a few options to avoid checking out all the sources.

Using the GitHub SVN bridge

The GitHub SVN native bridge allows to checkout a subdirectory directly:

> svn co https://github.com/llvm/llvm-projects/trunk/compiler-rt compiler-rt —username=...

This checks out only compiler-rt and provides commit access using "svn commit", in the same way as it would do today.

Using a Subproject Git Nirror

You can use *git-svn* and one of the sub-project mirrors:

```
# Clone from the single read-only Git repo
git clone http://llvm.org/git/llvm.git
cd llvm
# Configure the SVN remote and initialize the svn metadata
$ git svn init https://github.com/joker-eph/llvm-project/trunk/llvm —username=...
git config svn-remote.svn.fetch :refs/remotes/origin/master
git svn rebase -l
```

In this case the repository contains only a single sub-project, and commits can be made using `git` *svn dcommit*, again exactly as we do today.

Using a Sparse Checkouts

You can hide the other directories using a Git sparse checkout:

```
git config core.sparseCheckout true
echo /compiler-rt > .git/info/sparse-checkout
git read-tree -mu HEAD
```

The data for all sub-projects is still in your *.git* directory, but in your checkout, you only see *compiler-rt*. Before you push, you'll need to fetch and rebase (*git pull –rebase*) as usual.

Note that when you fetch you'll likely pull in changes to sub-projects you don't care about. If you are using spasre checkout, the files from other projects won't appear on your disk. The only effect is that your commit hash changes.

You can check whether the changes in the last fetch are relevant to your commit by running:

```
git log origin/master@{1}..origin/master -- libcxx
```

This command can be hidden in a script so that *git llvmpush* would perform all these steps, fail only if such a dependent change exists, and show immediately the change that prevented the push. An immediate repeat of the command would (almost) certainly result in a successful push. Note that today with SVN or git-svn, this step is not possible since the "rebase" implicitly happens while committing (unless a conflict occurs).

Checkout/Clone Multiple Projects, with Commit Access

Let's look how to assemble llvm+clang+libcxx at a given revision.

Currently

```
svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm -r $REVISION
cd llvm/tools
svn co http://llvm.org/svn/llvm-project/clang/trunk clang -r $REVISION
cd ../projects
svn co http://llvm.org/svn/llvm-project/libcxx/trunk libcxx -r $REVISION
```

Or using git-svn:

```
git clone http://llvm.org/git/llvm.git
cd llvm/
git svn init https://llvm.org/svn/llvm-project/llvm/trunk --username=<username>
git config svn-remote.svn.fetch :refs/remotes/origin/master
git svn rebase -l
git checkout `git svn find-rev -B r258109`
cd tools
git clone http://llvm.org/git/clang.git
cd clang/
git svn init https://llvm.org/svn/llvm-project/clang/trunk --username=<username>
git config svn-remote.svn.fetch :refs/remotes/origin/master
git svn rebase -l
git checkout `git svn find-rev -B r258109`
cd ../../projects/
git clone http://llvm.org/git/libcxx.git
cd libcxx
git svn init https://llvm.org/svn/llvm-project/libcxx/trunk --username=<username>
git config svn-remote.svn.fetch :refs/remotes/origin/master
git svn rebase -l
git checkout `git svn find-rev -B r258109`
```

Note that the list would be longer with more sub-projects.

Multirepo Variant

With the multirepo variant, the umbrella repository will be used. This is where the mapping from a single revision number to the individual repositories revisions is stored.:

```
git clone https://github.com/llvm-beanz/llvm-submodules
cd llvm-submodules
git checkout $REVISION
git submodule init
git submodule update clang llvm libcxx
# the list of sub-project is optional, `git submodule update` would get them all.
```

At this point the clang, llvm, and libcxx individual repositories are cloned and stored alongside each other. There are CMake flags to describe the directory structure; alternatively, you can just symlink *clang* to *llvm/tools/clang*, etc.

Another option is to checkout repositories based on the commit timestamp:

```
git checkout `git rev-list -n 1 --before="2009-07-27 13:37" master`
```

Monorepo Variant

The repository contains natively the source for every sub-projects at the right revision, which makes this straightforward:

```
git clone https://github.com/llvm/llvm-projects.git llvm-projects
cd llvm-projects
git checkout $REVISION
```

As before, at this point clang, llvm, and libcxx are stored in directories alongside each other.

Commit an API Change in LLVM and Update the Sub-projects

Today this is possible, even though not common (at least not documented) for subversion users and for git-svn users. For example, few Git users try to update LLD or Clang in the same commit as they change an LLVM API.

The multirepo variant does not address this: one would have to commit and push separately in every individual repository. It would be possible to establish a protocol whereby users add a special token to their commit messages that causes the umbrella repo’s updater bot to group all of them into a single revision.

The monorepo variant handles this natively.

Branching/Stashing/Updating for Local Development or Experiments

Currently

SVN does not allow this use case, but developers that are currently using git-svn can do it. Let’s look in practice what it means when dealing with multiple sub-projects.

To update the repository to tip of trunk:

```
git pull
cd tools/clang
git pull
cd ../../projects/libcxx
git pull
```

To create a new branch:

```
git checkout -b MyBranch
cd tools/clang
git checkout -b MyBranch
cd ../../projects/libcxx
git checkout -b MyBranch
```

To switch branches:

```
git checkout AnotherBranch
cd tools/clang
git checkout AnotherBranch
cd ../../projects/libcxx
git checkout AnotherBranch
```

Multirepo Variant

The multirepo works the same as the current Git workflow: every command needs to be applied to each of the individual repositories. However, the umbrella repository makes this easy using *`git submodule foreach`* to replicate a command on all the individual repositories (or submodules in this case):

To create a new branch:

```
git submodule foreach git checkout -b MyBranch
```

To switch branches:

```
git submodule foreach git checkout AnotherBranch
```

Monorepo Variant

Regular Git commands are sufficient, because everything is in a single repository:

To update the repository to tip of trunk:

```
git pull
```

To create a new branch:

```
git checkout -b MyBranch
```

To switch branches:

```
git checkout AnotherBranch
```

Bisecting

Assuming a developer is looking for a bug in clang (or lld, or lldb, ...).

Currently

SVN does not have builtin bisection support, but the single revision across sub-projects makes it possible to script around.

Using the existing Git read-only view of the repositories, it is possible to use the native Git bisection script over the llvm repository, and use some scripting to synchronize the clang repository to match the llvm revision.

Multirepo Variant

With the multi-repositories variant, the cross-repository synchronization is achieved using the umbrella repository. This repository contains only submodules for the other sub-projects. The native Git bisection can be used on the umbrella repository directly. A subtlety is that the bisect script itself needs to make sure the submodules are updated accordingly.

For example, to find which commit introduces a regression where clang-3.9 crashes but not clang-3.8 passes, one should be able to simply do:

```
git bisect start release_39 release_38
git bisect run ./bisect_script.sh
```

With the *bisect_script.sh* script being:

```
#!/bin/sh
cd $UMBRELLA_DIRECTORY
git submodule update llvm clang libcxx #....
cd $BUILD_DIR

ninja clang || exit 125   # an exit code of 125 asks "git bisect"
                          # to "skip" the current commit

./bin/clang some_crash_test.cpp
```

When the *git bisect run* command returns, the umbrella repository is set to the state where the regression is introduced. The commit diff in the umbrella indicate which submodule was updated, and the last commit in this sub-projects is the one that the bisect found.

Monorepo Variant

Bisecting on the monorepo is straightforward, and very similar to the above, except that the bisection script does not need to include the *git submodule update* step.

The same example, finding which commit introduces a regression where clang-3.9 crashes but not clang-3.8 passes, will look like:

```
git bisect start release_39 release_38
git bisect run ./bisect_script.sh
```

With the *bisect_script.sh* script being:

```
#!/bin/sh
cd $BUILD_DIR

ninja clang || exit 125   # an exit code of 125 asks "git bisect"
                          # to "skip" the current commit

./bin/clang some_crash_test.cpp
```

Also, since the monorepo handles commits update across multiple projects, you're less like to encounter a build failure where a commit change an API in LLVM and another later one "fixes" the build in clang.

References

[LattnerRevNum] Chris Lattner, http://lists.llvm.org/pipermail/llvm-dev/2011-July/041739.html

[TrickRevNum] Andrew Trick, http://lists.llvm.org/pipermail/llvm-dev/2011-July/041721.html

[JSonnRevNum] Joerg Sonnenberg, http://lists.llvm.org/pipermail/llvm-dev/2011-July/041688.html

[TorvaldRevNum] Linus Torvald, http://git.661346.n2.nabble.com/Git-commit-generation-numbers-td6584414.html

[MatthewsRevNum] *(1, 2)* Chris Matthews, http://lists.llvm.org/pipermail/cfe-dev/2016-July/049886.html

[submodules] Git submodules, https://git-scm.com/book/en/v2/Git-Tools-Submodules)

[statuschecks] GitHub status-checks, https://help.github.com/articles/about-required-status-checks/

[LebarCHERI] Port *CHERI* to a single repository rewriting history, http://lists.llvm.org/pipermail/llvm-dev/2016-July/102787.html

[AminiCHERI] Port *CHERI* to a single repository preserving history, http://lists.llvm.org/pipermail/llvm-dev/2016-July/102804.html