LLVM’s Analysis and Transform Passes

- **-argpromotion**: Promote 'by reference' arguments to scalars
- **-bb-vectorize**: Basic-Block Vectorization
- **-block-placement**: Profile Guided Basic Block Placement
- **-break-crit-edges**: Break critical edges in CFG
- **-codegenprepare**: Optimize for code generation
- **-constmerge**: Merge Duplicate Global Constants
- **-dce**: Dead Code Elimination
- **-deadargelim**: Dead Argument Elimination
- **-deadtypeelim**: Dead Type Elimination
- **-die**: Dead Instruction Elimination
- **-dse**: Dead Store Elimination
- **-function-attrs**: Deduce function attributes
- **-globaldce**: Dead Global Elimination
- **-globalopt**: Global Variable Optimizer
- **-gvn**: Global Value Numbering
- **-indvars**: Canonicalize Induction Variables
- **-inline**: Function Integration/Inlining
- **-instcombine**: Combine redundant instructions
- **-aggressive-instcombine**: Combine expression patterns
- **-internalize**: Internalize Global Symbols
- **-ipsccp**: Interprocedural Sparse Conditional Constant Propagation
- **-jump-threading**: Jump Threading
- **-lcssa**: Loop-Closed SSA Form Pass
- **-licm**: Loop Invariant Code Motion
- **-loop-deletion**: Delete dead loops
- **-loop-extract**: Extract loops into new functions
- **-loop-extract-single**: Extract at most one loop into a new function
- **-loop-reduce**: Loop Strength Reduction
- **-loop-rotate**: Rotate Loops
- **-loop-simplify**: Canonicalize natural loops
- **-loop-unroll**: Unroll loops
- **-loop-unroll-and-jam**: Unroll and Jam loops
- **-loop-unswitch**: Unswitch loops
- **-loweratomic**: Lower atomic intrinsics to non-atomic form
- **-lowerinvoke**: Lower invokes to calls, for unwindless code generators
- **-lowerswitch**: Lower **SwitchInst**s to branches
- **-mem2reg**: Promote Memory to Register
- **-memcpyopt**: MemCpy Optimization
- **-mergefunc**: Merge Functions
- **-mergereturn**: Unify function exit nodes
- **-partial-inliner**: Partial Inliner
- **-prune-eh**: Remove unused exception handling info
- **-reassociate**: Reassociate expressions
- **-reg2mem**: Demote all values to stack slots
- **-sroa**: Scalar Replacement of Aggregates
- **-sccp**: Sparse Conditional Constant Propagation
- **-simplifycfg**: Simplify the CFG
- **-sink**: Code sinking
- **-strip**: Strip all symbols from a module
- **-strip-dead-debug-info**: Strip debug info for unused symbols
- **-strip-dead-prototypes**: Strip Unused Function Prototypes
- **-strip-debug-declare**: Strip all **llvm.dbg.declare** intrinsics
- **-strip-nondebug**: Strip all symbols, except dbg symbols, from a module
- **-tailcallelim**: Tail Call Elimination

- Utility Passes
  - **-deadarghaX0r**: Dead Argument Hacking (BUGPOINT USE ONLY; DO NOT USE)
  - **-extract-blocks**: Extract Basic Blocks From Module (for bugpoint use)

Introduction

This document serves as a high level summary of the optimization features that LLVM provides. Optimizations are implemented as Passes that traverse some portion of a program to either collect information or transform the program. The table below divides the passes that LLVM provides into three categories. Analysis passes compute information that other passes can use or for debugging or program visualization purposes. Transform passes can use (or invalidate) the analysis passes. Transform passes all mutate the program in some way. Utility passes provides some utility but don't otherwise fit categorization. For example passes to extract functions to bitcode or write a module to bitcode are neither analysis nor transform passes. The table of contents above provides a quick summary of each pass and links to the more complete pass description later in the document.

Analysis Passes

This section describes the LLVM Analysis Passes.

-aa-eval: Exhaustive Alias Analysis Precision Evaluator

This is a simple N^2 alias analysis accuracy evaluator. Basically, for each function in the program, it simply queries to see how the alias analysis implementation answers alias queries between each pair of pointers in the function.

This is inspired and adapted from code by: Naveen Neelakantam, Francesco Spadini, and Wojciech Stryjewski.

-basic-aa: Basic Alias Analysis (stateless AA impl)

A basic alias analysis pass that implements identities (two different globals cannot alias, etc), but does no stateful analysis.

-basiccg: Basic CallGraph Construction

Yet to be written.

-count-aa: Count Alias Analysis Query Responses

A pass which can be used to count how many alias queries are being made and how the alias analysis implementation being used responds.

-da: Dependence Analysis

Dependence analysis framework, which is used to detect dependences in memory accesses.

-debug-aa: AA use debugger

This simple pass checks alias analysis users to ensure that if they create a new value, they do not query AA without informing it of the value. It acts as a shim over any other AA pass you want.

Yes keeping track of every value in the program is expensive, but this is a debugging pass.

-domfrontier: Dominance Frontier Construction

This pass is a simple dominator construction algorithm for finding forward dominator frontiers.

-domtree: Dominator Tree Construction

This pass is a simple dominator construction algorithm for finding forward dominators.

-dot-callgraph: Print Call Graph to "dot" file

This pass, only available in `opt`, prints the call graph into a `.dot` graph. This graph can then be processed with the "dot" tool to convert it to postscript or some other suitable format.

-dot-cfg: Print CFG of function to "dot" file

This pass, only available in `opt`, prints the control flow graph into a `.dot` graph. This graph can then be processed with the **dot** tool to convert it to postscript or some other suitable format.

-dot-cfg-only: Print CFG of function to "dot" file (with no function bodies)

This pass, only available in `opt`, prints the control flow graph into a `.dot` graph, omitting the function bodies. This graph can then be processed with the **dot** tool to convert it to postscript or some other suitable format.

-dot-dom: Print dominance tree of function to "dot" file

This pass, only available in `opt`, prints the dominator tree into a `.dot` graph. This graph can then be processed with the **dot** tool to convert it to postscript or some other suitable format.

-dot-dom-only: Print dominance tree of function to "dot" file (with no function bodies)

This pass, only available in `opt`, prints the dominator tree into a `.dot` graph, omitting the function bodies. This graph can then be processed with the **dot** tool to convert it to postscript or some other suitable format.

-dot-postdom: Print postdominance tree of function to "dot" file

This pass, only available in `opt`, prints the post dominator tree into a `.dot` graph. This graph can then be processed with the **dot** tool to convert it to postscript or some other suitable format.

-dot-postdom-only: Print postdominance tree of function to "dot" file (with no function bodies)

This pass, only available in `opt`, prints the post dominator tree into a `.dot` graph, omitting the function bodies. This graph can then be processed with the **dot** tool to convert it to postscript or some other suitable format.

-globalsmodref-aa: Simple mod/ref analysis for globals

This simple pass provides alias and mod/ref information for global values that do not have their address taken, and keeps track of whether functions read or write memory (are "pure"). For this simple (but very common) case, we can provide pretty accurate and useful information.

-instcount: Counts the various types of **Instruction**s

This pass collects the count of all instructions and reports them.

`-intervals`: Interval Partition Construction

This analysis calculates and represents the interval partition of a function, or a preexisting interval partition.

In this way, the interval partition may be used to reduce a flow graph down to its degenerate single node interval partition (unless it is irreducible).

`-iv-users`: Induction Variable Users

Bookkeeping for “interesting” users of expressions computed from induction variables.

`-lazy-value-info`: Lazy Value Information Analysis

Interface for lazy computation of value constraint information.

`-libcall-aa`: LibCall Alias Analysis

LibCall Alias Analysis.

`-lint`: Statically lint-checks LLVM IR

This pass statically checks for common and easily-identified constructs which produce undefined or likely unintended behavior in LLVM IR.

It is not a guarantee of correctness, in two ways. First, it isn’t comprehensive. There are checks which could be done statically which are not yet implemented. Some of these are indicated by TODO comments, but those aren’t comprehensive either. Second, many conditions cannot be checked statically. This pass does no dynamic instrumentation, so it can’t check for all possible problems.

Another limitation is that it assumes all code will be executed. A store through a null pointer in a basic block which is never reached is harmless, but this pass will warn about it anyway.

Optimization passes may make conditions that this pass checks for more or less obvious. If an optimization pass appears to be introducing a warning, it may be that the optimization pass is merely exposing an existing condition in the code.

This code may be run before instcombine. In many cases, instcombine checks for the same kinds of things and turns instructions with undefined behavior into unreachable (or equivalent). Because of this, this pass makes some effort to look through bitcasts and so on.

`-loops`: Natural Loop Information

This analysis is used to identify natural loops and determine the loop depth of various nodes of the CFG. Note that the loops identified may actually be several natural loops that share the same header node... not just a single natural loop.

`-memdep`: Memory Dependence Analysis

An analysis that determines, for a given memory operation, what preceding memory operations it depends on. It builds on alias analysis information, and tries to provide a lazy, caching interface to a common kind of alias information query.

`-module-debuginfo`: Decodes module-level debug info

This pass decodes the debug info metadata in a module and prints in a (sufficiently-prepared-) human-readable form.

For example, run this pass from `opt` along with the `-analyze` option, and it'll print to standard output.

`-postdomfrontier`: Post-Dominance Frontier Construction

This pass is a simple post-dominator construction algorithm for finding post-dominator frontiers.

`-postdomtree`: Post-Dominator Tree Construction

This pass is a simple post-dominator construction algorithm for finding post-dominators.

`-print-alias-sets`: Alias Set Printer

Yet to be written.

`-print-callgraph`: Print a call graph

This pass, only available in `opt`, prints the call graph to standard error in a human-readable form.

`-print-callgraph-sccs`: Print SCCs of the Call Graph

This pass, only available in `opt`, prints the SCCs of the call graph to standard error in a human-readable form.

`-print-cfg-sccs`: Print SCCs of each function CFG

This pass, only available in `opt`, printsthe SCCs of each function CFG to standard error in a human-readable fom.

`-print-dom-info`: Dominator Info Printer

Dominator Info Printer.

`-print-externalfnconstants`: Print external fn callsites passed constants

This pass, only available in `opt`, prints out call sites to external functions that are called with constant arguments. This can be useful when looking for standard library functions we should constant fold or handle in alias analyses.

`-print-function`: Print function to stderr

The `PrintFunctionPass` class is designed to be pipelined with other `FunctionPasses`, and prints out the functions of the module as they are processed.

`-print-module`: Print module to stderr

This pass simply prints out the entire module when it is executed.

`-print-used-types`: Find Used Types

This pass is used to seek out all of the types in use by the program. Note that this analysis explicitly does not include types only used by the symbol table.

`-regions`: Detect single entry single exit regions

The `RegionInfo` pass detects single entry single exit regions in a function, where a region is defined as any subgraph that is connected to the remaining graph at only two spots. Furthermore, a hierarchical region tree is built.

`-scalar-evolution`: Scalar Evolution Analysis

The `ScalarEvolution` analysis can be used to analyze and catagorize scalar expressions in loops. It specializes in recognizing general induction variables, representing them with the abstract and opaque `SCEV` class. Given this analysis, trip counts of loops and other important properties can be obtained.

This analysis is primarily useful for induction variable substitution and strength reduction.

`-scev-aa`: ScalarEvolution-based Alias Analysis

Simple alias analysis implemented in terms of `ScalarEvolution` queries.

This differs from traditional loop dependence analysis in that it tests for dependencies within a single iteration of a loop, rather than dependencies between different iterations.

`ScalarEvolution` has a more complete understanding of pointer arithmetic than `BasicAliasAnalysis`' collection of ad-hoc analyses.

`-stack-safety`: Stack Safety Analysis

The `StackSafety` analysis can be used to determine if stack allocated variables can be considered safe from memory access bugs.

This analysis' primary purpose is to be used by sanitizers to avoid unnecessary instrumentation of safe variables.

`-targetdata`: Target Data Layout

Provides other passes access to information on how the size and alignment required by the target ABI for various data types.

Transform Passes

This section describes the LLVM Transform Passes.

`-adce`: Aggressive Dead Code Elimination

ADCE aggressively tries to eliminate code. This pass is similar to DCE but it assumes that values are dead until proven otherwise. This is similar to SCCP, except applied to the liveness of values.

`-always-inline`: Inliner for `always_inline` functions

A custom inliner that handles only functions that are marked as "always inline". debug ok

`-argpromotion`: Promote 'by reference' arguments to scalars

This pass promotes "by reference" arguments to be "by value" arguments. In practice, this means looking for internal functions that have pointer arguments. If it can prove, through the use of alias analysis, that an argument is *only* loaded, then it can pass the value into the function instead of the address of the value. This can cause recursive simplification of code and lead to the elimination of allocas (especially in C++ template code like the STL).

This pass also handles aggregate arguments that are passed into a function, scalarizing them if the elements of the aggregate are only loaded. Note that it refuses to scalarize aggregates which would require passing in more than three operands to the function, because passing thousands of operands for a large array or structure is unprofitable!

Note that this transformation could also be done for arguments that are only stored to (returning the value instead), but does not currently. This case would be best handled when and if LLVM starts supporting multiple return values from functions.

`-bb-vectorize`: Basic-Block Vectorization

This pass combines instructions inside basic blocks to form vector instructions. It iterates over each basic block, attempting to pair compatible instructions, repeating this process until no additional pairs are selected for vectorization. When the outputs of some pair of compatible instructions are used as inputs by some other pair of compatible instructions, those pairs are part of a potential vectorization chain. Instruction pairs are only fused into vector instructions when they are part of a chain longer than some threshold length. Moreover, the pass attempts to find the best possible chain for each pair of compatible instructions. These heuristics are intended to prevent vectorization in cases where it would not yield a performance increase of the resulting code.

`-block-placement`: Profile Guided Basic Block Placement

This pass is a very simple profile guided basic block placement algorithm. The idea is to put frequently executed blocks together at the start of the function and hopefully increase the number of fall-through conditional branches. If there is no profile information for a particular function, this pass basically orders blocks in depth-first order. reorders but requires profile

`-break-crit-edges`: Break critical edges in CFG

Break all of the critical edges in the CFG by inserting a dummy basic block. It may be "required" by passes that cannot deal with critical edges. This transformation obviously invalidates the CFG, but can update forward dominator (set, immediate dominators, tree, and frontier) information. debug ok

`-codegenprepare`: Optimize for code generation

This pass munges the code in the input function to better prepare it for SelectionDAG-based code generation. This works around limitations in its basic-block-at-a-time approach. It should eventually be removed.

`-constmerge`: Merge Duplicate Global Constants

Merges duplicate global constants together into a single constant that is shared. This is useful because some passes (i.e., TraceValues) insert a lot of string constants into the program, regardless of whether or not an existing string is available.

`-dce`: Dead Code Elimination

Dead code elimination is similar to dead instruction elimination, but it rechecks instructions that were used by removed instructions to see if they are newly dead.

tell that the instr has been elim

`-deadargelim`: Dead Argument Elimination

This pass deletes dead arguments from internal functions. Dead argument elimination removes arguments which are directly dead, as well as arguments only passed into function calls as dead arguments of other functions. This pass also deletes dead arguments in a similar way.

This pass is often useful as a cleanup pass to run after aggressive interprocedural passes, which add possibly-dead arguments.

`-deadtypeelim`: Dead Type Elimination

This pass is used to cleanup the output of GCC. It eliminate names for types that are unused in the entire translation unit, using the find used types pass.

`-die`: Dead Instruction Elimination

Dead instruction elimination performs a single pass over the function, removing instructions that

are obviously dead.

`-dse`: Dead Store Elimination

A trivial dead store elimination that only considers basic-block local redundant stores.

`-function-attrs`: Deduce function attributes

A simple interprocedural pass which walks the call-graph, looking for functions which do not access or only read non-local memory, and marking them `readnone`/`readonly`. In addition, it marks function arguments (of pointer type) "`nocapture`" if a call to the function does not create any copies of the pointer value that outlive the call. This more or less means that the pointer is only dereferenced, and not returned from the function or stored in a global. This pass is implemented as a bottom-up traversal of the call-graph.

`-globaldce`: Dead Global Elimination

This transform is designed to eliminate unreachable internal globals from the program. It uses an aggressive algorithm, searching out globals that are known to be alive. After it finds all of the globals which are needed, it deletes whatever is left over. This allows it to delete recursive chunks of the program which are unreachable.

`-globalopt`: Global Variable Optimizer

This pass transforms simple global variables that never have their address taken. If obviously true, it marks read/write globals as constant, deletes variables only stored to, etc.

`-gvn`: Global Value Numbering

This pass performs global value numbering to eliminate fully and partially redundant instructions. It also performs redundant load elimination.

what

`-indvars`: Canonicalize Induction Variables

This transformation analyzes and transforms the induction variables (and computations derived from them) into simpler forms suitable for subsequent analysis and transformation.

This transformation makes the following changes to each loop with an identifiable induction variable:

this messes up debugability

- All loops are transformed to have a *single* canonical induction variable which starts at zero and steps by one.
- The canonical induction variable is guaranteed to be the first PHI node in the loop header block.
- Any pointer arithmetic recurrences are raised to use array subscripts.

If the trip count of a loop is computable, this pass also makes the following changes:

- The exit condition for the loop is canonicalized to compare the induction value against the exit value. This turns loops like:

```
for (i = 7; i*i < 1000; ++i)
```

into

```
for (i = 0; i != 25; ++i)
```

- Any use outside of the loop of an expression derived from the indvar is changed to compute the derived value outside of the loop, eliminating the dependence on the exit value of the induction variable. If the only purpose of the loop is to compute the exit value of some

derived expression, this transformation will make the loop dead.

This transformation should be followed by strength reduction after all of the desired loop transformations have been performed. Additionally, on targets where it is profitable, the loop could be transformed to count down to zero (the "do loop" optimization).

`-inline`: Function Integration/Inlining

Bottom-up inlining of functions into callees.

`-instcombine`: Combine redundant instructions

Combine instructions to form fewer, simple instructions. This pass does not modify the CFG. This pass is where algebraic simplification happens.

this messes up debugability

This pass combines things like:

```
%Y = add i32 %X, 1
%Z = add i32 %Y, 1
```

into:

```
%Z = add i32 %X, 2
```

This is a simple worklist driven algorithm.

This pass guarantees that the following canonicalizations are performed on the program:

1. If a binary operator has a constant operand, it is moved to the right-hand side.
2. Bitwise operators with constant operands are always grouped so that shifts are performed first, then `or`s, then `and`s, then `xor`s.
3. Compare instructions are converted from <, >, ≤, or ≥ to = or ≠ if possible.
4. All `cmp` instructions on boolean values are replaced with logical operations.
5. `add X, X` is represented as `mul X, 2` ⇒ `shl X, 1`
6. Multiplies with a constant power-of-two argument are transformed into shifts.
7. … etc.

This pass can also simplify calls to specific well-known function calls (e.g. runtime library functions). For example, a call `exit(3)` that occurs within the `main()` function can be transformed into simply `return 3`. Whether or not library calls are simplified is controlled by the -function-attrs pass and LLVM's knowledge of library calls on different targets.

`-aggressive-instcombine`: Combine expression patterns

Combine expression patterns to form expressions with fewer, simple instructions. This pass does not modify the CFG.

debug ko

For example, this pass reduce width of expressions post-dominated by TruncInst into smaller width when applicable.

It differs from instcombine pass in that it contains pattern optimization that requires higher complexity than the O(1), thus, it should run fewer times than instcombine pass.

`-internalize`: Internalize Global Symbols

This pass loops over all of the functions in the input module, looking for a main function. If a main function is found, all other functions and all global variables with initializers are marked as internal.

`-ipsccp`: Interprocedural Sparse Conditional Constant Propagation

An interprocedural variant of Sparse Conditional Constant Propagation.

`-jump-threading`: Jump Threading

Jump threading tries to find distinct threads of control flow running through a basic block. This pass looks at blocks that have multiple predecessors and multiple successors. If one or more of the predecessors of the block can be proven to always cause a jump to one of the successors, we forward the edge from the predecessor to the successor by duplicating the contents of this block.

An example of when this can occur is code like this:

```
if () { ...
  X = 4;
}
if (X < 3) {
```

In this case, the unconditional branch at the end of the first if can be revectored to the false side of the second if.

`-lcssa`: Loop-Closed SSA Form Pass

This pass transforms loops by placing phi nodes at the end of the loops for all values that are live across the loop boundary. For example, it turns the left into the right code:

```
for (...)                 for (...)
    if (c)                    if (c)
        X1 = ...                  X1 = ...
    else                      else
        X2 = ...                  X2 = ...
    X3 = phi(X1, X2)          X3 = phi(X1, X2)
... = X3 + 4                  X4 = phi(X3)
                              ... = X4 + 4
```

This is still valid LLVM; the extra phi nodes are purely redundant, and will be trivially eliminated by `InstCombine`. The major benefit of this transformation is that it makes many other loop optimizations, such as `LoopUnswitch`ing, simpler. You can read more in the loop terminology section for the LCSSA form.

`-licm`: Loop Invariant Code Motion

This pass performs loop invariant code motion, attempting to remove as much code from the body of a loop as possible. It does this by either hoisting code into the preheader block, or by sinking code to the exit blocks if it is safe. This pass also promotes must-aliased memory locations in the loop to live in registers, thus hoisting and sinking "invariant" loads and stores.

This pass uses alias analysis for two purposes: debug ko

1. Moving loop invariant loads and calls out of loops. If we can determine that a load or call inside of a loop never aliases anything stored to, we can hoist it or sink it like any other instruction.
2. Scalar Promotion of Memory. If there is a store instruction inside of the loop, we try to move the store to happen AFTER the loop instead of inside of the loop. This can only happen if a few conditions are true:
    1. The pointer stored through is loop invariant.
    2. There are no stores or loads in the loop which *may* alias the pointer. There are no calls in the loop which mod/ref the pointer.

    If these conditions are true, we can promote the loads and stores in the loop of the pointer to use a temporary alloca'd variable. We then use the mem2reg functionality to construct the appropriate SSA form for the variable.

`-loop-deletion`: Delete dead loops

This file implements the Dead Loop Deletion Pass. This pass is responsible for eliminating loops with non-infinite computable trip counts that have no side effects or volatile instructions, and do not contribute to the computation of the function's return value.

`-loop-extract`: Extract loops into new functions

A pass wrapper around the `ExtractLoop()` scalar transformation to extract each top-level loop into its own new function. If the loop is the *only* loop in a given function, it is not touched. This is a pass most useful for debugging via bugpoint.

`-loop-extract-single`: Extract at most one loop into a new function

Similar to Extract loops into new functions, this pass extracts one natural loop from the program into a function if it can. This is used by **bugpoint**.

`-loop-reduce`: Loop Strength Reduction

This pass performs a strength reduction on array references inside loops that have as one or more of their components the loop induction variable. This is accomplished by creating a new value to hold the initial value of the array access for the first iteration, and then creating a new GEP instruction in the loop to increment the value by the appropriate amount.

debug ko

`-loop-rotate`: Rotate Loops

A simple loop rotation transformation. A summary of it can be found in Loop Terminology for Rotated Loops.

`-loop-simplify`: Canonicalize natural loops

This pass performs several transformations to transform natural loops into a simpler form, which makes subsequent analyses and transformations simpler and more effective. A summary of it can be found in Loop Terminology, Loop Simplify Form.

Loop pre-header insertion guarantees that there is a single, non-critical entry edge from outside of the loop to the loop header. This simplifies a number of analyses and transformations, such as LICM.

Loop exit-block insertion guarantees that all exit blocks from the loop (blocks which are outside of the loop that have predecessors inside of the loop) only have predecessors from inside of the loop (and are thus dominated by the loop header). This simplifies transformations such as store-sinking that are built into LICM.

This pass also guarantees that loops will have exactly one backedge.

debug ko

Note that the simplifycfg pass will clean up blocks which are split out but end up being unnecessary, so usage of this pass should not pessimize generated code.

This pass obviously modifies the CFG, but updates loop information and dominator information.

`-loop-unroll`: Unroll loops

This pass implements a simple loop unroller. It works best when loops have been canonicalized by the indvars pass, allowing it to determine the trip counts of loops easily.

debug ko

`-loop-unroll-and-jam`: Unroll and Jam loops

This pass implements a simple unroll and jam classical loop optimisation pass. It transforms loop from:

```
for i.. i+= 1                   for i.. i+= 4
```

```
for j..                          for j..
  code(i, j)                       code(i, j)
                                   code(i+1, j)
                                   code(i+2, j)
                                   code(i+3, j)
                                 remainder loop
```

debug ko

Which can be seen as unrolling the outer loop and "jamming" (fusing) the inner loops into one. When variables or loads can be shared in the new inner loop, this can lead to significant performance improvements. It uses Dependence Analysis for proving the transformations are safe.

`-loop-unswitch`: Unswitch loops

This pass transforms loops that contain branches on loop-invariant conditions to have multiple loops. For example, it turns the left into the right code:

```
for (...)                        if (lic)
    A                                for (...)
    if (lic)                             A; B; C
        B                        else
    C                                for (...)
                                         A; C
```

debug ko

This can increase the size of the code exponentially (doubling it every time a loop is unswitched) so we only unswitch if the resultant code will be smaller than a threshold.

This pass expects LICM to be run before it to hoist invariant conditions out of the loop, to make the unswitching opportunity obvious.

`-loweratomic`: Lower atomic intrinsics to non-atomic form

This pass lowers atomic intrinsics to non-atomic form for use in a known non-preemptible environment.

The pass does not verify that the environment is non-preemptible (in general this would require knowledge of the entire call graph of the program including any libraries which may not be available in bitcode form); it simply lowers every atomic intrinsic.

`-lowerinvoke`: Lower invokes to calls, for unwindless code generators

This transformation is designed for use by code generators which do not yet support stack unwinding. This pass converts `invoke` instructions to `call` instructions, so that any exception-handling `landingpad` blocks become dead code (which can be removed by running the `-simplifycfg` pass afterwards).

`-lowerswitch`: Lower `SwitchInst`s to branches

Rewrites switch instructions with a sequence of branches, which allows targets to get away with not implementing the switch instruction until it is convenient.

probably debug ok

`-mem2reg`: Promote Memory to Register

This file promotes memory references to be register references. It promotes alloca instructions which only have loads and stores as uses. An `alloca` is transformed by using dominator frontiers to place phi nodes, then traversing the function in depth-first order to rewrite loads and stores as appropriate. This is just the standard SSA construction algorithm to construct "pruned" SSA form.

`-memcpyopt`: MemCpy Optimization

This pass performs various transformations related to eliminating `memcpy` calls, or transforming sets of stores into `memset`s.

`-mergefunc`: Merge Functions

This pass looks for equivalent functions that are mergable and folds them.

Total-ordering is introduced among the functions set: we define comparison that answers for every two functions which of them is greater. It allows to arrange functions into the binary tree.

For every new function we check for equivalent in tree.

If equivalent exists we fold such functions. If both functions are overridable, we move the functionality into a new internal function and leave two overridable thunks to it.

If there is no equivalent, then we add this function to tree.

Lookup routine has O(log(n)) complexity, while whole merging process has complexity of O(n*log(n)).

Read this article for more details.

probably debug ko

`-mergereturn`: Unify function exit nodes

Ensure that functions have at most one `ret` instruction in them. Additionally, it keeps track of which node is the new exit node of the CFG.

debug ko

`-partial-inliner`: Partial Inliner

This pass performs partial inlining, typically by inlining an `if` statement that surrounds the body of the function.

`-prune-eh`: Remove unused exception handling info

This file implements a simple interprocedural pass which walks the call-graph, turning invoke instructions into call instructions if and only if the callee cannot throw an exception. It implements this as a bottom-up traversal of the call-graph.

`-reassociate`: Reassociate expressions

This pass reassociates commutative expressions in an order that is designed to promote better constant propagation, GCSE, LICM, PRE, etc.

For example: 4 + (x + 5) ⇒ x + (4 + 5)

In the implementation of this algorithm, constants are assigned rank = 0, function arguments are rank = 1, and other values are assigned ranks corresponding to the reverse post order traversal of current function (starting at 2), which effectively gives values in deep loops higher rank than values not in loops.

`-reg2mem`: Demote all values to stack slots

This file demotes all registers to memory references. It is intended to be the inverse of mem2reg. By converting to `load` instructions, the only values live across basic blocks are `alloca` instructions and `load` instructions before `phi` nodes. It is intended that this should make CFG hacking much easier. To make later hacking easier, the entry block is split into two, such that all introduced `alloca` instructions (and nothing else) are in the entry block.

`-sroa`: Scalar Replacement of Aggregates

The well-known scalar replacement of aggregates transformation. This transform breaks up `alloca` instructions of aggregate type (structure or array) into individual `alloca` instructions for each member if possible. Then, if possible, it transforms the individual `alloca` instructions into nice clean scalar SSA form.

debug ??

`-sccp`: Sparse Conditional Constant Propagation

Sparse conditional constant propagation and merging, which can be summarized as:

- Assumes values are constant unless proven otherwise
- Assumes BasicBlocks are dead unless proven otherwise
- Proves values to be constant, and replaces them with constants
- Proves conditional branches to be unconditional

Note that this pass has a habit of making definitions be dead. It is a good idea to run a DCE pass sometime after running this pass.

`-simplifycfg`: Simplify the CFG

Performs dead code elimination and basic block merging. Specifically:

- Removes basic blocks with no predecessors.
- Merges a basic block into its predecessor if there is only one and the predecessor only has one successor.
- Eliminates PHI nodes for basic blocks with a single predecessor.
- Eliminates a basic block that only contains an unconditional branch.

debug meh

`-sink`: Code sinking

This pass moves instructions into successor blocks, when possible, so that they aren't executed on paths where their results aren't needed.

debug ko

`-strip`: Strip all symbols from a module

Performs code stripping. This transformation can delete:

- names for virtual registers
- symbols for internal globals and functions
- debug information

Note that this transformation makes code much less readable, so it should only be used in situations where the strip utility would be used, such as reducing code size or making it harder to reverse engineer code.

`-strip-dead-debug-info`: Strip debug info for unused symbols

performs code stripping. this transformation can delete:

- names for virtual registers
- symbols for internal globals and functions
- debug information

note that this transformation makes code much less readable, so it should only be used in situations where the strip utility would be used, such as reducing code size or making it harder to reverse engineer code.

`-strip-dead-prototypes`: Strip Unused Function Prototypes

This pass loops over all of the functions in the input module, looking for dead declarations and removes them. Dead declarations are declarations of functions for which no implementation is available (i.e., declarations for unused library functions).

`-strip-debug-declare`: Strip all `llvm.dbg.declare` intrinsics

This pass implements code stripping. Specifically, it can delete:

1. names for virtual registers
2. symbols for internal globals and functions
3. debug information

Note that this transformation makes code much less readable, so it should only be used in situations where the ‘strip’ utility would be used, such as reducing code size or making it harder to reverse engineer code.

-strip-nondebug: Strip all symbols, except dbg symbols, from a module

This pass implements code stripping. Specifically, it can delete:

1. names for virtual registers
2. symbols for internal globals and functions
3. debug information

Note that this transformation makes code much less readable, so it should only be used in situations where the ‘strip’ utility would be used, such as reducing code size or making it harder to reverse engineer code.

-tailcallelim: Tail Call Elimination

This file transforms calls of the current function (self recursion) followed by a return instruction with a branch to the entry of the function, creating a loop. This pass also implements the following extensions to the basic algorithm: debug ko ??

1. Trivial instructions between the call and return do not prevent the transformation from taking place, though currently the analysis cannot support moving any really useful instructions (only dead ones).
2. This pass transforms functions that are prevented from being tail recursive by an associative expression to use an accumulator variable, thus compiling the typical naive factorial or fib implementation into efficient code.
3. TRE is performed if the function returns void, if the return returns the result returned by the call, or if the function returns a run-time constant on all exits from the function. It is possible, though unlikely, that the return returns something else (like constant 0), and can still be TRE’d. It can be TRE’d if *all other* return instructions in the function return the exact same value.
4. If it can prove that callees do not access theier caller stack frame, they are marked as eligible for tail call elimination (by the code generator).

Utility Passes

This section describes the LLVM Utility Passes.

-deadarghaX0r: Dead Argument Hacking (BUGPOINT USE ONLY; DO NOT USE)

Same as dead argument elimination, but deletes arguments to functions which are external. This is only for use by bugpoint.

-extract-blocks: Extract Basic Blocks From Module (for bugpoint use)

This pass is used by bugpoint to extract all blocks from the module into their own functions.

-instnamer: Assign names to anonymous instructions

This is a little utility pass that gives instructions names, this is mostly useful when diffing the effect of an optimization because deleting an unnamed instruction can change all other instruction

numbering, making the diff very noisy.

`-verify`: Module Verifier

Verifies an LLVM IR code. This is useful to run after an optimization which is undergoing testing. Note that llvm-as verifies its input before emitting bitcode, and also that malformed bitcode is likely to make LLVM crash. All language front-ends are therefore encouraged to verify their output before performing optimizing transformations.

1. Both of a binary operator's parameters are of the same type.
2. Verify that the indices of mem access instructions match other operands.
3. Verify that arithmetic and other things are only performed on first-class types. Verify that shifts and logicals only happen on integrals f.e.
4. All of the constants in a switch statement are of the correct type.
5. The code is in valid SSA form.
6. It is illegal to put a label into any other type (like a structure) or to return one.
7. Only phi nodes can be self referential: `%x = add i32 %x`, `%x` is invalid.
8. PHI nodes must have an entry for each predecessor, with no extras.
9. PHI nodes must be the first thing in a basic block, all grouped together.
10. PHI nodes must have at least one entry.
11. All basic blocks should only end with terminator insts, not contain them.
12. The entry node to a function must not have predecessors.
13. All Instructions must be embedded into a basic block.
14. Functions cannot take a void-typed parameter.
15. Verify that a function's argument list agrees with its declared type.
16. It is illegal to specify a name for a void value.
17. It is illegal to have an internal global value with no initializer.
18. It is illegal to have a `ret` instruction that returns a value that does not agree with the function return value type.
19. Function call argument types match the function prototype.
20. All other things that are tested by asserts spread about the code.

Note that this does not provide full security verification (like Java), but instead just tries to ensure that code is well-formed.

`-view-cfg`: View CFG of function

Displays the control flow graph using the GraphViz tool.

`-view-cfg-only`: View CFG of function (with no function bodies)

Displays the control flow graph using the GraphViz tool, but omitting function bodies.

`-view-dom`: View dominance tree of function

Displays the dominator tree using the GraphViz tool.

`-view-dom-only`: View dominance tree of function (with no function bodies)

Displays the dominator tree using the GraphViz tool, but omitting function bodies.

`-view-postdom`: View postdominance tree of function

Displays the post dominator tree using the GraphViz tool.

`-view-postdom-only`: View postdominance tree of function (with no function bodies)

Displays the post dominator tree using the GraphViz tool, but omitting function bodies.

-transform-warning: Report missed forced transformations

Emits warnings about not yet applied forced transformations (e.g. from `#pragma omp simd`).