Constant Propagation: A Fresh, Demand-Driven Look*

Eric Stoltz, Michael Wolfe, and Michael P. Gerlek
Department of Computer Science, Oregon Graduate Institute of Science & Technology

Keywords: constant propagation, compiler optimization, SSA, demand-driven analysis

Abstract

Constant propagation is a well-known static compiler technique in which values of variables which are determined to be constants can be passed to expressions which use these constants. Code size reduction, bounds propagation, and dead-code elimination are some of the optimizations which benefit from this analysis.

In this paper, we present a new method for detecting constants, based upon an optimistic *demand-driven* recursive solver, as opposed to more traditional iterative solvers. The problem with iterative solvers is that they may evaluate an expression many times, while our technique evaluates each expression only once. To consider conditional code, we augment the standard Static Single Assignment (SSA) form with merge operators called γ-functions, adapted from the interpretable Gated Single Assignment (GSA) model.

Introduction

Constant propagation is a static technique employed by the compiler to determine values which do not change regardless of the program path taken. In fact, it is an application of abstract interpretation which generalizes *constant folding*, the deduction at compile time that the value of an expression is constant, and is frequently used as a preliminary to other optimizations. The results can often be propagated to other expressions, enabling further applications of the technique. It is this recursive nature of the data-flow problem which suggests using a *demand-driven* method instead of the more usual iterative techniques.

In the following example, the compiler substitutes the value of 5 in S1 for x, which is a canonical instance of constant folding. Since the value of x is now constant, the compiler can propagate this value into S2, which, after applying constant folding once again, results in the determination that y is the constant 20.

```
S1:    x = 2 + 3
S2:    y = 4 * x
```

*This research supported by NSF grant CCR9113885, ARPA grant F3062-92-C-135, and grants from Intel Corporation and Matsushita Electric Industrial.

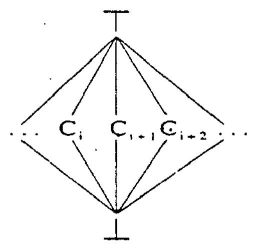

Figure 1: Standard constant propagation lattice.

Although in general constant propagation is an undecidable problem, it is nonetheless extremely useful and profitable for a number of optimizations. These include dead code elimination [10], array- and loop-bound propagation, and procedure integration (inlining), which we believe to be a major source of detectable constants [5]. Due to these benefits, constant propagation is an integral component of modern optimizing commercial compilers.

The paper is organized as follows. We first examine the standard framework employed to perform constant propagation and relate it to previous methods and algorithms. Second, we define the structure used for this work, with particular attention given to the necessary intermediate form (based upon Static Single Assignment) required to implement the algorithms we present. Next, the extension of the method to conditional code is given, and we discuss variables within loops, which we have found to be closely tied to induction variable recognition. We close with future directions and conclusions.

Background and Other Work

Constant propagation operates on a standard 3-level lattice, as shown in Figure 1. Each symbol has its lattice value initialized to Top ($\top$). When comparing two lattice element values, the meet operator ($\sqcap$) is applied, as given in Figure 2. After analysis is complete, all symbols will have lattice value equal to bottom ($\perp$) when it cannot be determined to be constant, a constant value, or $\top$ for unexecutable code. These foundations, originally introduced by Kildall [7], are standard for many constant propagation methods [5, 10]. We note that values can only move down in the lattice, due to the meet operator. By initializing lattice values to $\top$, an optimistic approach is taken, which assumes all symbols can be determined to be constant until proven otherwise.

Previous methods perform the analysis as an iterative data-flow problem, in which iterations continue until a fixed point is reached. We will see in the next section that an alternative

$$
\begin{array}{llll}
\top & \sqcap & \text{any} & = \text{any} \\
\bot & \sqcap & \text{any} & = \bot \\
\text{constant}_i & \sqcap & \text{constant}_j & = \begin{cases} \text{constant}_i & \text{if } i = j \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

Figure 2: Rules for meet ($\sqcap$) operator.

```
z = 3                  z =  3
if ( P ) then          if ( z < 5 ) then
   y = 5                  y = 5
else                   else
   y = z + 2              y = 2
endif                  endif

   (a)                    (b)
```

Figure 3: Constant propagation with (a) simple, and (b) conditional, constants.

demand-driven recursive algorithm offers advantages over the traditional approach.

Classification of Methods

As explained by Wegman and Zadeck [10], constant propagation algorithms can be classified in two ways: (*i*) using the entire graph or a sparse graph representation, and (*ii*) detecting simple or conditional constants. This naturally creates four classes of algorithms. It is clear that propagating information about each symbol to every node in a graph is inefficient, since not all nodes contain references or definitions of the symbol under consideration. Sparse representations, on the other hand, such as def-use or use-def chains, Static Single Assignment (SSA) [2], Dependence Flow Graphs (DFG) [6], or Program Dependence Graphs (PDG) [3], have all shown the virtue of operating on a sparse graph for analysis.

The distinction between simple (all paths) constants and conditional constants can be seen in Figure 3. The simple value of y is determined to be constant only if both branches which merge at the **endif** are constant with identical value, as is the case in (a). However, if the predicate which controls branching can be determined to be constant, then only one of the branches will be executed, allowing not only y to be recognized as constant in (b), but also identifying the other path to be dead code.

The distinction between the four types of algorithms is explained well by Wegman and Zadeck [10], and the reader is referred to their paper for more detail. We will look at the algorithm that they present, since it incorporates both sparse graph representation and conditional code. The sparse graph employed is the SSA form, described next.

Graph Preliminaries

The algorithms to convert a program into SSA form are based upon the *Control Flow Graph* (CFG), which is a graph G = < *V*,*E*,*Entry*,*Exit*>, where *V* is a set of nodes representing basic blocks in the program, *E* is a set of edges representing sequential control flow in the program, and *Entry* and *Exit* are nodes representing the unique entry point into

```
x = 0                          x0 = 0
y = 0                          y0 = 0
z = 0                          z0 = 0
if ( P ) then                  if ( P ) then
   y = y + 1                      y1 = y0 + 1
endif                          endif
                               y2 = φ ( y0, y1 )
x = y                          x1 = y2
z = 2 * y - 1                  z1 = 2 * y2 - 1
   (a)                            (b)
x0 = 0
y0 = 0
z0 = 0
if ( P ) then
   y1 = y0 + 1
endif
y2 = γ ( P, true→ y1, false→ y0 )
x1 = y2
z1 = 2 * y2 - 1
   (c)
```

Figure 4: Program in (a) normal form, (b) SSA form, and (c) GSA form.

the program and the unique exit point from the program. *Branch* nodes have their outgoing edges determined by a *predicate*. After a program has been converted into SSA form, it has two key properties:

1. Every use of a variable in the program has exactly one reaching definition, and
2. At confluence points in the CFG, merge functions called *ϕ-functions* are introduced. A ϕ-function for a variable merges the values of the variable from distinct incoming control flow paths (in which a definition occurs along at least one of these paths), and has one argument for each control flow predecessor. The ϕ-function is itself considered a new definition of the variable.

For details on SSA graph construction the reader is referred to the paper by Cytron et al. [2]. A sample program converted into SSA form is shown in Figure 4(a) and (b).

A Closer Look at One Algorithm

The constant propagation algorithm described by Wegman and Zadeck uses two worklists, a work list of flow edges (edges from the CFG graph all initially marked *unexecutable*), and a work list of SSA edges (def-use edges added to the CFG graph once the program has been transformed into SSA form). Edges are examined from either worklist until empty, with flow edges being marked *executable*. Expressions are evaluated the first time a node is the destination of a flow edge, and also when the expression is the target of an SSA edge and at least one incoming flow edge is executable. The destination node for these edges also have their ϕ-functions evaluated by taking the meet of all the arguments whose corresponding CFG predecessors are marked *executable*. More detail can be found in their paper [10].

This algorithm finds all simple constants, plus additional constants that can be discovered when the predicate controlling a branch node is determined to be constant. The time complexity is proportional to the size of the SSA graph, and each SSA edge can be processed at most twice.

Since ϕ-functions are re-evaluated each time an edge with that node as a destination is examined, Wegman and Zadeck note that expressions which depend on the value of a ϕ-function may be re-evaluated twice for each of its operands. For example, in this program fragment:

```
if ( P )
then
  y1 = 1
  z1 = 2
else
  y2 = 1
  z2 = 3
endif
y3 = φ(y1, y2)
z3 = φ(z1, z2)
x1 = y3 + z3
```

if P is not constant, the expression for x_1 may be evaluated many times. If the flow edge from the true branch for P is processed first, then x_1 equals 3, and it may stay at 3 if the SSA edges for y are examined next. Eventually, x_1 will evaluate to $\perp$, as the merge for z becomes non-constant. It is this multiple expression evaluation which we seek to avoid.

FUD Chains and Simple Constants

Our method also converts the intermediate representation into SSA form. In order to achieve the single-assignment property each new definition of a variable receives a new name. Practically, however, this is undesirable (managing the symbol table explosion alone precludes this option), so the SSA properties are maintained by providing links between each use and its one reaching definition. Instead of providing def-use links, as is the common implementation [6, 10], we provide use-def links, giving rise to an SSA graph comprising factored use-def chains (FUD chains). This approach yields several advantages, such as constant space per node and an ideal form with which to perform demand-driven analysis [8].

Our analysis of programs begins within a framework consisting of the CFG and an SSA data-flow graph. Each basic block contains a list of intermediate code tuples, which themselves are linked together as part of the data-flow graph. Tuples are of the form *<op,left,right,ssalink,lattice>*, where *op* is the operation code and *left* and *right* are the two operands (both are not always required, e.g. a unary minus). The *ssalink* is used for fetches and arguments of ϕ-functions, as well as indexed stores (which are not discussed further in this paper). The *ssalink*, if applicable, represents the one reaching definition for the variable in question at that point in the program. The *left*, *right*, and *ssalink* fields are pointers: they are all essentially use-def links. Thus, to perform an operation associated with any tuple, a request (or *demand*) is made for the information at the target of these links. Each tuple also has a lattice element, *lattice*, assigned to it, initialized to $\top$.

We first show how to implement simple constant propagation within our framework. Our algorithm efficiently propagates simple constants in the SSA data-flow graph by demanding the lattice value from the unique definition point of each use. We visit all CFG nodes, examining each of its tuples, calling *propagate*() recursively on any unvisited left or right tuples. Assignment statements are evaluated, calling *propagate*() on all references with an *ssalink*. When a ϕ-function is encountered, recursive calls to the arguments are made, followed by taking the meet of those arguments. In the case of data-flow cycles, characterized by ϕ-functions at loop-header nodes, $\perp$ is returned. The algorithm is given in Figure 5.

```
∀ t ∈ tuples,
    lattice( t ) = ⊤
    unvisited( t ) = true

Visit all basic blocks B in the program
    Visit all tuples t within B
        if unvisited ( t ) then propagate( t )

propagate ( tuple t )
    unvisited ( t ) = false
    if ssa_link( t ) ≠ ∅ then
        if unvisited( ssa_link( t ) )
        then propagate( ssa_link( t ) )
        lattice( t ) = lattice( t ) ⊓ lattice( ssa_link( t ) )
    endif
    if unvisited( left ( t ) ) then propagate( left( t ) )
    if unvisited ( right ( t ) ) then propagate( right( t ) )
    case on type (t )
        constant C: lattice( t ) = C
        arithmetic operation:
            if all operands have constant lattice value
            then lattice( t ) = arithmetic result of
                    lattice values of operands
            else lattice( t ) = ⊥
            endif
        store: lattice( t ) = lattice( RHS )
        φ-function:
            if loop-header φ then lattice( t ) = ⊥
            else lattice( t ) = ⊓ of φ-arguments of t
            endif
        default: lattice( t ) = ⊥
    end case
end propagate
```

Figure 5: Demand-driven propagation of simple constants.

Several points are worth noting with respect to this algorithm:

- This is not an iterative solver. It is a recursive demand-driven technique which will completely solve the problem in the absence of cycles. The order in which basic blocks are visited is not important.
- When at a merge node, we take the meet of the demanded classification of the ϕ-arguments. By Figure 2, this will result in a constant iff all ϕ-arguments are constant and identical.
- Each expression is evaluated once, since the node containing the expression will only be evaluated after all referenced definitions are classified.
- The asymptotic complexity is proportional to the size of the SSA data-flow graph, since it requires each SSA edge to be examined once.

- In the presence of data-flow cycles (due to loops in the CFG), the solver will fail to classify constant valued tuples, either as a function of the loop's trip count or even if it remains constant throughout the loop. A more complex solver, such as the one described in the next section, is needed to account for cycles.

Conditionals and Loops

Extending SSA to GSA

When demanding the classification of a variable at a merge node, we take the meet of the demanded classification of its ϕ-arguments, as noted in the last section. However, if only one of the branches can ever be taken, we would like to only propagate the value along that path. In previous methods, as illustrated earlier, the predicate at a branch node is first evaluated, and if found constant, the executable edge is added to a worklist. Thus, when the corresponding merge node is processed, expressions at that node will be evaluated in terms of the incoming executable edges. As we have seen, this may result in expressions being evaluated more than once.

In our method, if a symbol demands the value from a merge node, we want to process the predicate that determines which path to follow. Examine Figure 4(b). When attempting to classify x_1, the value is demanded from the use-def SSA link of y_2, which points to the ϕ-function. However, a ϕ-function is not interpretable [1]. Thus, we have no information about which path may or may not be taken. Since the predicate P in our example determines the path taken, if P is constant we can determine which argument of the ϕ-function to evaluate. If P is not constant, the best we can do is to take the meet of the ϕ-arguments.

Augmentation of the ϕ-function is needed to include this additional information. We extend the SSA form to a *gated single assignment* form (GSA), introduced by Ballance et al.[1], which allows us to evaluate conditionals based upon their predicates. Figure 4(c) shows a simple program converted to GSA form. Briefly, ϕ-functions are reclassified into μ- and γ-functions. Most ϕ-functions contained within loop-header nodes are renamed μ-functions, while most other ϕ-functions are converted to γ-functions[*]. The γ-function, $v=\gamma(P, true \rightarrow v_1, false \rightarrow v_2)$, means *if* P *then* $v=v_1$ *else* $v=v_2$. In this form, the γ-function represents an `if-then-else` construct, but it is also extended to include more complex branch conditions, such as **case** statements. We provide the complete algorithm to convert ϕ-functions to μ- and γ-functions in our technical report [9].

Multiple levels of conditionals result in nested γ-functions. Examine Figure 6(a), which is an unstructured code fragment (although structured code with nested `if-then` constructs also result in nested γ-functions). Figure 6(b) shows the program translated into GSA form. It is quite an interesting example for constant propagation, since if we know the value of predicate P we always know what possible value of x can reach the merge at 40. However, if we don't know P, then the value of predicate Q becomes crucial:

- If Q is true, only x_1 can reach 40.
- If Q is false, we have no clear information on what value of x to propagate.

[*]A ϕ-function cannot be converted to a μ- or γ-function in the presence of irreducible loops.

```
   x = 2
   if ( P ) goto 30
   if ( Q ) goto 50
   else goto 40
30 x = 3
40 y = x
50 continue
      (a)
```

```
   x0 = 2
   if ( P ) goto 30
   if ( Q ) goto 50
   else goto 40
30 x1 = 3
40 x2 = γ( P, t→x1, f→γ ( Q, t→ ⊤, f→ x0) )
   y1 = x2
50 continue
      (b)
```

Figure 6: Conditional code which results in nested γ-functions

Conditional Constant Propagation

Once converted into GSA form, we can improve upon the *propagate*() routine to take advantage of predicates that can be determined to be constant. When encountering a γ-function, we first attempt to evaluate the predicate. If constant, we follow the indicated branch, propagating constant values as found. If not constant, we take the meet of its arguments. The revised algorithm is given in Figure 7.

Loops

Cycles in the SSA or GSA data-flow graph are the result of loops within the original program. Variables defined within these cycles are classified with induction variable analysis. Induction variables are traditionally detected as a precursor to strength reduction, and more recently for dependence analysis with regard to subscript expressions. We have developed methods for detecting and classifying induction variables based on strongly-connected regions in the SSA data-flow graph [11]. These techniques make use of an exit function, the η-function, which holds the exit value of a variable assigned within the loop. The exit value may be a function of the loop trip count (which may itself be an expression determined to be constant), or may be invariant with respect to the loop. An exit expression is held by the η-argument, which, if constant, can be used to propagate values outside of the loop. Ballance et al. introduced η-functions in their GSA form. We place η-functions in *postexit* nodes as part of our SSA translation phase. For each edge exiting the loop, a postexit node is inserted outside the loop, between the source of the exit edge (within the loop body) and the target (outside the loop).

We are able to propagate constants through loops (single and nested) by taking advantage of specialized solvers which detect and classify a large assortment of linear and non-linear induction variables [4].

```
∀ t ∈ tuples,
    lattice( t ) = ⊤
    unvisited( t ) = true

Visit all basic blocks B in the program
    Visit all tuples t within B
        if unvisited ( t ) then propagate( t )

propagate ( tuple t )
    unvisited ( t ) = false
    if ssa_link( t ) ≠ ∅ then
        if unvisited( ssa_link( t ) )
        then propagate( ssa_link( t ) )
        lattice( t ) = lattice( t ) ⊓ lattice( ssa_link( t ) )
    endif
    if unvisited ( left ( t ) ) then propagate( left( t ) )
    if unvisited ( right ( t ) ) then propagate( right( t ) )
    case on type (t)
        constant C: lattice( t ) = C
        arithmetic operation:
            if all operands have constant lattice value
            then lattice( t ) = arithmetic result of
                lattice values of operands
            else lattice( t ) = ⊥
            endif
        store: lattice( t ) = lattice( RHS )
        φ-function: lattice( t ) = ⊓ of φ-arguments of t
        γ-function:
            if lattice( predicate ) = C then
                lattice( t ) = lattice value of
                γ-argument corresponding to C
            else lattice( t ) = ⊓ of all γ-arguments of t
            endif
        μ-function: lattice( t ) = ⊥
        η-function: lattice( t ) = lattice ( η-argument )
        default: lattice( t ) = ⊥
    end case
end propagate
```

Figure 7: Demand-driven propagation with conditional constants.

Future Work and Conclusions

We plan many extensions to this work. We have preliminary data on the number of intraprocedural constants found in scientific Fortran programs [9], and are extending this work to include interprocedural analysis and procedure integration. Although some work has already been done in this area, we are interested in applying our demand-driven style to the problem.

Other planned projects include extending our work into the area of non-integer and symbolic expression propagation and fully developing dead-code identification and elimination.

We have presented a new demand-driven method for performing conditional constant propagation, which works on sparse data-flow graphs, finds the same class of constants as previous algorithms and avoids evaluating expressions more than once. We have detailed specific algorithms to accomplish this task, and are currently performing a comparative study with other algorithms, a necessary step to measure the efficiency of our approach.

Corresponding author: Eric Stoltz
Oregon Graduate Institute of Science & Technology
P.O. Box 91000
Portland, Oregon 97291-1000
e-mail: stoltz@cse.ogi.edu

References

[1] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pp. 257–271, June 1990.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing Static Single Assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[3] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.

[4] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *submitted for publication*, Sept. 1993.

[5] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 90–99, June 1993.

[6] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 78–89, June 1993.

[7] G. A. Kildall. A unified approach to global program optimization. In *Conference Record of the First ACM Symposium on Principles of Programming Languages,* pp. 194–206, Oct. 1973.

[8] E. Stoltz, M. P. Gerlek, and M. Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. In *1994 ACM Conf. Proceedings Hawaii International Conference on System Sciences*, Jan. 1994. *to appear.*

[9] E. Stoltz, M. Wolfe, and M. P. Gerlek. Demand-driven constant propagation. Technical Report 93-023, Oregon Graduate Institute, 1993.

[10] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, July 1991.

[11] M. Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 162–174, June 1992.