Design of the Swift optimizer

This document describes the design of the Swift Optimizer. It is intended for developers who wish to debug, improve or simply understand what the Swift optimizer does. Basic familiarity with the Swift programming language and knowledge of compiler optimizations is required.

Optimization pipeline overview

The Swift compiler translates textual Swift programs into LLVM-IR and uses multiple representations in between. The Swift frontend is responsible for translating textual Swift programs into well-formed and type-checked programs that are encoded in the SIL intermediate representation. The frontend emits SIL in a phase that's called SILGen (stands for SIL-generation). Next, the Swift compiler performs a sequence of transformations, such as inlining and constant propagation that allow the Swift compiler to emit diagnostic messages (such as warnings for uninitialized variables or arithmetic overflow). Next, the Swift optimizer transforms the code to make it faster. The optimizer removes redundant reference counting operations, devirtualizes function calls, and specializes generics and closures. This phase of the compiler is the focus of this document. Finally, the SIL intermediate representation is passed on to the IRGen phase (stands for intermediate representation generation phase) that lowers SIL into LLVM IR. The LLVM backend optimizes and emits binary code for the compiled program.

Please refer to the document "Swift Intermediate Language (SIL)" for more details about the SIL IR.

The compiler optimizer is responsible for optimizing the program using the high-level information that's available at the SIL level. When SIL is lowered to LLVM-IR the high-level information of the program is lost. It is important to understand that after IRGen, the LLVM optimizer can't perform high-level optimizations such as inlining of closures or CSE-ing of virtual method lookups. When a SIL virtual call or some code that deals with generics is lowered to LLVM-IR, to the optimizer this code looks like a bunch of loads, stores and opaque calls, and the LLVM-optimizer can't do anything about it.

The goal of the Swift optimizer is not to reimplement optimizations that already exist in the LLVM optimizer. The goal is to implement optimizations that can't be implemented in LLVM-IR because the high-level semantic information is lost. The Swift optimizer does have some optimizations that are similar to LLVM optimizations, like SSA-construction and function inlining. These optimizations are implemented at the SIL-level because they are required for exposing other higher-level optimizations. For example, the ARC optimizer and devirtualizer need SSA representation to analyze the program, and dead-code-elimination is a prerequisite to the array optimizations.

The Swift Pass Manager

The Swift pass manager is the unit that executes optimization passes on the functions in the Swift module. Unlike the LLVM optimizer, the Swift pass manager does not schedule analysis or optimization passes. The pass manager simply runs optimization passes on the functions in the module. The order of the optimizations is statically defined in the file "Passes.cpp".

TODO: The design that's described in the paragraph below is still under development.

The pass manager scans the module and creates a list of functions that are organized at a bottom-up order. This means that the optimizer first optimizes callees, and later optimizes the callers. This means that when optimizing some caller function, the optimizer had already optimized all of its callees and the optimizer can inspect the callee for side effects and inline it into the caller.

The pass manager is also responsible for the registry and invalidation of analysis. We discuss this topic at length below. The pass manager provides debug and logging utilities, such as the ability to print the content of the module after specific optimizations and to measure how much time is spent in each pass.

Optimization passes

There are two kind of optimization passes in Swift: Function passes, and Module passes. Function passes can inspect the entire module but can only modify a single function. Function passes can't control the order in which functions in the module are being processed - this is the job of the Pass Manager. Most optimizations are function passes. Optimizations such as LICM and CSE are function passes because they only modify a single function. Function passes are free to analyze other functions in the program.

Module passes can scan, view and transform the entire module. Optimizations that create new functions or that require a specific processing order needs to be module optimizations. The Generic Specializer pass is a module pass because it needs to perform a top-down scan of the module in order to propagate type information down the call graph.

This is the structure of a simple function pass:

```
class CSE : public SILFunctionTransform {
  void run() override {
    // .. do stuff ..
  }

  StringRef getName() override {
    return "CSE";
  }
};
```

Analysis Invalidation

Swift Analysis are very different from LLVM analysis. Swift analysis are simply a cache behind some utility that performs computation. For example, the dominators analysis is a cache that saves the result of the utility that computes the dominator tree of function.

Optimization passes can ask the pass manager for a specific analysis by name. The pass manager will return a pointer to the analysis, and optimization passes can query the analysis.

The code below requests access to the Dominance analysis.

```
  DominanceAnalysis* DA = getAnalysis<DominanceAnalysis>();
```

Passes that transform the IR are required to invalidate the analysis. However, optimization passes are not required to know about all the existing analysis. The mechanism that swift uses to invalidate analysis is broadcast-invalidation. Passes ask the pass manager to invalidate specific traits. For example, a pass like simplify-cfg will ask the pass manager to announce that it modified some branches in the code. The pass manager will send a message to all of the available analysis that says "please invalidate yourself if you care about branches for function F". The dominator tree would then invalidate the dominator tree for function F because it knows that changes to branches can mean that the dominator tree was modified.

The code below invalidates sends a message to all of the analysis saying that some instructions (that are not branches or calls) were modified in the function that the current function pass is processing.

```
    if (Changed) {
      invalidateAnalysis(InvalidationKind::Instructions);
    }
```

The code below is a part of an analysis that responds to invalidation messages. The analysis checks if any calls in the program were modified and invalidates the cache for the function that was modified.

```
  virtual void invalidate(SILFunction *F,
                          InvalidationKind K) override {
    if (K & InvalidationKind::Calls) {
      Storage[F].clear();
    }
  }
```

The invalidation traits that passes can invalidate are:

1. Instructions - some instructions were added, deleted or moved.
2. Calls - some call sites were added or deleted.
3. Branches - branches in the code were added, deleted or modified.
4. Functions - Some functions were added or deleted.

Semantic Tags

The Swift optimizer has optimization passes that target specific data structures in the Swift standard library. For example, one optimization can remove the Array copy-on-write uniqueness checks and hoist them out of loops. Another optimization can remove array access bounds checks.

The Swift optimizer can detect code in the standard library if it is marked with special attributes @_semantics, that identifies the functions.

This is an example of the *@_semantics* attribute as used by Swift Array:

```
@public @_semantics("array.count")
func getCount() -> Int {
  return _buffer.count
 }
```

Notice that as soon as we inline functions that have the @_semantics attribute the attribute is lost and the optimizer can't analyze the content of the function. For example, the optimizer can identify the array 'count' method (that returns the size of the array) and can hoist this method out of loops. However, as soon as this method is inlined, the code looks to the optimizer like a memory read from an undetermined memory location, and the optimizer can't optimize the code anymore. In order to work around this problem we prevent the inlining of functions with the @_semantics attribute until after all of the data-structure specific optimizations are done. Unfortunately, this lengthens our optimization pipeline.

Please refer to the document "High-Level SIL Optimizations" for more details.

Instruction Invalidation in SIL

Swift Passes and Analysis often keep instruction pointers in internal data structures such as Map or Set. A good example of such data structure is a list of visited instructions, or a map between a SILValue and the memory aliasing effects of the value.

Passes and utilities delete instructions in many different places in the optimizer. When instructions are deleted pointers that are saved inside maps in the different data structures become invalid. These pointers point to deallocated memory locations. In some cases malloc allocates new instructions with the same address as the freed instruction. Instruction reallocation bugs are often difficult to detect because hash maps return values that are logically incorrect.

LLVM handles this problem by keeping ValueHandles, which are a kind of smart pointers that handle instruction deletion and replaceAllUsesWith events. ValueHandles are special uses of LLVM values. One problem with this approach is that value handles require additional memory per-value and require doing extra work when working with values.

The Swift optimizer approach is to let the Context (currently a part of the SILModule) handle the invalidation of instructions. When instructions are deleted the context notifies a list of listeners. The invalidation mechanism is relatively efficient and incurs a cost only when instructions are deleted. Every time an instruction is deleted the context notifies all of the listeners that requested to be notified. A typical notifications list is very short and is made of the registered Analysis and the currently executed Pass.

Passes and Analysis are registered by the PassManager to receive instruction deletion notifications. Passes and Analysis simply need to implement the following virtual method:

```
virtual void handleNotification(swift::ValueBase *Value) override {
  llvm::errs()<<"SILCombine Deleting: " << Value<<"\n";
}
```

Debugging the optimizer

TODO.

Whole Module Optimizations

TODO.

List of passes

The updated list of passes is available in the file "Passes.def".