Debugging The Compiler

This document contains some useful information for debugging:

- The Swift compiler.
- Intermediate output of the Swift Compiler.
- Swift applications at runtime.

Please feel free to add any useful tips that one finds to this document for the benefit of all Swift developers.

Table of Contents

Debugging the Compiler Itself

Basic Utilities

Often, the first step to debug a compiler problem is to re-run the compiler with a command line, which comes from a crash trace or a build log.

The script `split-cmdline` in `utils/dev-scripts` splits a command line into multiple lines. This is helpful to understand and edit long command lines.

Printing the Intermediate Representations

The most important thing when debugging the compiler is to examine the IR. Here is how to dump the IR after the main phases of the Swift compiler (assuming you are compiling with optimizations enabled):

- **Parser** To print the AST after parsing:

```
swiftc -dump-ast -O file.swift
```

- **SILGen** To print the SIL immediately after SILGen:

```
swiftc -emit-silgen -O file.swift
```

- **Mandatory SIL passes** To print the SIL after the mandatory passes:

```
swiftc -emit-sil -Onone file.swift
```

Well, this is not quite true, because the compiler is running some passes for -Onone after the mandatory passes, too. But for most purposes you will get what you want to see.

- **Performance SIL passes** To print the SIL after the complete SIL optimization pipeline:

```
swiftc -emit-sil -O file.swift
```

- **Debug info in SIL** To print debug info from `file.swift` in SIL:

```
swiftc -g -emit-sil -O file.swift
```

- **IRGen** To print the LLVM IR after IR generation:

```
swiftc -emit-ir -Xfrontend -disable-llvm-optzns -O file.swift
```

- **LLVM passes** To print the LLVM IR after LLVM passes:

```
swiftc -emit-ir -O file.swift
```

- **Code generation** To print the final generated code:

```
swiftc -S -O file.swift
```

Compilation stops at the phase where you print the output. So if you want to print the SIL *and* the LLVM IR, you have to run the compiler twice. The output of all these dump options (except `-dump-ast`) can be redirected with an additional `-o <file>` option.

Debugging Diagnostic Emission

Asserting on first emitted Warning/Assert Diagnostic

When changing the type checker and various SIL passes, one can cause a series of cascading diagnostics (errors/warnings) to be emitted. Since Swift does not by default assert when emitting such diagnostics, it becomes difficult to know where to stop in the debugger. Rather than trying to guess/check if one has an asserts swift compiler, one can use the following options to cause the diagnostic engine to assert on the first error/warning:

- `-Xllvm -swift-diagnostics-assert-on-error=1`
- `-Xllvm -swift-diagnostics-assert-on-warning=1`

These allow one to dump a stack trace of where the diagnostic is being emitted (if run without a debugger) or drop into the debugger if a debugger is attached.

Finding Diagnostic Names

Some diagnostics rely heavily on format string arguments, so it can be difficult to find their implementation by searching for parts of the emitted message in the codebase. To print the corresponding diagnostic name at the end of each emitted message, use the `-Xfrontend -debug-diagnostic-names` argument.

Debugging the Type Checker

Enabling Logging

To enable logging in the type checker, use the following argument: `-Xfrontend -debug-constraints`. This will cause the typechecker to log its internal state as it solves constraints and present the final type checked solution, e.g.:

```
---Constraint solving for the expression at [test.swift:3:10 - line:3:10]---
---Initial constraints for the given expression---
(integer_literal_expr type='$T0' location=test.swift:3:10 range=[test.swift:3:10 - line:3:10] value=0)
Score: 0 0 0 0 0 0 0 0 0 0 0 0 0
Contextual Type: Int
Type Variables:
  #0 = $T0 [inout allowed]

Active Constraints:

Inactive Constraints:
  $T0 literal conforms to ExpressibleByIntegerLiteral [[locator@0x7ffa3a865a00 [IntegerLiteral@test.swift:3:10]]];
  $T0 conv Int [[locator@0x7ffa3a865a00 [IntegerLiteral@test.swift:3:10]]];
($T0 literal=3 bindings=(subtypes of) (default from ExpressibleByIntegerLiteral) Int)
Active bindings: $T0 := Int
(trying $T0 := Int
```

```
  (found solution 0 0 0 0 0 0 0 0 0 0 0 0 0)
)
---Solution---
Fixed score: 0 0 0 0 0 0 0 0 0 0 0 0 0
Type variables:
  $T0 as Int

Overload choices:

Constraint restrictions:

Disjunction choices:

Conformances:
  At locator@0x7ffa3a865a00 [IntegerLiteral@test.swift:3:10]
(normal_conformance type=Int protocol=ExpressibleByIntegerLiteral lazy
  (normal_conformance type=Int protocol=_ExpressibleByBuiltinIntegerLiteral lazy))
(found solution 0 0 0 0 0 0 0 0 0 0 0 0 0)
---Type-checked expression---
(call_expr implicit type='Int' location=test.swift:3:10 range=[test.swift:3:10 - line:3:10] arg_labels=_builtinIntegerLiteral:
  (constructor_ref_call_expr implicit type='(_MaxBuiltinIntegerType) -> Int' location=test.swift:3:10 range=[test.swift:3:10 -
line:3:10]
    (declref_expr implicit type='(Int.Type) -> (_MaxBuiltinIntegerType) -> Int' location=test.swift:3:10 range=[test.swift:3:10 -
line:3:10] decl=Swift.(file).Int.init(_builtinIntegerLiteral:) function_ref=single)
    (type_expr implicit type='Int.Type' location=test.swift:3:10 range=[test.swift:3:10 - line:3:10] typerepr='Int'))
  (tuple_expr implicit type='(_builtinIntegerLiteral: Int2048)' location=test.swift:3:10 range=[test.swift:3:10 - line:3:10]
names=_builtinIntegerLiteral
    (integer_literal_expr type='Int2048' location=test.swift:3:10 range=[test.swift:3:10 - line:3:10] value=0)))
```

When using the integrated swift-repl, one can dump the same output for each expression as one evaluates the expression by enabling constraints debugging by typing `:constraints debug on`:

```
$ swift -frontend -repl -enable-objc-interop -module-name REPL
***  You are running Swift's integrated REPL,  ***
***  intended for compiler and stdlib          ***
***  development and testing purposes only.    ***
***  The full REPL is built as part of LLDB.   ***
***  Type ':help' for assistance.              ***
(swift) :constraints debug on
```

Debugging on SIL Level

Options for Dumping the SIL

Often it is not sufficient to dump the SIL at the beginning or end of the optimization pipeline. The SILPassManager supports useful options to dump the SIL also between pass runs.

A short (non-exhaustive) list of SIL printing options:

- `-Xllvm '-sil-print-function=SWIFT_MANGLED_NAME'`: Print the specified function after each pass which modifies the function. Note that for module passes, the function is printed if the pass changed *any* function (the pass manager doesn't know which functions a module pass has changed). Multiple functions can be specified as a comma separated list.
- `-Xllvm '-sil-print-functions=NAME'`: Like `-sil-print-function`, except that functions are selected if NAME is *contained* in their mangled names.
- `-Xllvm -sil-print-all`: Print all functions when ever a function pass modifies a function and print the entire module if a module pass modifies the SILModule.
- `-Xllvm -sil-print-around=$PASS_NAME`: Print the SIL before and after a pass with name `$PASS_NAME` runs on a function or module. By default it prints the whole module. To print only specific functions, add `-sil-print-function` and/or `-sil-print-functions`.
- `-Xllvm -sil-print-before=$PASS_NAME`: Like `-sil-print-around`, but prints the SIL only *before* the specified pass runs.
- `-Xllvm -sil-print-after=$PASS_NAME`: Like `-sil-print-around`, but prints the SIL only *after* the specified pass did run.

NOTE: This may emit a lot of text to stderr, so be sure to pipe the output to a file.

Getting CommandLine for swift stdlib from Ninja to enable dumping stdlib SIL

If one builds swift using ninja and wants to dump the SIL of the stdlib using some of the SIL dumping options from the previous section, one can use the following one-liner:

```
ninja -t commands | grep swiftc | grep Swift.o | grep " -c "
```

This should give one a single command line that one can use for Swift.o, perfect for applying the previous sections options to.

Dumping the SIL and other Data in LLDB

When debugging the Swift compiler with LLDB (or Xcode, of course), there is even a more powerful way to examine the data in the compiler, e.g. the SIL. Following LLVM's dump() convention, many SIL classes (as well as AST classes) provide a dump() function. You can call the dump function with LLDB's `expression --` or `print` or `p` command.

For example, to examine a SIL instruction:

```
(lldb) p Inst->dump()
%12 = struct_extract %10 : $UnsafeMutablePointer<X>, #UnsafeMutablePointer._rawValue // user: %13
```

To dump a whole function at the beginning of a function pass:

```
(lldb) p getFunction()->dump()
```

SIL modules and even functions can get very large. Often it is more convenient to dump their contents into a file and open the file in a separate editor. This can be done with:

```
(lldb) p getFunction()->dump("myfunction.sil")
```

You can also dump the CFG (control flow graph) of a function:

```
(lldb) p Func->viewCFG()
```

This opens a preview window containing the CFG of the function. To continue debugging press -C on the LLDB prompt. Note that this only works in Xcode if the PATH variable in the scheme's environment setting contains the path to the dot tool.

swift/Basic/Debug.h includes macros to help contributors declare these methods with the proper attributes to ensure they'll be available in the debugger. In particular, if you see `SWIFT_DEBUG_DUMP` in a class declaration, that class has a `dump()` method you can call.

Debugging and Profiling on SIL level

SIL source level profiling

The compiler provides a way to debug and profile on SIL level. To enable SIL debugging add the front-end option -sil-based-debuginfo together with -g. Example:

```
swiftc -g -Xfrontend -sil-based-debuginfo -O test.swift -o a.out
```

This writes the SIL after optimizations into a file and generates debug info for it. In the debugger and profiler you can then see the SIL code instead of the Swift source code. For details see the SILDebugInfoGenerator pass.

To enable SIL debugging and profiling for the Swift standard library, use the build-script-impl option `--build-sil-debugging-stdlib`.

ViewCFG: Regex based CFG Printer for SIL/LLVM-IR

ViewCFG (`./utils/viewcfg`) is a script that parses a textual CFG (e.g. a llvm or sil function) and displays a .dot file of the CFG. Since the parsing is done using regular expressions (i.e. ignoring language semantics), ViewCFG can:

1. Parse both SIL and LLVM IR
2. Parse blocks and functions without needing to know contextual information. Ex: types and declarations.

The script assumes that the relevant text is passed in via stdin and uses open to display the .dot file.

Additional, both emacs and vim integration is provided. For vim integration add the following commands to your .vimrc:

```
com! -nargs=? Funccfg silent ?{$?,/^}/w !viewcfg <args>
com! -range -nargs=? Viewcfg silent <line1>,<line2>w !viewcfg <args>
```

This will add:

```
:Funccfg        displays the CFG of the current SIL/LLVM function.
:<range>Viewcfg displays the sub-CFG of the selected range.
```

For emacs users, we provide in sil-mode (`./utils/sil-mode.el`) the function:

```
sil-mode-display-function-cfg
```

To use this feature, placed the point in the sil function that you want viewcfg to graph and then run `sil-mode-display-function-cfg`. This will cause viewcfg to be invoked with the sil function body. Note, `sil-mode-display-function-cfg` does not take any arguments.

NOTE viewcfg must be in the $PATH for viewcfg to work.

NOTE Since we use open, .dot files should be associated with the Graphviz app for viewcfg to work.

There is another useful script to view the CFG of a disassembled function: `./utils/dev-scripts/blockifyasm`. It splits a disassembled function up into basic blocks which can then be used with viewcfg:

```
(lldb) disassemble
  <copy-paste output to file.s>
$ blockifyasm < file.s | viewcfg
```

Debugging the Compiler using advanced LLDB Breakpoints

LLDB has very powerful breakpoints, which can be utilized in many ways to debug the compiler and Swift executables. The examples in this section show the LLDB command lines. In Xcode you can set the breakpoint properties by clicking 'Edit breakpoint'.

Let's start with a simple example: sometimes you see a function in the SIL output and you want to know where the function was created in the compiler. In this case you can set a conditional breakpoint in SILFunction constructor and check for the function name in the breakpoint condition:

```
(lldb) br set -c 'hasName("_TFC3nix1Xd")' -f SILFunction.cpp -l 91
```

Sometimes you may want to know which optimization inserts, removes or moves a certain instruction. To find out, set a breakpoint in `ilist_traits<SILInstruction>::addNodeToList` or `ilist_traits<SILInstruction>::removeNodeFromList`, which are defined in `SILInstruction.cpp`. The following command sets a breakpoint which stops if a `strong_retain` instruction is removed:

```
(lldb) br set -c 'I->getKind() == ValueKind::StrongRetainInst' -f SILInstruction.cpp -l 63
```

The condition can be made more precise e.g. by also testing in which function this happens:

```
(lldb) br set -c 'I->getKind() == ValueKind::StrongRetainInst &&
           I->getFunction()->hasName("_TFC3nix1Xd")'
           -f SILInstruction.cpp -l 63
```

Let's assume the breakpoint hits somewhere in the middle of compiling a large file. This is the point where the problem appears. But often you want to break a little bit earlier, e.g. at the entrance of the optimization's `run` function.

To achieve this, set another breakpoint and add breakpoint commands:

```
(lldb) br set -n GlobalARCOpts::run
Breakpoint 2
(lldb) br com add 2
> p int $n = $n + 1
> c
> DONE
```

Run the program (this can take quite a bit longer than before). When the first breakpoint hits see what value $n has:

```
(lldb) p $n
(int) $n = 5
```

Now remove the breakpoint commands from the second breakpoint (or create a new one) and set the ignore count to $n minus one:

```
(lldb) br delete 2
(lldb) br set -i 4 -n GlobalARCOpts::run
```

Run your program again and the breakpoint hits just before the first breakpoint.

Another method for accomplishing the same task is to set the ignore count of the breakpoint to a large number, i.e.:

```
(lldb) br set -i 9999999 -n GlobalARCOpts::run
```

Then whenever the debugger stops next time (due to hitting another breakpoint/crash/assert) you can list the current breakpoints:

```
(lldb) br list
1: name = 'GlobalARCOpts::run', locations = 1, resolved = 1, hit count = 85 Options: ignore: 1 enabled
```

which will then show you the number of times that each breakpoint was hit. In this case, we know that `GlobalARCOpts::run` was hit 85 times. So, now we know to ignore swift_getGenericMetadata 84 times, i.e.:

```
(lldb) br set -i 84 -n GlobalARCOpts::run
```

A final trick is that one can use the -R option to stop at a relative assembly address in lldb. Specifically, lldb resolves the breakpoint normally and then just adds the argument -R to the address. So for instance, if I want to stop at the address at +38 in the function with the name 'foo', I would write:

```
(lldb) br set -R 38 -n foo
```

Then lldb would add 38 to the offset of foo and break there. This is really useful in contexts where one wants to set a breakpoint at an assembly address that is stable across multiple different invocations of lldb.

Debugging the Compiler using LLDB Scripts

LLDB has powerful capabilities of scripting in Python among other languages. An often overlooked, but very useful technique is the -s command to lldb. This essentially acts as a pseudo-stdin of commands that lldb will read commands from. Each time lldb hits a stopping point (i.e. a breakpoint or a crash/assert), it will run the earliest command that has not been run yet. As an example of this consider the following script (which without any loss of generality will be called test.lldb):

```
env DYLD_INSERT_LIBRARIES=/usr/lib/libgmalloc.dylib
break set -n swift_getGenericMetadata
break mod 1 -i 83
process launch -- --stdlib-unittest-in-process --stdlib-unittest-filter
"DefaultedForwardMutableCollection<OpaqueValue<Int>>.Type.subscript(_: Range)/Set/semantics"
break set -l 224
c
expr pattern->CreateFunction
break set -a $0
c
dis -f
```

TODO: Change this example to apply to the Swift compiler instead of to the stdlib unittests.

Then by running `lldb test -s test.lldb`, lldb will:

1. Enable guard malloc.
2. Set a break point on swift_getGenericMetadata and set it to be ignored for 83 hits.
3. Launch the application and stop at swift_getGenericMetadata after 83 hits have been ignored.
4. In the same file as swift_getGenericMetadata introduce a new breakpoint at line 224 and continue.
5. When we break at line 224 in that file, evaluate an expression pointer.
6. Set a breakpoint at the address of the expression pointer and continue.
7. When we hit the breakpoint set at the function pointer's address, disassemble the function that the function pointer was passed to.

Using LLDB scripts can enable one to use complex debugger workflows without needing to retype the various commands perfectly every time.

Custom LLDB Commands

If you've ever found yourself repeatedly entering a complex sequence of commands within a debug session, consider using custom lldb commands. Custom commands are a handy way to automate debugging tasks.

For example, say we need a command that prints the contents of the register `rax` and then steps to the next instruction. Here's how to define that command within a debug session:

```
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()' or Ctrl-D.
>>> def custom_step():
...   print "rax =", lldb.frame.FindRegister("rax")
...   lldb.thread.StepInstruction(True)
...
>>> ^D
```

You can call this function using the `script` command, or via an alias:

```
(lldb) script custom_step()
rax = ...
<debugger steps to the next instruction>

(lldb) command alias cs script custom_step()
(lldb) cs
rax = ...
<debugger steps to the next instruction>
```

Printing registers and single-stepping are by no means the only things you can do with custom commands. The LLDB Python API surfaces a lot of useful functionality, such as arbitrary expression evaluation.

There are some pre-defined custom commands which can be especially useful while debugging the swift compiler. These commands live in `swift/utils/lldb/lldbToolBox.py`. There is a wrapper script available in `SWIFT_BINARY_DIR/bin/lldb-with-tools` which launches lldb with those commands loaded.

A command named `sequence` is included in lldbToolBox. `sequence` runs multiple semicolon separated commands together as one command. This can be used to define custom commands using just other lldb commands. For example, `custom_step()` function defined above could be defined as:

```
(lldb) command alias cs sequence p/x $rax; stepi
```

Debugging at LLVM Level

Options for Dumping LLVM IR

Similar to SIL, one can configure LLVM to dump the llvm-ir at various points in the pipeline. Here is a quick summary of the various options:

- `-Xllvm -print-before=$PASS_ID` : Print the LLVM IR before a specified LLVM pass runs.
- `-Xllvm -print-before-all` : Print the LLVM IR before each pass runs.
- `-Xllvm -print-after-all` : Print the LLVM IR after each pass runs.
- `-Xllvm -filter-print-funcs=$FUNC_NAME_1,$FUNC_NAME_2,...,$FUNC_NAME_N` : When printing IR for functions for print-[before|after]-all options, Only print the IR for functions whose name is in this comma separated list.

Debugging assembly and object code

Understanding layout of compiler-generated metadata can sometimes involve looking at assembly and object code.

Working with a single file

Here's how to generate assembly or object code:

```
# Emit assembly in Intel syntax (AT&T syntax is the default)
swiftc tmp.swift -emit-assembly -Xllvm -x86-asm-syntax=intel -o tmp.S

# Emit object code
swiftc tmp.swift -emit-object -o tmp.o
```

Understanding mangled names can be hard though: `swift demangle` to the rescue!

```
swiftc tmp.swift -emit-assembly -Xllvm -x86-asm-syntax=intel -o - \
  | swift demangle > tmp-demangled.S

swiftc tmp.swift -emit-object -o tmp.o

# Look at where different symbols are located, sorting by address (-n)
# and displaying section names (-m)
nm -n -m tmp.o | swift demangle > tmp.txt

# Inspect disassembly of an existing dylib (AT&T syntax is the default)
objdump -d -macho --x86-asm-syntax=intel /path/to/libcake.dylib \
  | swift demangle > libcake.S
```

Working with multiple files

Some bugs only manifest in WMO, and may involve complicated Xcode projects. Moreover, Xcode may be passing arguments via `-filelist` and expecting outputs via `-output-filelist`, and those file lists may be in temporary directories.

If you want to inspect assembly or object code for individual files when compiling under WMO, you can mimic this by doing the following:

```
# Assuming all .swift files from the MyProject/Sources directory
# need to be included
find MyProject/Sources -name '*.swift' -type f > input-files.txt

# In some cases, projects may use multiple files with the same
# name but in different directories (for different schemes),
# which can be a problem. Having a file list makes working around
# this convenient as you can manually manually edit out the files
# that are not of interest at this stage.

mkdir Output

# 1. -output-filelist doesn't recreate a subdirectory structure,
#    so first strip out directories
# 2. map .swift files to assembly files
sed -e 's|.*/|Output/|;s|\.swift|.S|' input-files.txt > output-files.txt

# Save command-line arguments from Xcode's 'CompileSwiftSources' phase in
# the build log to a file for convenience, say args.txt.
#
# -sdk /path/to/sdk <... other args ...>

xcrun swift-frontend @args.txt \
  -filelist input-files.txt \
  -output-filelist output-files.txt \
  -O -whole-module-optimization \
  -emit-assembly
```

If you are manually calling `swift-frontend` without an Xcode invocation to use as a template, you will need to at least add `-sdk "$(xcrun --show-sdk-path macosx)"` (if compiling for macOS), and `-I /path/to/includedir` to include necessary swift modules and interfaces.

Working with multi-architecture binaries

On macOS, one might be interested in debugging multi-architecture binaries such as universal binaries. By default `nm` will show symbols from all architectures, so a universal binary might look funny due to two copies of everything. Use `nm -arch` to look at a specific architecture:

```
nm -n -m -arch x86_64 path/to/libcake.dylib | swift demangle
```

Other helpful tools

TODO: This section should mention information about non-macOS platforms: maybe we can have a table with rows for use cases and columns for platforms (macOS, Linux, Windows), and the cells would be tool names. We could also mention platforms next to the tool names.

In the previous sub-sections, we've seen how using different tools can make working with assembly and object code much nicer. Here is a short listing of commonly used tools on macOS, along with some example use cases:

- Miscellaneous:
- `strings`: Find printable strings in a binary file.
- Potential use cases: If you're building a binary in multiple configurations, and forgot which binary corresponds to which configuration, you can look through the output of `strings` to identify differences.
- `c++filt`: The C++ equivalent of `swift-demangle`.
    - Potential use cases: Looking at the generated code for the Swift runtime, investigating C++ interop issues.
- Linking:

- `libtool` : A tool to create static and dynamic libraries. Generally, it's easier to instead ask `swiftc` to link files, but potentially handy as a higher-level alternative to `ld` , `ar` and `lipo` .

- Debug info:
  - `dwarfdump` : Extract debug info in human-readable form.
    - Potential use cases: If you want to quickly check if two binaries are identical, you can compare their UUIDs. For on-disk binaries, you can obtain the UUID using `dwarfdump --uuid` For binaries loaded by a running application, you can obtain the UUID using `image list` in LLDB.

- `objdump` : Dump object files. Some examples of using `objdump` are documented in the previous subsection. If you have a Swift compiler build, you can use `llvm-objdump` from `$LLVM_BUILD_DIR/bin` instead of using the system `objdump` .

  Compared to other tools on this list, `objdump` packs a LOT of functionality; it can show information about sections, relocations and more. It also supports many flags to format and filter the output.

- Linker information (symbol table, sections, binding):
  - `nm` : Display symbol tables. Some examples of using `nm` are documented in the previous subsection.
  - `size` : Get high-level information about sections in a binary, such as the sizes of sections and where they are located.
  - `dyldinfo` : Display information used by dyld, such as which dylibs an image depends on.
  - `install_name_tool` : Change the name for a dynamic shared library, and query or modify the runpath search paths (aka 'rpaths') it uses.

- Multi-architecture binaries:
  - `lipo` : A tool that can be used to create, inspect and dissect universal binaries.
    - Potential use cases: If you have a universal binary on an Apple Silicon Mac, but want to quickly test if the issue would reproduce on `x86_64` , you can extract the `x86_64` slice by using `lipo` . The `x86_64` binary will automatically run under Rosetta 2.

Bisecting Compiler Errors

Bisecting on SIL optimizer pass counts to identify optimizer bugs

If a compiled executable is crashing when built with optimizations, but not crashing when built with -Onone, it's most likely one of the SIL optimizations which causes the miscompile.

Currently there is no tool to automatically identify the bad optimization, but it's quite easy to do this manually:

1. Add the compiler option `-Xllvm -sil-opt-pass-count=<n>` , where `<n>` is the number of optimizations to run.
2. Bisect: find n where the executable crashes, but does not crash with n-1. First just try n = 10, 100, 1000, 10000, etc. to find an upper bound). Then can either bisect the invocation by hand or place the invocation into a script and use `./llvm-project/llvm/utils/bisect` to automatically bisect based on the scripts error code. Example invocation:

   bisect --start=0 --end=10000 ./invoke_swift_passing_N.sh "%(count)s"
3. Add another option `-Xllvm -sil-print-last` . The output can be large, so it's best to redirect stderr to a file (`2> output`). The output contains the SIL before and after the bad optimization.
4. Copy the two functions from the output into separate files and compare both files. Try to figure out what the optimization pass did wrong. To simplify the comparison, it's sometimes helpful to replace all SIL values (e.g. `%27`) with a constant string (e.g. `%x`).
5. If the bad optimization is SILCombine or SimplifyCFG (which do a lot of transformations in a single run) it's helpful to continue bisecting on the sub-pass number. The option `-Xllvm -sil-opt-pass-count=<n>.<m>` can be used for that, where `m` is the sub-pass number.

Using git-bisect in the presence of branch forwarding/feature branches

`git-bisect` is a useful tool for finding where a regression was introduced. Sadly `git-bisect` does not handle long lived branches and will in fact choose commits from upstream branches that may be missing important content from the downstream branch. As an example, consider a situation where one has the following straw man commit flow graph:

```
github/main -> github/tensorflow
```

In this case if one attempts to use `git-bisect` on github/tensorflow, `git-bisect` will sometimes choose commits from github/main resulting in one being unable to compile/test specific tensorflow code that has not been upstreamed yet. Even worse, what if we are trying to bisect in between two that were branched from github/tensorflow and have had subsequent commits cherry-picked on top. Without any loss of generality, lets call those two tags `tag-tensorflow-bad` and `tag-tensorflow-good` . Since both of these tags have had commits cherry-picked on top, they are technically not even on the github/tensorflow branch, but rather in a certain sense are a tag of a feature branch from main/tensorflow. So, `git-bisect` doesn't even have a clear history to bisect on in multiple ways.

With those constraints in mind, we can bisect! We just need to be careful how we do it. Lets assume that we have a test script called `test.sh` that indicates error by the error code. With that in hand, we need to compute the least common ancestor of the good/bad commits. This is traditionally called the "merge base" of the commits. We can compute this as so:

```
TAG_MERGE_BASE=$(git merge-base tags/tag-tensorflow-bad tags/tag-tensorflow-good)
```

Given that both tags were taken from the feature branch, the reader can prove to themselves that this commit is guaranteed to be on `github/tensorflow` and not `github/main` since all commits from `github/main` are forwarded using git merges.

Then lets assume that we checked out `$TAG_MERGE_BASE` and then ran `test.sh` and did not hit any error. Ok, we can not bisect. Sadly, as mentioned above if we run git-bisect in between `$TAG_MERGE_BASE` and `tags/tag-tensorflow-bad` , `git-bisect` will sometimes choose commits from `github/main` which would cause

`test.sh` to fail if we are testing tensorflow specific code! To work around this problem, we need to start our bisect and then tell `git-bisect` to ignore those commits by using the skip sub command:

```
git bisect start tags/tag-tensorflow-bad $TAG_MERGE_BASE
for rev in $(git rev-list $TAG_MERGE_BASE..tags/tag-tensorflow-bad --merges --first-parent); do
    git rev-list $rev^2 --not $rev^
done | xargs git bisect skip
```

Once this has been done, one uses `git-bisect` normally. One thing to be aware of is that `git-bisect` will return a good/bad commits on the feature branch and if one of those commits is a merge from the upstream branch, one will need to analyze the range of commits from upstream for the bad commit afterwards. The commit range in the merge should be relatively small though compared with the large git history one just bisected.

Reducing SIL test cases using bug_reducer

There is functionality provided in ./swift/utils/bug_reducer/bug_reducer.py for reducing SIL test cases by:

1. Producing intermediate sib files that only require some of the passes to trigger the crasher.
2. Reducing the size of the sil test case by extracting functions or partitioning a module into unoptimized and optimized modules.

For more information and a high level example, see: ./swift/utils/bug_reducer/README.md.

Syncing branches during bisects

When bisecting it might be necessary to run the `update-checkout` script each time you change shas. To do this you can pass `--match-timestamp` to automatically checkout match the timestamp of the `apple/swift` repo across the other repos.

Debugging the Compiler Build

Build Dry Run

A "dry-run" invocation of the `build-script` (using the `--dry-run` flag) will print the commands that would be executed in a given build, without executing them. A dry-run script invocation output can be used to inspect the build stages of a given `build-script` configuration, or create script corresponding to one such configuration.

Debugging the Compiler Driver

The Swift compiler uses a standalone compiler-driver application written in Swift: swift-driver. When building the compiler using `build-script`, by default, the standalone driver will be built first, using the host toolchain, if the host toolchain contains a Swift compiler. If the host toolchain does not contain Swift, a warning is emitted and the legacy compiler-driver (integrated in the C++ code-base) will be used. In the future, a host toolchain containing a Swift compiler may become mandatory. Once the compiler is built, the compiler build directory (`swift-<OS>-<ARCH>`) is updated with a symlink to the standalone driver, ensuring calls to the build directory's `swift` and `swiftc` always forward to the standalone driver.

For more information about the driver, see: github.com/apple/swift-driver/blob/main/README.md

Swift Compiler Driver F.A.Q.

> *What's the difference between invoking 'swiftc' vs. 'swift-driver' at the top level?*

Today, `swift` and `swiftc` are symbolic links to the compiler binary (`swift-frontend`). Invoking `swiftc` causes the executable to detects that it is a compiler-driver invocation, and not a direct compiler-frontend invocation, by examining the invoked program's name. The compiler frontend can be invoked directly by invoking the `swift-frontend` executable, or passing in the `-frontend` option to `swiftc`.

The standalone Compiler Driver is installed as a separate `swift-driver` executable in the Swift toolchain's `bin` directory. When a user launches the compiler by invoking `swiftc`, the C++ based compiler executable forwards the invocation to the `swift-driver` executable if one is found alongside it. This forwarding mechanism is in-place temporarily, to allow for an easy fallback to the legacy driver via one of the two escape hatches:

- `-disallow-use-new-driver` command line flag
- `SWIFT_USE_OLD_DRIVER` environment variable

If the user is to directly invoke the `swift-driver` executable, the behaviour should be the same as invoking the `swiftc` executable, but without the option for a legacy driver fallback.

Once the legacy driver is deprecated, `swift` and `swiftc` executables will become symbolic links to the `swift-driver` executable directly.

> *Will 'swiftc ... -###' always print the same set of commands for the old/new driver? Do they call 'swift-frontend' the same way?*

The standalone Compiler Driver is meant to be a direct drop-in replacement for the C++-based legacy driver. It has the exact same command-line interface. The expectation is that its behaviour closely matches the legacy driver; however, during, and after the transition to the new driver being the default its behaviour may start to diverge from the legacy driver as par for the course of its evolution and gaining new features, etc. Today, broadly-speaking, sets of `swift-frontend` invocations generated by the two drivers are expected to be very similar.

Building the compiler without the standalone driver

One can build the compiler that does not rely on the standalone driver and instead uses the legacy, built-in driver using the `build-script` option: `--skip-early-swift-driver`.

Invoking the compiler without forwarding to the standalone driver

The Swift compiler can currently be invoked in an execution mode that will use the legacy C++-based compiler driver using one of the following two options:

- Passing `-disallow-use-new-driver` argument to the `swiftc` invocation
- Setting the `SWIFT_USE_OLD_DRIVER` environment variable

Reproducing the Compiler Driver build steps

A "dry-run" invocation of the `build-script` can be used to examine the SwiftDriver build stage and commands, without executing it. For example:

```
$ utils/build-script --release-debuginfo --dry-run
+ mkdir -p /SwiftWorkspace/build/Ninja-RelWithDebInfoAssert
--- Building earlyswiftdriver ---
+ /SwiftWorkspace/swift-driver/Utilities/build-script-helper.py build --package-path /SwiftWorkspace/swift-driver --build-path
/SwiftWorkspace/build/Ninja-RelWithDebInfoAssert/earlyswiftdriver-macosx-x86_64 --configuration release --toolchain
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr --ninja-bin
/Applications/Xcode.app/Contents/Developer/usr/local/bin/ninja --cmake-bin
/Applications/Xcode.app/Contents/Developer/usr/local/bin/cmake --local_compiler_build
Building the standard library for: swift-test-stdlib-macosx-x86_64
...
```

One of the first steps is an invocation of the driver's `build-script-helper.py` script which specifies that the driver us to be built (`build`) using the host toolchain (`--toolchain`) to a specified location (`--build-path`).

Installing the Compiler Driver

In order to create a Swift compiler installation (`--install-swift`), the standalone driver must be built as a separate build product using the *just-built* Swift compiler and toolchain (the ones built in the same `build-script` invocation, preceding the SwiftDriver build product). The additional build product is added to the build by specifying the `--swift-driver` option of the `build-script` . The driver product is installed into the resulting toolchain installation by specifying the `--install-swift-driver` option of the `build-script` .

Note, a "dry-run" `build-script` invocation when installing the standalone driver will demonstrate the commands required to build and install the driver as a standalone build product:

```
$ utils/build-script --release-debuginfo --dry-run --swift-driver --install-swift-driver
...
--- Cleaning swiftdriver ---
+ /SwiftWorkspace/swift-driver/Utilities/build-script-helper.py clean --package-path /SwiftWorkspace/swift-driver --build-path
/SwiftWorkspace/build/Ninja-RelWithDebInfoAssert/swiftdriver-macosx-x86_64 --configuration release --toolchain
/SwiftWorkspace/build/Ninja-RelWithDebInfoAssert/toolchain-macosx-
x86_64/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr --ninja-bin
/Applications/Xcode.app/Contents/Developer/usr/local/bin/ninja --cmake-bin
/Applications/Xcode.app/Contents/Developer/usr/local/bin/cmake
--- Building swiftdriver ---
+ /SwiftWorkspace/swift-driver/Utilities/build-script-helper.py build --package-path /SwiftWorkspace/swift-driver --build-path
/SwiftWorkspace/build/Ninja-RelWithDebInfoAssert/swiftdriver-macosx-x86_64 --configuration release --toolchain
/SwiftWorkspace/build/Ninja-RelWithDebInfoAssert/toolchain-macosx-
x86_64/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr --ninja-bin
/Applications/Xcode.app/Contents/Developer/usr/local/bin/ninja --cmake-bin
/Applications/Xcode.app/Contents/Developer/usr/local/bin/cmake
--- Installing swiftdriver ---
+ /SwiftWorkspace/swift-driver/Utilities/build-script-helper.py install --package-path /SwiftWorkspace/swift-driver --build-path
/SwiftWorkspace/build/Ninja-RelWithDebInfoAssert/swiftdriver-macosx-x
86_64 --configuration release --toolchain /SwiftWorkspace/build/Ninja-RelWithDebInfoAssert/toolchain-macosx-
x86_64/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr --ninja-bin
/Applications/Xcode.app/Contents/Developer/usr/local/bin/ninja --cmake-bin
/Applications/Xcode.app/Contents/Developer/usr/local/bin/cmake
```

These invocations of the driver's `build-script-helper.py` script specify the individual build actions (`clean` , `build` , `install`), the product build path (`--build-path`), and the *just-built* toolchain which should be used (`--toolchain`).

Debugging Swift Executables

One can use the previous tips for debugging the Swift compiler with Swift executables as well. Here are some additional useful techniques that one can use in Swift executables.

Determining the mangled name of a function in LLDB

One problem that often comes up when debugging Swift code in LLDB is that LLDB shows the demangled name instead of the mangled name. This can lead to mistakes where due to the length of the mangled names one will look at the wrong function. Using the following command, one can find the mangled name of the function in the current frame:

```
(lldb) image lookup -va $pc
Address: CollectionType3[0x0000000100004db0] (CollectionType3.__TEXT.__text + 16000)
Summary: CollectionType3`ext.CollectionType3.CollectionType3.MutableCollectionType2<A where A:
CollectionType3.MutableCollectionType2>.(subscript.materializeForSet : (Swift.Range<A.Index>) -> Swift.MutableSlice<A>).(closure
#1)
Module: file = "/Volumes/Files/work/solon/build/build-swift/validation-test-macosx-
x86_64/stdlib/Output/CollectionType.swift.gyb.tmp/CollectionType3", arch = "x86_64"
Symbol: id = {0x0000008c}, range = [0x0000000100004db0-0x0000000100056f0),
```

```
name="ext.CollectionType3.CollectionType3.MutableCollectionType2<A where A: CollectionType3.MutableCollectionType2>.
(subscript.materializeForSet : (Swift.Range<A.Index>) -> Swift.MutableSlice<A>).(closure #1)",
mangled="_TFFeRq_15CollectionType322MutableCollectionType2_S_S0_m9subscriptFGVs5Rangeqq_s16MutableIndexable5Index_GVs12MutableSlice
```

Manually symbolication using LLDB

One can perform manual symbolication of a crash log or an executable using LLDB without running the actual executable. For a detailed guide on how to do this, see: https://lldb.llvm.org/symbolication.html.

Viewing allocation history, references, and page-level info

The `malloc_history` tool (macOS only) shows the history of `malloc` and `free` calls for a particular pointer. To enable malloc_history, you must run the target process with the environment variable MallocStackLogging=1. Then you can see the allocation history of any pointer:

```
malloc_history YourProcessName 0x12345678
```

By default, this will show a compact call stack representation for each event that puts everything on a single line. For a more readable but larger representation, pass -callTree.

This works even when you have the process paused in the debugger!

The `leaks` tool (macOS only) can do more than just find leaks. You can use its pointer tracing engine to show you where a particular block is referenced:

```
leaks YourProcessName --trace=0x12345678
```

Like malloc_history, this works even when you're in the middle of debugging the process.

Sometimes you just want to know some basic info about the region of memory an address is in. The `memory region` lldb command will print out basic info about the region containing a pointer, such as its permissions and whether it's stack, heap, or a loaded image.

lldb comes with a heap script that offers powerful tools to search for pointers:

```
(lldb) p (id)[NSApplication sharedApplication]
(id) $0 = 0x00007fc50f904ba0
(lldb) script import lldb.macosx.heap
"crashlog" and "save_crashlog" command installed, use the "--help" option for detailed help
"malloc_info", "ptr_refs", "cstr_refs", "find_variable", and "objc_refs" commands have been installed, use the "--help" options on
these commands for detailed help.
(lldb) ptr_refs 0x00007fc50f904ba0
0x0000600003a49580: malloc(    48) -> 0x600003a49560 + 32
0x0000600003a6cfe0: malloc(    48) -> 0x600003a6cfc0 + 32
0x0000600001f80190: malloc(   112) -> 0x600001f80150 + 64     NSMenuItem55 bytes after NSMenuItem
0x0000600001f80270: malloc(   112) -> 0x600001f80230 + 64     NSMenuItem55 bytes after NSMenuItem
0x0000600001f80350: malloc(   112) -> 0x600001f80310 + 64     NSMenuItem55 bytes after NSMenuItem
...
```

Printing memory contents

lldb's `x` command is cryptic but extremely useful for printing out memory contents. Example:

```
(lldb) x/5a `(Class)objc_getClass("NSString")`
0x7fff83f6d660: 0x00007fff83f709f0 (void *)0x00007fff8c6550f0: NSObject
0x7fff83f6d668: 0x00007fff8c655118 (void *)0x00007fff8c6550f0: NSObject
0x7fff83f6d670: 0x000060000089c500 -> 0x00007fff2d49c550 "_getCString:maxLength:encoding:"
0x7fff83f6d678: 0x000580100000000f
0x7fff83f6d680: 0x000060000348e784
```

Let's unpack the command a bit. The `5` says that we want to print five entries. `a` means to print them as addresses, which gives you some automatic symbol lookups and pointer chasing as we see here. Finally, we give it the address. The backticks around the expression tells it to evaluate that expression and use the result as the address. Another example:

```
(lldb) x/10xb 0x000060000089c500
0x60000089c500: 0x50 0xc5 0x49 0x2d 0xff 0x7f 0x00 0x00
0x60000089c508: 0x77 0x63
```

Here, `x` means to print the values as hex, and `b` means to print byte by byte. The following specifiers are available:

- o - octal
- x - hexadecimal
- d - decimal
- u - unsigned decimal
- t - binary
- f - floating point
- a - address
- c - char
- s - string
- i - instruction
- b - byte

- h - halfword (16-bit value)
- w - word (32-bit value)
- g - giant word (64-bit value)

Debugging LLDB failures

Sometimes one needs to be able to while debugging actually debug LLDB and its interaction with Swift itself. Some examples of problems where this can come up are:

1. Compiler bugs when LLDB attempts to evaluate an expression. (expression debugging)
2. Swift variables being shown with no types. (type debugging)

To gain further insight into these sorts of failures, we use LLDB log categories. LLDB log categories provide introspection by causing LLDB to dump verbose information relevant to the category into the log as it works. The two log channels that are useful for debugging Swift issues are the "types" and "expression" log channels.

For more details about any of the information below, please run:

```
(lldb) help log enable
```

"Types" Log

The "types" log reports on LLDB's process of constructing SwiftASTContexts and errors that may occur. The two main tasks here are:

1. Constructing the SwiftASTContext for a specific single Swift module. This is used to implement frame local variable dumping via the lldb `frame variable` command, as well as the Xcode locals view. On failure, local variables will not have types.
2. Building a SwiftASTContext in which to run Swift expressions using the "expression" command. Upon failure, one will see an error like: "Shared Swift state for has developed fatal errors and is being discarded."

These errors can be debugged by turning on the types log:

```
(lldb) log enable -f /tmp/lldb-types-log.txt lldb types
```

That will write the types log to the file passed to the -f option.

NOTE Module loading can happen as a side-effect of other operations in lldb (e.g. the "file" command). To be sure that one has enabled logging before /any/ module loading has occurred, place the command into either:

```
~/.lldbinit
$PWD/.lldbinit
```

This will ensure that the type import command is run before /any/ modules are imported.

"Expression" Log

The "expression" log reports on the process of wrapping, parsing, SILGen'ing, JITing, and inserting an expression into the current Swift module. Since this can only be triggered by the user manually evaluating expression, this can be turned on at any point before evaluating an expression. To enable expression logging, first run:

```
(lldb) log enable -f /tmp/lldb-expr-log.txt lldb expression
```

and then evaluate the expression. The expression log dumps, in order, the following non-exhaustive list of state:

1. The unparsed, textual expression passed to the compiler.
2. The parsed expression.
3. The initial SILGen.
4. SILGen after SILLinking has occurred.
5. SILGen after SILLinking and Guaranteed Optimizations have occurred.
6. The resulting LLVM IR.
7. The assembly code that will be used by the JIT.

NOTE LLDB runs a handful of preparatory expressions that it uses to set up for running Swift expressions. These can make the expression logs hard to read especially if one evaluates multiple expressions with the logging enabled. In such a situation, run all expressions before the bad expression, turn on the logging, and only then run the bad expression.

Multiple Logs at a Time

Note, you can also turn on more than one log at a time as well, e.x.:

```
(lldb) log enable -f /tmp/lldb-types-log.txt lldb types expression
```

Compiler Tools/Options for Bug Hunting

Using `clang-tidy` to run the Static Analyzer

Recent versions of LLVM package the tool `clang-tidy`. This can be used in combination with a json compilation database to run static analyzer checks as well as cleanups/modernizations on a code-base. Swift's cmake invocation by default creates one of these json databases at the root path of the swift host build, for example on macOS:

```
$PATH_TO_BUILD/swift-macosx-$(uname -m)/compile_commands.json
```

Using this file, one invokes `clang-tidy` on a specific file in the codebase as follows:

```
clang-tidy -p=$PATH_TO_BUILD/swift-macosx-$(uname -m)/compile_commands.json $FULL_PATH_TO_FILE
```

One can also use shell regex to visit multiple files in the same directory. Example:

```
clang-tidy -p=$PATH_TO_BUILD/swift-macosx-$(uname -m)/compile_commands.json $FULL_PATH_TO_DIR/*.cpp
```