Passes in LLVM

Passes in LLVM

Part 1: Introduction and Concepts

You may be expecting...

Sorry, no cat photos, memes, or dragons.
We do have few WAT moments though...


Ok, one dragon.
akakatta

Motivating questions

- What is a “pass” in LLVM really?
- How do I decide what kind of pass my pass should be?
- What can my pass do? What can’t it do?
- What passes does LLVM run over my code?
- Where should I run my pass within the optimizer?

What is a *pass* in LLVM?

An operation on a unit of IR.

- Could mutate the IR
- Could compute something about the IR
- It could even be a boat

What units of IR?

- Modules
- Functions
- Basic blocks?
- Instructions?

We also have a fake, a liar, and a cheat...

The SCC pass is a fake.

- No representational model for the call graph
- Instead, LLVM computes the call graph by analyzing call instructions
- The unit isn’t really just the SCC
  - Everything above (callers) or below (callees) the SCC in the SCC DAG is included
  - But only things below (callees) can be mutated (we’ll come back to this…)

The Loop pass is a liar.

Mark him well, for he will act outside his scope.

- The idea is to operate on a particular layer of a loop nest.
- Actually modifies inner loops (OK)
- And modifies outer loops (less OK)
- And modifies the outer function (Yikes!)

The Immutable pass is a cheat.

- Examples:
  - The old DataLayoutPass
  - Basic alias analysis
- The desired behavior isn’t supported by the existing pass implementation
  - So cheat by defining away the actual unit of IR so that no part of the implementation applies?
- Still, fundamentally, tied to a module of IR!

(but fakes, liars, and cheats are really useful…)

Analysis Passes

Compute some higher-order information about the unit of IR without mutating it.

- The pass can be thought of as producing a result which can then be queried.
- Example: a DominatorTree is produced by running an analysis pass over a function.

Analysis Passes

- One analysis pass may use another analysis pass's result to compute its result
- Forms a dependency graph
- Can always satisfy this (baring cycles) because the IR isn't mutated, so results remain valid

Analysis Passes

- All analysis pass results are cacheable and only invalidated when the unit of IR is mutated.

Leads to the other form passes take...

Transformation Passes

Transform a unit of IR in some way.

- LLVM operates in-place, so no “result”
- All passes *must* leave the IR in a valid state
- Can depend on analysis pass results
- Can preserve analysis pass results even while transforming IR
  - Unless explicitly preserved, the default is to invalidate analysis results conservatively

Transformation Passes

- Cannot depend on other transformation passes!
  - Forms a new, sub-set IR, which is problematic.
  - How do you resolve two such dependencies? Only one can come last…
  - Causes rampant re-running of invalidated analyses

Yet, we do this today. =/ It's a pretty big wart.

How do passes combine?

Not functions, so no functional composition...

Two dimensions of pass aggregation

- Running multiple passes over a unit of IR
  - A function pass which runs several other function passes
- Decomposing a unit of IR into smaller units
  - A module pass that runs a function pass over each function in the module

These are somewhat conflated in the current implementation.

PassManagerBuilder

- Provides canned aggregations of passes
  - Different use cases modeled
  - Some pluggable interfaces for extensions
  - Exposes primary coarse grained parameters used across frontends
- Used by Clang ‘-O*’ flags, opt ‘-O*’ flags, etc.

The LLVM Pass Pipelines

How is it organized, and why?

Three phases of IR optimization:

1. Cleanup and canonicalization
2. Simplification and canonicalization
3. Target-specific optimization and lowering

A not-so-brief digression...

What's the big deal with canonicalization?

- TMTOWTDI, many IR forms are equivalent
- Possible patterns of multiple operations are combinatorially many
- Recognizing all of them in analyses is infeasible
- Instead, pick a canonical form & convert to it

A not-so-brief digression...

Canonicalization is a *part* of optimization

LLVM does more than just canonicalize, it also simplifies which something different!

- Deleting dead code
- Proving equivalence of simpler forms
- etc...

Cleanup: Making frontends simpler

Primary goal: cleanup the IR from a frontend

- Form SSA - doing this in each FE is a waste
- Lower frontend-friendly intrinsics and patterns into analysis-friendly annotations
- Canonicalize expression forms and CFG

Not all messy IR is efficient for this...

Simplification: IPO

The call graph SCC pass manager is the outer framework

- Primary IPO mechanism in LLVM
- All finer grained optimization happens inside

Simplification: IPO via CGSCC

Core idea: pair-wise IPO across call edges

- Process SCCs in bottom(callee)-up order
- For every call site, callee is in-SCC or fully simplified and canonicalized
  - This maximizes the ability to analyze the callee

Simplification: IPO via CGSCC

Transformations here include:

- Argument promotion (deletion?)
- Call and argument attribute synthesis
- Inlining
- Core per-function simplification
- (Outlining?)
- (IP constant propagation?)

Simplification: Per-function

Primarily run within the SCC over each function

- Fairly standard scalar optimization pipeline
  - Decomposes aggregates into scalar SSA values
  - Reassociates math into canonical expression trees
  - Propagates equivalent values through memory
  - Puts loops into canonical loop form
  - Runs loop optimizations iteratively on each loop nest inside out

Simplification: Per-loop

Why inside-out per loop-nest?

- Phase ordering problems plague loop optimizations:
  - Re-arranging loop makes its result computable
  - That in turn allows deleting the loop
  - That in turn makes the enclosing loop’s result computable

Simplification: Per-loop

Specific loop optimizations:

- Rotate the loop structure
- Hoist invariant code
- Unswitch loop
- Canonicalize the induction variable’s form
- Unroll loops with small, constant trip counts
- (Interchange, fusion, fission, peeling, …)

Lowering: it’s still IR!

Lowering and target specific optimizations still produce perfectly valid LLVM IR!

- About forming patterns, not representations
- Still reuses IR analyses and utilities
- Can make things non-canonical at the end

Lowering: Loop tuning

Tunes loop structure for fast execution rather than easy analysis

- Widens loop to use SIMD vector unit where profitable and safe
- Widens loop to leverage ILP
- Unrolls loop to improve decoder loop detection and minimize branch penalties
- Loop strength reduction to match addressing modes

Lowering: Target preparation

- Form target-specific patterns that are non-canonical
- Ease the burden of DAG building (and instruction selection) using intrinsic placeholders

“CodeGenPrep” today.

Done!
We have optimized IR!

Summary:

- Passes operate over units of IR
  - Real units and fakes, using lies and cheats
- Analysis passes are pure functions on the IR computing some higher-order information
  - Part of dependency graph
  - Results can be invalidated
- Transformation passes mutate their IR unit
  - Can depend on analysis passes, not transformation
  - Invalidates analysis results for that IR unit

Summary:

- PassManagerBuilder is used to produce an LLVM pass pipeline
- Three phases: cleanup, simplification, and lowering.
- Canonicalization is an important part of both cleanup and simplification
- Lowering loses canonicalization, but forms IR that is better suited to specific targets

Questions?

(And many thanks! Look forward to part 2!)