Optimization

Given a particular input program, there are many sequences of output target code which produce the correct result. The goal of the optimizer is to find the "best" one. What then is the definition of "best?" There are many possible attributes. More often than not, when one speaks of optimizing, one is concerned with the run-time speed of the target code. In target environments where memory is tight, such as embedded devices, target code size or memory usage may be more important than speed.

When code is generated in a straightforward, mechanical fashion such as described in unit #5, it is easy to see the correspondence between elements of the source code and specific lines of IR and ultimately assembly code. An aggressive optimizer which eliminates redundant operations and restructures code and data flow, may obscure that relationship. This may impact "debugability" of the target code, e.g. when single-stepping the code (at the assembly-language level) with a debugger or emulator, it may not be possible to follow along in the the source program. As a result, the heaviest levels of optimization are often mutually exclusive with source-level debugging (e.g. the `-g` option in UNIX C compilers).

Architecture Dependent and Independent Optimizations

Optimization is the application of a series of transformations to the IR. Each transformation must have the property that it retains the "correctness" of the program. Specifically:
• The optimized program must produce identical output for a given input as the un-optimized program.
• The optimized program must not cause any exceptions (such as a division by 0) that the unoptimized one would not given the same input.
• The optimizer is not responsible for maintaining the correctness of incorrect code. E.g. if in the original program a variable is read before it has been assigned a value, the optimized program is not required to reproduce the same garbage answer.

We can broadly divide optimizations into those which depend on the target architecture, and those which are general. Another way of looking at this is that architecture-independent optimizations transform IR into IR, while dependent optimizations are really part of the target code generation process, i.e. the "back-end."

Examples of architecture-dependent optimizations include selection of the best instructions and addressing modes, scheduling of instructions to optimize pipelines and parallel execution units, and register allocation. Modern high-performance CPUs are very complicated and rely on a good compiler to generate good performing code. Usually the compiler can do a better job than a human on these machine-dependent optimizations. [Some aspects of back-end optimization are discussed in the "target code" unit.] In this unit, we will take an overview of architecture-independent techniques. For a more substantive discussion of optimization, see the texts.

Where does inefficiency come from?

If the task of the architecture-independent optimizer is to transform IR into a "more efficient" IR, we might ask the question of why the original IR is inefficient to begin with? Inefficiency may arise as a result of the IR generation algorithm. It is not always possible to see the most efficient IR for a particular language construct in one pass. It is also often easier to make the IR generation straightforward, and allow the optimizer to improve the code quality.

Inefficiency may also arise due to the programmer (author of the source code) being lazy, ignorant, or stylistic. One goal of high-level languages is to allow programming constructs and concepts to be expressed in higher-level terms, such as expressions, types and objects. This can mean writing code which, if translated literally is not the most efficient way of writing it, but which better presents the meaning or underlying algorithm.

In the 1980s and into the 1990s, C programmers were encouraged to write "efficient code," which often meant expressing the program using certain constructs that the programmer knew would result in more efficient assembly. This was often target-specific. For example, in an integer computation, instead of writing `i=i*16`, the programmer might write `i<<=4` In the former case, the compiler would select a Multiply opcode, which on older processors was often significantly slower than the Shift and Add opcodes which the compiler would generate in the latter case. Even the very existence of operators such as += and ++ was a mechanism for allowing the programmer to "coach" the compiler on good instruction selection. E.g. the += operator says this is really a 2-address operation (destination is the same as one of the source operands). The ++ operator might have triggered an "Incremement" instruction rather than an "Add Immediate", or in the case of pointers, a Register Indirect with auto-post-incremement addressing mode.

This sort of hand-optimization is no longer considered good practice. Although code "bloat" has risen considerably, disk, cpu and memory performance has outpaced it, meaning a modern compiler can do much more thorough analysis and still give shorter compilation times. The best coding style now is to write the program as clearly as possible, and let the compiler's optimizer see opportunities for machine-level efficiency. The programmer may also write inefficient code out of laziness (e.g. cut-and-paste coding) or ignorance of what is or is not efficient. Finally, the programmer may have written errant code, which the optimizer discovers as unreachable or ineffectual code. This last case is often an error, and the compiler would be kind to issue a warning.

Another possible source of inefficiency is source code which is not hand-generated, either because it passed through the standard C preprocessor or was generated by another tool (e.g. lex and yacc).

Optimization therefore is a series of passes in which analysis (e.g. control flow or data flow) is done to understand the program, followed by transformation which can be proven correct as a result of the analysis, for all possible valid run-time inputs. There are situations in which the optimizer can not optimize, because the source language constructs do not provide enough "hints" into the run-time behavior of the program. For example, after a call to an external function, the optimizer must discard any prior knowledge of the values (or constrained ranges of values) of global variables, because it can not prove that the function did not alter them.

The optimizer is also incapable of understanding the algorithms and data structures which the programmer is using. This would elevate the optimizer from being an analysis engine to being an artificially sentient entity. The optimizer can not fix bad design, bad data structures, or bad algorithms. There is still plenty of room for optimized coding on the part of the programmer.

Optimization Phases & Iterative analysis

Optimization is a complicated process. Many of the optimization phases interact with each other. Running optimization A, then running B, might reveal further possibilities for A. In some cases, optimizations conflict, and the result of making a particular transformation will inhibit another optimization from being discovered. Below, an overview of several example optimization phases is presented, without regard to order, and without detailed explanations of the algorithms and implementations involved. Further detail can be found in the texts.

Many of these algorithms are iterative. In general, data flow analysis within a single basic block is straightforward because there is only one control path, and the problem can be solved by linear traversal of the quads, either from first to last or from last to first as the case may require.

However, most real programs extend over multiple basic blocks and contain at least one loop in the control flow graph. In these cases an iterative solution is required, because the results from one basic block may affect the results of another basic block, which affects another, etc. etc. which ultimately affects the first block.

In an iterative solution, we associate one or more sets of values with each basic block and initialize all of these sets to some initial condition. Then we examine each basic block and recompute the per-block sets for each. This process continues until we have made one complete iteration and there are no changes to any of the sets. We say that the iterative solution has then converged. In order for an iterative approach to be correct, it must be shown to converge in finite time, and it must always be "conservative", in that if there is any ambiguity about the analysis, it must err on the side of preserving the original code.

The texts contain a theoretical presentation on the framework on which any iterative data flow analysis may be constructed. What will be described here are informal examples.

Finding useless or unreachable code

Given that the IR for a given function (or the entire program) is modeled with a Control Flow Graph (CFG) having an initial node and a final node, there may be nodes which have no incoming edges in the CFG. These nodes can never be reached during program execution, and that fact is apparent from simple analysis, without looking at data flow. This situation sometimes happens as an artifact of code generation techniques. It is also possible that the programmer has made an error. Later, we'll see some techniques which combine data and control flow analysis to find other blocks which are not as obviously unreachable.

Useless code is that which produces no useful result. Useful results in the C language are: returning from the function, calling a function (other than inline functions), modifying a global variable, modifying a value through pointer dereferencing (unless we know that the pointer is not pointing at something "useful"). In the following example:

```
int f()
{
 int a;
          g(1,2);
          a=1;
          return 2;
}
```

the call to `g()` is potentially useful (we don't know unless we can analyze `g()` too), as is the return statement. The assignment to the local variable `a` is useless and can be eliminated.

Like unreachable code, useless code may be generated by accident during IR generation, or it may be programmer error, or possibly stylistic (e.g. a body of code which is disabled but which serves a debugging role).

To find useless code, we can use an iterative algorithm. First, we visit all IR operations and mark those which are obviously useful (they meet one of the 4 criteria above). Recall that each IR operation is in the form (src1 OP src2 -> dst). If the operation is useful, we trace back src1 and src2 to the operation(s) in which they receive their value, and mark that operation as useful. We continue this until we are no longer discovering additional useful IR quads.

Then for each basic block which contains at least one useful quad, we walk back in the CFG and visit all **decision points** which determine whether that useful basic block is executed. A basic block D is a decision point for basic block B if there are two or more paths leaving D, one of which leads to B, and the other does not. All decision points for useful basic blocks are marked as useful operations (i.e. we mark the conditional branch as useful, then trace back the operands leading to that decision and mark them as well).

When all of this is done, we have marked useful vs useless quads and branches. Any useless branches can be replaced with unconditional jumps directly to the nearest useful basic block in the CFG. Then all useless quads can be deleted from their containing basic blocks. As a result of this, additional optimizations, such as finding and deleting empty basic blocks, or merging basic blocks, may result.

A second example:

```
void f(a,b)
int a,b;
{
 extern int e;
 int l;
         e++;
         if (a<b)
                  l++;
         e++;
}

Resulting IR:
BB0:
         e=       ADD      e,1                 *useful
         CMP      a,b
         BRGE     BB2,BB1
BB1:
         l=       ADD      l,1
         BR       BB2
BB2:
         e=       ADD      e,1                 *useful
```

The first and last ADD are marked as useful. The BRGE is not a useful decision point, because it does not affect whether useful operations take place (BB1 is not useful). Therefore, it can be replaced by an unconditional branch, and the CMP instruction is not useful. The transformed code is:

```
BB0:
         e=       ADD      e,1
         BR       BB2
BB2:
         e=       ADD      e,1
```

Subsequent optimizations will merge BB0 with BB2 (if there are no other branches to BB2) and may even be able to replace the two ADD instructions with one.

Useless control flow, Branch Folding, Empty Basic Blocks

We have already discussed unreachable basic blocks, i.e. a basic block with no incoming edges. This entire block can be deleted, along with any edges leaving that block. This, in turn, may expose additional blocks as being unreachable. Pruning unreachable blocks reduces code size but does little to improve execution speed.

Branch folding fixes a situation which may arise during optimization. Let us say that BB4 has two conditional exits, both of which are directly to BB5, because other optimization passes have removed or coalesced intervening basic blocks. The conditional branch (and the compare or whatever sets the condition code) can clearly be removed and replaced with an unconditional branch. This may then pave the way for combining basic blocks. If there is a single edge connecting BB4 with BB5, and no other edges enter BB5, then BB4 and BB5 can be merged into a single basic block.

Empty basic blocks frequently arise during optimization. An empty block which contains only an unconditional branch can be removed. Any edges which were pointing to the empty block are then retargeted at the successor to the empty block. If the empty block has multiple exits (i.e. a conditional branch), this is probably a bad sign, because the determinant for the conditional would appear to have vanished!

Code Motion

Optimizing transformations in which code is relocated (but otherwise not changed) are known as code motion. A simple example is factoring a loop invariant:

```
do
{
        a=b*c;
        z+=a/f(z);
} while (z<q)
```

Assume that b and c are local variables and thus can not be accessed by function f(). The calculation of `b*c` can be shown to involve values which can not change during the course of the loop. The resulting value in `a` does not depend on how many times the loop executes. Therefore, that entire expression is invariant with respect to the loop and can be moved outside of it.

Reduction in Strength

The Operator Strength Reduction optimization looks for "expensive" operators inside a loop in which one operand is a *region constant*, i.e. its value does not change within the loop, and the other is an *induction variable* whose value changes consistently with each iteration. This expensive operation can be replaced with a weaker, less expensive operation which yields the same result by creating a different induction variable which "factors out" the expensive operation.

The classic example is accessing an array within a loop:

```
int a[1024];
for(i=0;i<1024;i++)
{
```

```
        sum+=a[i];
}
```

Recall that `a[i]` is equivalent to `*(a+i)`, and that the addition of the pointer (the array name) to the integer contains an inherent multiply. I.e. within the loop we will see something like:

```
        i=      MOV     0
BB100:
        T100=   LEA     a
        T101=   MUL     i,4
        T102=   ADD     T100,T101
        T103=   LOAD    (T102)
        sum=    ADD     sum,T103
BB101:
        i=      ADD     i,1
                CMP     i,1024
                BRLT    BB100,BB102
BB102:
```

Now if the LEA is moved outside of the loop, so T100 is initialized to the base address of the array, we could make it the induction variable:

```
        i=      MOV     0
        T100=   LEA     a
BB100:
        T103=   LOAD    (T100)
        sum=    ADD     sum,T103
BB101:
        i=      ADD     i,1
        T100=   ADD     T100,4
                CMP     i,1024
                BRLT    BB100,BB102
```

This is an improvement if, on the architecture in question, integer multiplication is "harder" than integer addition.

As a further optimization, called Linear Function Test Replacement, we could change the control variable of the loop to make it the induction variable directly:

```
        T98=    LEA     a
        T99=    ADD     T99, 4096
BB100:
        T103=   LOAD    (T100)
        sum=    ADD     sum,T103
BB101:
        T100=   ADD     T100,4
                CMP     T100,T99
                BRLT    BB100,BB102
BB102:
        i=      MOV     1024     #Not needed if i is not live
```

Loop unrolling and unswitching

Loop unrolling trades code space for execution speed:
```
for(i=0;i<n;i++)
          a[i]=b[i]+c[i];

unrolled:
for(i=0;i<n;i+=4)
{
          a[i]=b[i]+c[i];
          a[i+1]=b[i+1]+c[i+1];
          a[i+2]=b[i+2]+c[i+2];
          a[i+3]=b[i+3]+c[i+3];
}
```
(this example is somewhat over-simplified in that is assumes n is a multiple of 4). Each iteration of the loop involves some overhead of the comparison and branch. By unrolling the loop, we get more computation for each iteration. If through constant propagation analysis (to be covered shortly) we can learn that n has a known, constant value, then the entire for loop can be removed and replaced with the known number of unrolled iterations.

Loop unswitching factors a conditional out of the loop, when the determinant of the conditional is invariant with respect to the loop:
```
for(i=0;i<n;i++)
{
          if (x>y)
                    a[i]=b[i]+c[i];
          else
                    a[i]=b[i]*c[i];
}

transformed:

if (x>y)
          for(i=0;i<n;i++)
                    a[i]=b[i]+c[i];
else
          for(i=0;i<n;i++)
                    a[i]=b[i]*c[i];
```
Again, the same number of iterations are performed, but we eliminate a useless comparison and branch inside the loop, where there are many potential iterations, and move it to a less costly place outside the loop.

Data Flow Analysis

We'll now look at several algorithms which analyze the flow of data within the program. Knowledge gained about data flow can guide further optimizations, e.g. the elimination of computations where the result will not be used ("live variables") or the elimination of duplicate computations ("available expressions").

Live Variables

The concept of whether a variable is "live" or not at a particular point in the code is very critical both for architecture-neutral IR optimization (this unit) and for optimal register allocation (subsequent unit). Definition: A name is **live** at a given point in a program if there is a subsequent *reachable* point in the control flow graph in which that name is used, and there is no intervening place where that name *could be* redefined.

Live variable analysis is an iterative algorithm that works "backwards" : given a particular point where a variable is used, we work back through preceeding operations looking for places where that variable is set. The result is a set of sets, LIVEIN[B] and LIVEOUT[B]. A variable in LIVEIN[B] is referenced in B before it is overwritten. A variable in LIVEOUT[B] is in the LIVEIN set of at least one successor block to B.

```
for each basic block B
          pre-compute the sets DEF[B] and REF[B]
             DEF: variables that are assigned to in B
             REF: variables that are used as src1 or src2, prior
                    to being assigned to
          initialize LIVEIN[B]=emptyset

do {
          for each basic block B
                    Compute LIVEOUT[B] the union of
                              all LIVEIN[i], where i iterates
                              over all the immediate successors to
                              B in the CFG
                    LIVEIN[B]=REF[B] UNION (LIVEOUT[B] - DEF[B])
}
while (LIVEIN or LIVEOUT changed during last iteration)
```

The initial conditions of this analysis are that the LIVE sets are empty, i.e. there are no variables known to be live. The LIVEOUT sets are immediately computed as the union of LIVEIN sets of successor blocks. Since all LIVEIN sets are initialized to empty, that will also be the initial value of LIVEOUT. Since the EXIT node of the CFG has no successors, its LIVEOUT set will never change, which upholds the boundary condition that no variables are live at the exit of the function.

DEF is the set, for each basic block, of variables which are defined (assigned to) at some point in the block. REF is the set of variables which are used as a source operand, **prior** to any assignment to that variable. REF and DEF can be computed by a single linear pass over the quads in the block. They only need to be computed once, because they are not affected by conditions outside of the block.

Any variable which is live on exit from the block (in the LIVEOUT[B] set) is also live on entry (LIVEIN[B]), except that variables in the DEF[B] set "kill" the liveness coming from the end of the block. Any variable in the REF[B] set must also be live at the input to the block. Therefore, the iteration correctly computes LIVEIN for each block given what we currently know about LIVEOUT for that block.

Iteration is required because there could be loops in the CFG, thus the a given basic block could be both a predecessor and a successor to another given BB, thus creating loops in the data flow as well. We can show that this iteration must eventually converge. At initialization, all LIVEIN and LIVEOUT sets were empty. At each iteration, the LIVEIN and LIVEOUT sets can only grow, because the only operation applied to them is set union. Since there is a finite number of variables, there is a limit to the set growth. Therefore there must be a finite number of iterations after which the sets will not grow further.

Given *LIVEOUT[B]*, we can work backwards in the basic block, building a set *LIVENOW* for each operation (quad) in the block, which contains the list of names which are live as of the completion of that operation. We can compute this set as follows:

```
LIVENOW[N]=LIVEOUT
for(i=N-1;i>=1;i--)          //quads numbered 1...N
{
        LIVENOW[i]=LIVENOW[i+1];
        remove from LIVENOW[i] the dest operand of quad i+1
        foreach (source operand of quad i+1)
                add this variable to LIVENOW[i]
}
```

The order of removing the destination before adding the source is important when considering an operation which uses a name as both a source and a destination.

A given name may be live in certain places in the code, and dead in other places. We call this a **live range**. Here is an example finding the live range of variable c, after global live variable analysis tells us that c is not in LIVEOUT of this BB:

```
1       c=      ADD     a,b
2       e=      ADD     c,d
3       c=      ADD     x,y
4       i=      ADD     g,h
5       l=      ADD     j,k
6       c=      ADD     p,q
7       w=      ADD     c,u
        [Assume LIVEOUT from this BB does not include c]
```

We know from LIVEOUT that variable c is not live at the end of instruction 7. It is live at 6 and at 1, creating two live ranges in the basic block for c: 1..1 and 6..6. Since c is not used as a source operand in instructions 4, 5 or 6 before being overwritten at 6, operation #3 was useless. Normally, the optimizer will have removed this useless operation.

Available Expressions

The Available Expressions analysis can inform us where we are needlessly recomputing a value which is already known. First, we need to define **AVAIL**able expressions. An expression, for these purposes, is a srcA op srcB 3-tuple. We say that an expression is AVAIL on entry to a particular basic block if the result has already been computed along all possible paths leading to this block from the entry node, and there have been no subsequent assignments to srcA or srcB after the computation.

```
for each basic block B
        pre-compute GEN[B], the set of expressions computed in B
                and not subsequently killed
        compute KILL[B], the set of expressions killed in B
        initialize AVAILOUT[B]=all possible expressions
do {
        for each basic block B
                AVAILIN[B]=INTERSECTION(foreach P,
                        a direct predecessor of B, AVAILOUT[P])
                AVAILOUT[B]=GEN[B] UNION (AVAILIN[B] - KILL[B])
} while (changes in any AVAILIN or AVAILOUT set)
```

While Live Variables was a backwards data flow analysis, Available Expressions is a forwards problem. In Live Variable analysis, the "meet" operator was a set union: the known live variables flowed backwards from the successor blocks to the predecessor blocks and their union formed LIVEOUT. In Available Expression analysis, the meet operator is set intersection: known available expressions flow forwards from predecessors to successor, and AVAILIN is the intersection of these values. The reason for intersection is that for an expression to be available, it must be available on ALL the paths leading to the block. On the other hand, for a variable to be live on exit from a block, it suffices that it is live on entry to just one of the successors.

For implementation, we could construct a hash table of srcA,op,srcB tuples and assign each an integer index. Then the set of available expressions can be represented by a bit vector. When we make our initial linear pass over all basic blocks to compute the GEN and KILL sets, any assignment to srcX "kills" all tuples (srcA,op,srcB) where A==X or B==X. Any quad of the form dst=srcA op srcB adds to the GEN set, unless (srcA,op,srcB) is killed later in that block.

We can show convergence. The initialization condition for all blocks is the "Universal Set", which is the identity for set intersection. At each iteration, set intersection is applied which can only diminish the size of the set.

Once available expressions have been identified, the optimizer can create a temporary name at the computation site, and replace subsequent redundant uses with a reference to the temporary. The text contains further algorithms which determine the optimum time to introduce this computation.

Reaching Definitions

Another forward data flow problem is Reaching Definitions. We can define each quad X=srcA op srcB as a definition point and assign it a unique number (note similarity to SSA form covered later). A given definition *reaches* a particular point in the code if, along any and all paths leading from the definition to that point, there are no subsequent assignments. A particular quad X=srcA op srcB kills any other definitions which involve X.

```
for each basic block B:
        compute KILL[B], the set of all definitions killed
        compute GEN[B], the set definitions created in this
                block and not killed in this block
        initialize REACHOUT[B]=empty

do {
        REACHIN[B]=UNION(all predecessors P,REACHOUT[P])
        REACHOUT[B]=GEN[B] UNION (REACHIN[B] - KILL[B])
} (while changes in any REACHIN or REACHOUT set)
```

Analysis of this algorithm is similar to the other two iterative dataflow algorithms presented above.

SSA

Some data flow problems are cumbersome because a particular variable can have different values depending on control flow. Static Single Assignment Form (SSA) is a way of re-writing the IR so that there is only one defining (i.e. assigning) instance for any given variable. This is of great assistance to the optimizer.

SSA does not, by itself, produce any optimization. It is merely a way to make certain optimization algorithms easier. Generally speaking, at some point in the optimization process, the IR is transformed into SSA form, then optimizations that work better in SSA are performed, then the code is translated back into non-SSA form. If there are several such optimizations, the time taken to convert into and out of SSA form might be justified by the increased execution speed of the optimization passes.

We introduce a value number, or subscript, for each variable. Let's say it begins at 0, so instead of x,y z, etc. our variables are initially numbered x0, y0, z0, etc. Each instance of writing to that variable causes the number to be incremented. Subsequent uses of that variable track the value number. E.g.

```
 # Original form of quads:
        x=      ADD     x,y
        z=      ADD     x,y
 # SSA transformation:
        x1=     ADD     x0,y0
        z1=     ADD     x1,y0
```

How do we then reconcile the use of a name when there are multiple edges entering a basic block? If that name is defined differently among those edges then it would be ambiguous which value-numbered name to use. The solution is to introduce what are known as phi-functions. This is a symbolic notation which indicates that the value is the one corresponding to entering on a particular edge.

For example, let's look at the following IR:

```
BB0:
          a=        ADD       b,c
          ...
          BR_xx     BB1,BB2
BB1:
          a=        ADD       e,f
          BR        BB4
BB2:
          a=        ADD       x,y
          BR        BB4
BB4:
          k=        ADD       j,a
```

Re-written in SSA form:

```
BB0:
          a0=       ADD       b0,c0               #Maybe optimized out as useless
          ...
          BR_xx     BB1,BB2
BB1:
          a1=       ADD       e0,f0
          BR        BB4
BB2:
          a2=       ADD       x0,y0
          BR        BB4
BB4:
          a3=       PHI(BB1:a1,BB2:a2)
          k0=       ADD       j0,a3
```

Of course, the phi function does not correspond to any typical assembly language instructions (although there is some similarity to super-scalar processor architecture). Once the IR has been rewritten in SSA form and subjected to numerous optimizations, it must be rewritten back into non-SSA form before being translated to target code. Generally speaking, this means inserting additional basic blocks which contain copy operations along each entering edge which is affected by phi function merges.

The reader is referred to the Dragon Book text, or *Engineering a Compiler* by Cooper and Torczon for more information on SSA algorithms.

Constant Propagation and Conditionally Unreachable Code

Expressions involving only constants are a natural artifact of preprocessors and certain coding styles. If we have `a=MACROA*MACROB`, and `#define MACROA 1` and `#define MACROB 2`, then we can easily replace this with `a=2` and eliminate the multiply operation at run time. The replacement can be done during quad generation and does not require extensive analysis. E.g. whenever we are asked to generate quads for a multiply instruction, and both operands are constants, we instead generate a simple copy instruction. We discussed this under the heading of "Constant Folding" when examining quad generation.

However, other optimizations involving constants are not that easy to see at compile time without further analysis:

```
z=17;
a=z+2;
```

It is obvious what to do here, but in order for the compiler to see it, it must be able to trace the propagation of the constant value into the variable z and know that z still has that same known value when the + expression is reached.

We will assume in this example that the IR is in Static Single Assignment (SSA) form, meaning the quads are in the form SAi OP SBj -> Dk, and each of these names refers unambiguously to a distinct value which is never overwritten. Then we can keep track of Value(Dx) for each SSA destination Dx. We do so using a system of accounting where Value(Dx) can be UNDEF, KNOWN:n, UNKNOWABLE. The state of UNDEF means that we have not yet learned if the value is knowable or unknowable. The UNKNOWABLE state is the lowest state, meaning we positively know that we can't know the value. An example would be any value which is external to our analysis (e.g. global variables if we are doing strictly intra-function analysis). The KNOWN state means we have proven that the SSA name Dx has a particular value n. The algorithm is iterative and operates correctly because these states have a ranking, and Value(x) can only stay the same or go down. I.e. there will never be a case where UNKNOWABLE gets "promoted" to KNOWN or UNDEF.

The analysis begins by marking all basic blocks as unreachable, with the exception of the initial node, which is obviously reachable. We also mark all SSA names as UNDEF, except those which are obviously UNKNOWABLE. Iteratively, we look at each reachable basic block, and analyze each SSA operation within, according to the following rules:

If the operation is x=c, where c is a constant and x is an SSA name, then we mark Value(x)=KNOWN:c.

If the operation is x=phi(y,z), then we use the "meet rules":

UNKNOWABLE meet anything = UNKNOWABLE
UNDEF meet UNDEF = UNDEF
UNDEF meet KNOWN:c = KNOWN:c
KNOWN:c1 meet KNOWN:c2 {c1!=c2} = UNKNOWABLE
KNOWN:c meet KNOWN:c = KNOWN:c

For all other operations, we try to exploit algebraic identities to improve our analysis. E.g. if the operation is multiply, then
KNOWN:0*anything = KNOWN:0

KNOWN*UNKNOWABLE = UNKNOWABLE
KNOWN:a*KNOWN:b = KNOWN:a*b.
UNDEF*UNKNOWABLE = UNDEF
With regard to the last equation, we might think the right answer is UNKNOWABLE, but the UNDEF might later be discovered to be KNOWN:0. If we marked the result prematurely as UNKNOWABLE, we would not be able to go back later and learn the fact that the result must be 0.

We can devise similar meet rules for other operators such as addition, and for comparison operators. E.g. when evaluating a>b, if a and b are unsigned and a is KNOWN:0, then the expression must always be false, regardless of b.

At the end of each basic block, we look at conditional branches. If the condition is either KNOWN:TRUE or UNKNOWABLE, we must mark the true leg of the branch as reachable. Likewise for FALSE. If the branch is unconditional, we mark the target as reachable.

This process continues iteratively until things stop changing, at which point we have discovered all of the constant values which are knowable, and all of the blocks which are knowably unreachable. We can then remove the operations which compute the constant values, and prune the unreachable code.