Constant Propagation with Conditional Branches

Mark N. Wegman
Frank Kenneth Zadeck

IBM T. J. Watson Research Center
Yorktown Heights., New York 10598

1.0 Introduction

Constant propagation is a well known global flow analysis problem. The goal of constant propagation is to discover values that are constant on all possible executions of a program and to propagate these constant values as far forward through the program as possible. Expressions that have all constant operands can be evaluated at compile time and the results further propagated. The use of the algorithms presented here can result in smaller and faster compiled programs.

While the constant propagation problem is easily shown to be undecidable in general (for example see Kam and Ullman [KaUl76]), there are many reasonable subsets of the problem that are decidable and have computationally efficient algorithms. We presents four such algorithms in this paper. Each algorithm presented here is *conservative* in the sense that all constants may not be found, but each constant found will be constant over all possible executions of the program. The algorithms are presented in order of increasing power; each algorithm finds at least the constants found by the previous algorithm. These algorithms are among the simplest, fastest, and most powerful global constant propagation algorithms known.

The first algorithm is by Kildall [Kild73]. His work was among the first to describe the constant propagation problem and give an algorithmic solution.

The second algorithm is an easily understood reformulation of the Reif and Lewis [ReLe77] algorithm. It is unfortunate that this algorithm is not better known, since it works in time linear to the size of the *DefUseChains* yet computes all the constant values of the algorithm by Kildall.

In the third algorithm, ConditionalDef, a new propagation strategy is presented. This algorithm uses the same input data structures as the algorithm by Reif and Lewis. The new idea presented in ConditionalDef is a technique for propagating the values so that if any conditional branches are found to have a constant conditional expression, the search for constants can ignore parts of the program that are never executed. The algorithm does a form of dead code elimination in combination with constant propagation. The first benefit of this approach is that the algorithm runs faster than the Reif and Lewis algorithm since it does not have to evaluate the sections of the program that are never executed. A second benefit is that values created in the dead areas cannot possibly kill potential constants. More constants can therefore be found by ConditionalDef than the Reif and Lewis algorithm.

The fourth algorithm, ConditionalConstant, uses a new data structure and the propagation technique presented in the ConditionalDef to find a class of constants equivalent to those of Wegbreit [Wegb75]. Wegbreit's algorithm discovers all constants that can be discovered by evaluating all conditional branches with all constant operands. ConditionalConstant finds more constants than either ConditionalDef or Kildall's algorithm. The worst case speedup over Wegbreit's approach is proportional to the number of variables in the program. The expected speedup is proportional to the number of variables in the program times the number of edges in the program flow graph.

1.1 Uses for Constant Propagation Algorithms

Constant propagation techniques serve several purposes in optimizing compilers.

- Expressions that can be evaluated at compile time do not have to be evaluated at execution time. If such expressions are inside loops, a single evaluation at compile time can save many evaluations at execution time.
- Code that is never executed can be deleted. This code, commonly called *Dead Code*, is discovered by determining conditional branches that always take one of the possible branch paths. Since many of the parameters to procedures are constants, constant propagation used with procedure integration can avoid the swelling of code size that often results from naive implementations of procedure integration.
- The detection of paths that are never taken simplifies the control flow of a program. The simplified control structure can aid the transformation of the program into a form suitable for vector processing. (See Furtney and Pratt [FuPr82] and Pratt [Prat78].)
- Constant propagation can be done over a variety of domains. For example one can look at the type fields of values, and do constant propagation over that. The basis of many register allocation algorithms is a technique called *Value Numbering*. Constant propagation can be an integral part of value numbering algorithms. Value Numbering assigns all uninitialized variables symbolic values, and propagates these. This may enable detection of equal values which allows better register allocation since variables with the same value can share a register.

2.0 Preliminary Definitions

2.1 Graph Definitions

Since the performance of an algorithm is usually specified in terms of the size of its input, it is necessary to define some common measures of program size:

N is the number of statements in the program. For notational convenience, each node in the program flow graph contains one expression. This number also closely approximates the number

of definition sites in the program since most statements assign a value.

E is the number of edges in the program flow graph. A good upper bound for E is twice N since conditional statements typically have only two successors.[1]

V is the number of variables in the program.

C is the size of the *DefUseChains*. See sections 4.0 and 8.1 .

Let a program consist of three types of nodes: *conditional nodes, label nodes,* and *assignment nodes.* There is one distinguished node, the *start node.* Conditional and label nodes represent potential deviations in control flow through a program. An expression is evaluated at a conditional node and control is subsequently transferred to a label node; for simplicity, assume that such expressions have no side-effects and that control is transferred by *binary branches.*[2] Assignment nodes represent sites at which variables are defined in terms of other variables and constants; for simplicity, assume that only scalar (i.e. non-subscripted) variables participate in assignment nodes.[3]

2.2 The Lattice for Constant Propagation

The output of a constant propagation algorithm is an *output assignment* of lattice values to variables at each node in the program. Not all variables need be given values at each node. These values will be correct whenever execution enters a node.

Let all variables defined or used within a given assignment or conditional node be characterized by two attributes, *LevelCell* and *ValueCell*, that represent compile time knowledge about the value of such variables at the exits of such nodes. As depicted in Figure 1, LevelCell can assume values from a lattice with three types of elements: the highest element is ***top***, the lowest is ***bottom***, and all elements in the middle are ***constant***. In the lattice theoretic sense, no constant is higher or lower than any other. At the end of constant propagation, a ***constant*** LevelCell at a node means that on all possible executions of the program, the associated variable always has the same value when that node is exited; ValueCell contains the actual value of the constant. ***Bottom*** means that a constant value cannot be guaranteed. ***Top*** means that the variable may be some (as yet) undetermined constant. Upon termination of a constant propagation algorithm, all LevelCells are either ***constant*** or ***bottom***.[4]

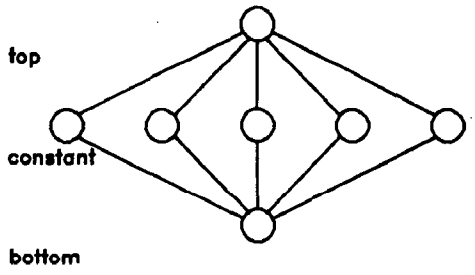

Figure 1. The Three Level Lattice

The algorithms start with the optimistic assumption of assigning ***top*** to the LevelCell of all variables at all nodes except the start node. All LevelCells at the start node are assigned ***bottom***.[5] The algorithms proceed by lowering (in the lattice theoretic sense) the LevelCells of the variables at each node, as more information is discovered. This proceeds until a fixed point is achieved. The additional information is added by applying the meet rules (defined in a later section). Each of the values for variables at a node corresponds to the conditions prior to the execution of the statement. If the node is an assignment, the value of the variable assigned is stored.

2.2.1 Meet Rules

The rules for lowering lattice values are as follows:

a	∩ ***top***	→ a	
a	∩ ***bottom***	→ ***bottom***	
constant	∩ ***constant***	→ ***constant***	(if the values are equal)
constant	∩ ***constant***	→ ***bottom***	(if the values are not equal)

Figure 2. Rules for Meet

2.2.2 Expression Rules

Variables have values at the beginning of nodes and at the exits of nodes. If a variable is not assigned a value, then its value is unchanged by the node. If the node is an assignment, and any of the variables used in the expression portion of it has a value of ***bottom***, the value exiting the statement for that variable is ***bottom***. If all values used in the expression portion are constant, the value of the assigned variable is the value exiting the expression, when evaluated with those constant values. Otherwise the value assigned is ***top***.

3.0 Kildall

In Kildall's algorithm [Kild73], the program flow graph is used for propagation of values. The process of visiting a node involves examination of every variable in the program. Initially a worklist is made up consisting of all nodes immediately following the start node. A node is chosen from the worklist, removed from the worklist, and examined. The lattice values stored at the examined node become the lowest values exiting any preceding node. This may cause the value of an assignment node to differ from the value stored. In that case all nodes following the examined node must be examined, and they are added to the worklist. If no values change, then no nodes are added to the worklist. The process repeats until the worklist is empty.

Used naively, the Kildall's algorithm will find those constants that Kildall calls *Simple Constants*. Simple constants are all values that can be proved to be constant subject to two constraints: no information is assumed about which direction branches will take, and only one value for each variable is maintained along each path in the program.

Since each variable can only have its lattice value lowered twice, each node may be visited at most $2 \times V$ times. The time required for Kildall's algorithm is $E \times V$ node visits, and V operations during each node visit. This results in a running time of $E \times V^2$ in the worst case. The space required is $N \times V$.

When Kildall's algorithm visits a node, it applies a function that maps each variable in the program to each variable in the program. In the following sections we reformulate that notion in order to more closely model what Reif and Lewis did and what we will do. This reformulation of Kildall's algorithm allows Reif and Lewis's algorithm to run faster than Kildall's algorithm. Yet it produces the same information that can be found by applying Kildall's algorithm.

4.0 DefUseChains

Many constant propagation algorithms work on a graph of *DefUseChains*. This data structure is common to optimizing compilers and is described in many textbooks on compilers such as Aho and Ullman [AhUl77]. DefUseChains can be built by many common dataflow analysis techniques such as those by Allen [Alle70], Graham

1 This has been experimentally verified by Allen [ACFF80] and Pratt [Prat78]. Graphs have up to N^2 edges for programs containing computed **goto** statements with all other statements as targets; these are very rare.

2 This is not an unreasonable assumption. Most structured programming constructs give rise to programs with binary branches; two common exceptions are the **case** statement in Pascal and the computed **goto** statement in Fortran. Such statements are easily represented by nested sets of binary branches.

3 See the section on "Areas for Future Research".

4 This will be later restricted to be all LevelCells that are shown to be executable.

5 It is possible to interpret the semantics of a programming language in such a way that it is legitimate to assign ***top*** to the variables at the start node. In some cases, such as when a block of initialization is controlled by a *first-time* flag, this will yield superior results.

and Wegman [GrWe76], Kam and Ullman [KaUl76], Tarjan [Tarj75], Wegman [Wegm83], and Zadeck [Zade84].

A DefUseChain is a connection from a *definition site* for a variable to a *use site* for that variable. This connection must be reachable along the *Program Flow Graph* without passing through another definition site for that variable. Use sites are normally operands of expressions. If as a side effect of computing such an expression a new value is stored, then there is a link between the operands of the expression and the new definition site. This definition can possibly lead to further uses of the variable. Thus, the DefUseChains can be viewed as a graph.

In order to simplify the presentation of the algorithms, we modify the definition of DefUseChains. We add new nodes called *join nodes* to the DefUseChain graph. One node is added for each use site. We divide the DefUseChains into two parts: the first part, the *DefJoinEdge*, starts at the definition site and terminates at a join node. The second part, the *JoinUseEdge*, starts at the join node and terminates at the use site. There can be several DefJoinEdges but only one JoinUseEdge connected to each join node. This division allows us to separate the place where several values for a single variable are joined together from the place where values for several variables come together as the operands of expressions. We will associate with each join node and each expression node a lattice element, i.e. a LevelCell and a ValueCell.

Figure 3 shows a Program Flow Graph and Figure 4 shows the resultant DefUseChain Graph for a simple program. The join nodes are shown as boxes.

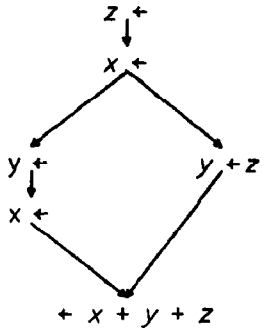

Figure 3. Program Flow Graph

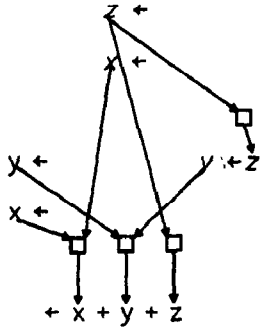

Figure 4. DefUseChains

Variables can be assigned expressions that do not depend on any other variables. Read statements or assignments of constants are examples of these. These distinguished definition sites are called the *RootEdges* because they form the roots of the DefUseChain Graph. Each of these statements also has a LevelCell and a ValueCell that supplies the initial value to replace the non-existent expression.

5.0 Reif and Lewis

The algorithm by Reif and Lewis [ReLe77][ReLe82] finds all Simple Constants. The significance of this algorithm is that it uses a sparse representation to propagate the values through the program. The use of this sparse representation improves the time and space complexity of Kildall's algorithm but still finds all Simple Constants.

Reif and Lewis's algorithm works as follows:[6]

1. A worklist is initialized to contain all RootEdges from the DefUseChain graph. The definition site LevelCells are assigned ***bottom*** if the value of the expression cannot be evaluated, otherwise they are assigned ***constant*** and the ValueCell is initialized to the value of the constant. All other LevelCells are assigned the value ***top***.
2. DefJoinEdges are taken off the worklist. The values at the source of the DefJoinEdge are compared to the values at the destination join node. These comparisons are performed under the rules given in "Meet Rules".
3. If this causes the value in the LevelCell to be lowered, the new values are propagated along the JoinUseEdge to the use site. The expression is recomputed and perhaps lowered by the rules in "Expression Rules". If the new value is lower than the value stored for the expression then all DefJoinEdges with their source at this node are added to the worklist.[7]

5.1 Asymptotic Complexity

The time complexity of this algorithm is proportional to the size of the DefUseChain Graph. The Reif and Lewis algorithm requires that each DefUseChain be visited at least once and at most twice. The visits occur when the value of its definition site is lowered to either ***constant*** or ***bottom***.

One lattice element is required for each join node and expression node in the DefUseChain Graph. The number of join nodes is smaller, but linearly related to the size of the DefUseChain Graph. The size of the DefUseChain Graph is discussed in the section that describes the Global Value Graph.

5.2 Advantages of Reif and Lewis

The algorithm by Reif and Lewis was not the first algorithm to use a sparse representation for propagation. It was the first to use such a representation and find all Simple Constants. There are many global constant propagation algorithms that are in common use and resemble Reif and Lewis but do not achieve the same results. The constant propagation algorithm described by Ferrante and Ottenstein [FeOt82] is inherently weaker than that done by Reif and Lewis.[8] Variants of the weaker algorithm are used by Horowitz in the Cornell Program Synthesizer Generator[9] and by Fabri in the Experimental Compiler System project at IBM.[10]

In these weaker algorithms, propagation is deferred at join nodes until all edges that reach that node have been visited. If they all have the same value, the join node is given that value. These algorithms start with the assumption that all expressions have ***bottom*** and attempt to raise the lattice value to ***constant*** when it can prove that all values that reach that location are constant. Each DefUseChain is only visited once. The weaker algorithms are less optimistic in their propagation.

---

6 The Reif and Lewis algorithm as originally presented used a Global Value Graph as its sparse representation. The following presentation uses the DefUseChain graph. These two representations produce equivalent results when used with this algorithm. The Global Value Graph data structure is presented later.

7 This implies that the expression may be evaluated twice for every term of the expression. If the expression is large, this can be expensive. Reif and Lewis store the expression as a tree and evaluate the leaves and internal nodes of the tree as their value changes. Another possible method is to have a counter associated with each expression. This counter is initialized to the number of terms in the expression and is decremented each time another term is lowered to value constant. The expression is only computed when the counter reaches zero and no input term has been assigned ***bottom***.

8 Many other algorithms commonly found in the literature are even weaker than this. The algorithm in Aho and Ullman [AhUl78] gives up if any use has more than one definition that reaches it.

9 This work has not been published as of the date of this paper. The knowledge of this work is by private communication.

10 The source of this reference is the internal documentation of the Experimental Compiler System at IBM. The paper was never published due to the untimely death of the author.

They never propagate any value unless they are certain that the value will never invalidated.

The major drawback of this approach is that propagation cannot proceed around cycles in the DefUseChain Graph. These cycles are typically the result of simple loops in the program. The Reif and Lewis algorithm finds more constants than the weaker algorithms since Reif and Lewis can propagate through loops. In Figure 5, the variable *i* always has the value 1 at the bottom of the loop. The weaker algorithms get stuck on the loop and fail to discover this constant.

```
i ← 1
do (...)
 j ← i
 i ← f(...)
 { no stores are done into j here }
 .....
 i ← j
end
```

Figure 5. Simple Program Loop

6.0 ConditionalDef

Kildall's and Reif and Lewis's algorithm find all constants that can be found without taking conditional branches into consideration. Consider the example shown in Figure 6. Neither algorithm is capable of finding this constant since they make no assumptions about the possible directions that branches can take. Since *i* is always 1, the conditional always takes the true branch and *j* is always equal to 1. Such code may be the result of procedure integration or abstract data type compilation.

```
i ← 1
...
if i = 1
  then j ← 1
  else j ← 2
```

Figure 6. Conditional Constant Definition

The goal of ConditionalDef is to find more constants than the Simple Constants found by Kildall or Reif and Lewis. This is accomplished by intelligently looking at the results of conditional branches. Whenever we can assume that a conditional expression is always constant, we assume that the branch that it guards will only go in one direction.

To exploit this knowledge we use a different ordering of the DefJoinEdges. In this new ordering we may not ever include some DefUseChains. Reif and Lewis are able to propagate any DefJoinEdge that did not have the value ***top***. In ConditionalDef, we will defer the evaluation of any DefJoinEdge until the Program Flow graph node that is the source of that DefJoinEdge is marked executable.

Edges are marked executable by symbolically executing the program, beginning with the Start Node. Whenever a node is executed, the nodes in the program flow graph immediately following that node are added to a worklist if evaluation of conditional expression indicates control may pass along those exit edges.

This algorithm is able to ignore any DefJoinEdge whose definition occurs on a program flow graph edge that is never executed. Thus, this algorithm accomplishes a form of *Dead Code Elimination*.[11]

This algorithm uses two worklists. The first, *FlowWorkList*, is a worklist of Program Flow Graph edges and the second, *DefWorkList*, is a worklist of DefJoinEdges.

ConditionalDef works as follows:

1. The FlowWorkList is initialized to have the edge entering the Start Node of the program. The LevelCells and ValueCells for the DefUseChain Graph are initialized as in Reif and Lewis. The DefWorkList is initially empty.

   Each node also has associated with it a flag, the

   ExecutableFlag that allows the expression to be evaluated. This flag is initially false for all nodes.

2. Execution is halted when both worklists become empty. Execution may proceed by processing items from either worklist.

3. If the item is a Program Flow Graph edge from the FlowWorkList the action is to mark the ExecutableFlag true. If the ExecutableFlag had been false, then evaluate the expression according to the rules in "Expression Rules". If the result is not ***top*** do VisitExpression.

4. If the item is a DefJoinEdge from the DefWorkList, the source value is combined with the value at the join node according to the rules in "Meet Rules". If this causes the lattice value to be lowered, the new value is then propagated to the expression. If the ExecutableFlag is true for that expression, the expression is evaluated. If evaluation causes the LevelCell of that expression to be lowered, do VisitExpression.

VisitExpression is defined as follows:

1. If the expression is part of an assignment node, add all DefJoinEdges starting at the definition for that node to the DefWorkList.

2. If the expression controls a conditional branch, some flow graph edges will have to be added to the FlowWorkList. If the LevelCell has value ***bottom***, both exit edges must be added to the FlowWorkList. If the value is ***constant*** only the flow graph edge that is executed as the result of the branch is added to the FlowWorkList.

6.1 Asymptotic Complexity

Each DefUseChain can only be examined twice, as in Kildall and in Reif and Lewis. Nodes in the program flow graph are visited once for each in-edge that they have. Because nodes may have at most two out-edges, the number of edges in the graph is limited to twice the number of nodes. This makes the asymptotic running complexity of this algorithm equal to twice the number of nodes in the flow graph plus twice the number of DefUseChains or $2 \times N + 2 \times C$.

[11] There are two classical techniques both called Dead Code Elimination. The goal of one technique is to eliminate code that can never be executed. This is what is accomplished here. The goal of the other technique is to delete sections of code whose results are never used. (See Allen and Cocke [AlCo72].) We do not do this. Each of these techniques finds a different class of dead code. Neither subsumes the other.

6.2 Significance of Algorithm

The asymptotic complexity of this algorithm is the same as the Reif and Lewis's algorithm. The actual execution time should be better in most cases. This algorithm can skip sections of the program that are inaccessible at execution time. In the compilers for languages such as Ada and PL.8 where procedure integration is performed, or for LISP compilers where macro expansion is performed, this may provide a significant improvement in both compile time and execution time performance. In Figure 7, the *plus* routine is a macro that is expanded by the compiler. At compile time, the conditional expression for the execution time type check can be totally eliminated.

```
(setq i 1)
loop
  (cond ((greaterp i 10) (go out)))
  ...
  (setq i (plus i 1))
  (go loop)
out

{plus is a macro which expands in line}
{to the following.}
{It replaces x with the first argument}
{to the macro and y with the second.}
  (cond
    ((and (integer x) (integer y))
      (integer__add x y))
      {The result of integer__add is an integer.}
  ...
  )
```

Figure 7. Removal of Conditional Type Check

This algorithm is also useful for propagating constants that are the result of constant parameters in Algol like languages. Allen et.al. [ACFF80] claims that over 24 percent of all parameters to subroutines in PL/I are constants. Scheifler [Sche77] and Ball [Ball79] have considered the effects of procedure integration on optimization. Ball does a form of ad hoc dataflow analysis to estimate the effects of a constant parameter being passed to a procedure. He reports positive results on the size and execution time of the complied code even though his constant propagation technique only discovers the Simple Constants found by Kildall. In many cases, procedure integration followed by conditional constant propagation could substantially improve the quality of code generated.

7.0 Procedure Integration

Even if the parameters are only potentially constant, procedure integration can be combined with constant propagation. A prepass can create the DefUseChains for all procedures.[12] Then it is possible to integrate only those statements which are executable based on constant propagation through the DefUseChains of the procedure. Consider Figure 8. The value of the string s always has value 'abc' at the beginning of the loop. Unless the *concat* and *trunc* routines are integrated these constants can never be discovered. Furthermore, even if it is not desirable to leave the integrated forms expanded inline (because of space constraints), the invariant value of the string s can still be safely guaranteed.

```
s ← 'abc'
do ...
  s ← concat(s, 'de')
  s ← trunc(s, 2)
end

procedure concat(a, b)
  return (a || b)
end concat

procedure trunc(a, i)
  if (length(a) < i)
    then do
      long involved error recovery
    end
    else return (a[1 : length(a) - i])
end trunc
```

Figure 8. Procedure Integration to Find Constants

The advantages of combining procedure integration with constant propagation, as opposed to the more naive approach of integrating and then doing constant propagation, are in the domain time and space saving. If one integrates first, one must expend both time and space to copy parts of the code that are not relevant for that execution. Then these irrelevant parts must be thrown away. In many compilers that perform procedure integration, the construction of the DefUseChains is not done until after the integration. Since this construction process requires time and space proportional to the number of variables times the number of statements, there is a considerable potential gain to be made by breaking the process into small distinct parts.

8.0 ConditionalConstant

The ConditionalDef algorithm is able to find all constants that Reif and Lewis can plus those extra constants that are the result of not applying definitions defined along paths not executable. While this is correct, it is not the best that can be done by taking all branches into account. ConditionalDef fails because even though both a definition site and a use site can be executable, there may be no executable, definition free, path in the flow graph that connects the two sites. When this happens, ConditionalDef unnecessarily propagates the definition site to the use site causing information to be lost. ConditionalConstant works because its DefUseChains do not span paths which are potentially executable at both end points but not in their middle.

Consider the program shown in Figure 9. Statement (b) provides the only possible value for statement (c). this is true since the path through (b) is the only path to (c) (we know this because the condition must always be true). ConditionalDef will not find this because it will apply all DefUseChains that start from (a) and the algorithm has no information that those values are really killed by (b).

```
i ← 1
j ← 2                (a)

if i = 1
  then j ← 3         (b)

k ← j                (c)
```

Figure 9. Constant Not Found by ConditionalDef.

In order to discover more constant values, the DefUseChains structure must be enhanced. To do this we will adopt a new representation which maybe less sparse than DefUseChains. In this new representation all DefUseChains are interrupted at critical sections of the program. By interrupted, we mean that there is a chain from the definition to the critical section and then another one away from the critical section. The

[12] A proper implementation must treat all variables used as procedure parameters as both definition sites and use sites.

one away from the critical section may be to either another critical section or to the use site.

The birthpoints of Reif and Tarjan (defined in the next section) turn out to identify the critical sections. In ConditionalConstant we modify the Global Value Graph of Reif and Tarjan so that it can be used with ConditionalDef and compute all possible Conditional Constants.

8.1 The Global Value Graph

Reif and Tarjan [ReTa81] have developed a more sophisticated variant of DefUseChains called a *Global Value Graph*. In the Global Value Graph, chains are built from each *birthpoint* of the variable to each reachable use site for that variable.

Definition: Birthpoints for a variable, v, are defined as follows:

- Each definition site for v is a birthpoint.
- Let n be a node with two or more in-edges. If there is a node m which is a birthpoint for v and there is a birthpoint free path from m to n along one in-edge to n but not along the other in-edges to n, then n is a birthpoint for v. A path from m to n is a *birthpoint free path* if the path traverses no nodes that are birthpoints for v.

In DefUseChains, there are many definitions that can reach a use. In the Global Value Graph, only one birthpoint can reach a use. This observation is critical to the Value Numbering problem and this was the reason Reif and Lewis introduced birthpoints.

The Global Value Graph has an advantage that the worst case size grows with $E \times V$ rather than $E^2 \times V$ for the DefUseChains Graph.

It is possible to construct a program where the size of the DefUseChains graph is $E^2 \times V$. An example of a program giving rise to this behavior is shown in Figure 10 through Figure 12. In this example, each of several definition sites for each variable reaches each use site for each for each variable. This behavior does not happen for the Global Value Graph since the join node is a birthpoint.

```
select j
  when a do
    i ← 1
  end
  when b do
    i ← 2
  end
  when c do
    i ← 3
  end
end

select k
  when a do
    a ← i
  end
  when b do
    b ← i
  end
  when c do
    c ← i
  end
end
```

Figure 10. Worst Case Behavior of DefUseChains

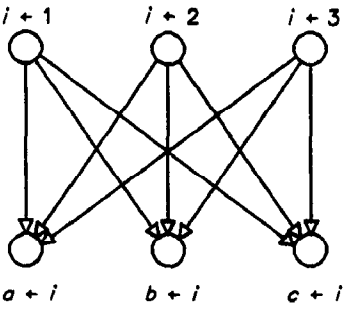

Figure 11. DefUseChains for Previous Program

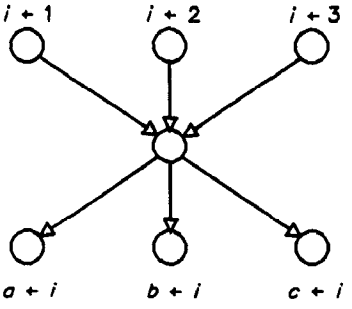

Figure 12. Global Value Graph for Previous Program

8.2 ConditionalConstant Algorithm

The Global Value Graph itself is not enough to solve the problems of the ConditionalDef algorithm. In the Global Value Graph, birthpoints are placed at each node in the Program Flow Graph where different information for a variable enters that node from different in-edges. We must augment this representation so that we can identify which of those in-edges through which execution can actually pass. This identification is accomplished by adding an identity assignment to each node that is an immediate predecessor to a birthpoint node, when that birthpoint is not a definition site. An *identity assignment* for the variable i is an assignment statement of the form $i \leftarrow i$. Thus all the chains that enter the node along an edge can only be propagated to the birthpoint if that in-edge is executable.

Conditional Constant works as follows:

1. Find the birthpoints for the program using the algorithm given by Reif and Tarjan.
2. At every node that is a birthpoint that is not also a definition site, insert an identity assignment along each edge into that node. An identity assignment should also be added at the birthpoint. This will cause a new node to be added to the graph for each identity assignment added.
3. Build a DefUseChain Graph over the resulting program.
4. Use the propagation technique presented in the ConditionalDef algorithm to propagate the constants over this graph.

8.3 Asymptotic Complexity

ConditionalConstant has the same asymptotic time and space complexity as the ConditionalDef algorithm. The average case may be somewhat larger because the average size of the Global Value Graph may be somewhat larger than that of DefUseChains.

This algorithm is equivalent in power to an algorithm by Wegbreit [Wegb75]. Wegbreit's algorithm is a variation of the algorithm by Kildall that accounts for conditional branches. Wegbreit's algorithm requires $E^2 \times V$ operations since the entire set of variables must be examined to assign ***bottom*** to a single variable.

9.0 Theorems and Propositions

In this section we will first show that the algorithm is conservative. By this we mean that the algorithm does not label variables, which are not constant, as constant. We then show that the algorithm is at least as powerful as Wegbreit's, and finds all the constants that his algorithm does.

The first observation in showing that the algorithm is conservative is to note that ConditionalConstant is derived from ConditionalDef by inserting some identity assignments, and then running ConditionalDef. Therefore, since adding identity assignments cannot change the semantics of a program we need only prove that ConditionalDef is conservative.

Before we define conservative more formally, we need the concept of an executable sequence.

Definition: An *executable sequence* is a sequence of tuples, where each tuple consists of a node in the flow graph and a lattice element for each variable. The first tuple contains the start node, and each subsequent tuple is derived by evaluating the expression at the node and changing values as appropriate. By deriving values of expressions, the next tuple may be determined.

Definition: An output assignment is conservative if the values of variables at each node are no higher in the lattice theoretic sense than the values when that node is reached on any possible executable sequence.[13]

Theorem: The output assignment derived from ConditionalDef is conservative and an execution sequence visits only those nodes ConditionalDef labels as executable.

Proof: Suppose to the contrary. Then there must be a shortest execution sequence which either visits a node not labelled executable or which has a value lower than the output assignment to a variable at that node. If a node not labelled executable, is on the path, it must be the first such node or else there is a shorter such path. The preceding node on the path must be a conditional, and the values of the variables used in the condition must be different in the sequence than those in the output assignment, otherwise the node would have been labelled executable. But then there is a shorter path.

Thus, there must be a value which is larger in the output assignment than on the sequence. That value is used at the last node in the shortest sequence. Therefore there must be a preceding assignment node to the variable, and at the assignment node the values agree with the output of ConditionalDef. There must be a DefUseChain from that assignment node to the last node in the sequence, since there are no intervening assignments nodes. Thus, the value at the last node must be correct. QED.

Now that we have shown that we do not find more constants than is correct, we wish to show that we claim as many constants as other algorithms. We compare our algorithm to Wegbreit's which finds more constants than Kildall's. Wegbreit's algorithm is identical to Kildall's except that Wegbreit's algorithm does not propagate values along branches from conditions, until it shows that the branch may be taken.

Theorem: The output assignment derived by ConditionalConstant gives each variable at each node a value which in the lattice theoretic sense is at least as large Wegbreit's algorithm's output assignment.

Proof: Without loss of generality we may assume that Wegbreit's algorithm is working on the same, modified graph that ConditionalConstant works on. Clearly, throwing in additional nodes with identity assignments does not change the values used by Wegbreit's algorithm.

Suppose to the contrary. Then there is a point in the execution of ConditionalConstant where it assigns the first value which is lower than Wegbreit's value.

Claim: That point cannot be at a node which Wegbreit's algorithm detects as unexecutable.

If it is at an unexecutable node then Wegbreit's algorithm must also detect that a branch can only go in one direction, and ConditionalConstant does not detect this. That implies that Wegbreit's algorithm detects a constant at an expression and that ConditionalConstant doesn't. Moreover, ConditionalConstant must lower the value of the expression before it gets to the unexecutable node, and hence there is an earlier point.

The variable whose value is lowered has a DefUseChain into it, along which the lower value comes. To arrive at a contradiction, all we must show is that there is a path of executable nodes from the definition site to this use site and there are no intervening assignment nodes on this path. We leave it as an exercise to the reader to show that Wegbreit's algorithm will always propagate values along such a path.

Because of the way that identity assignments have been added to the graph, the definition site is either (1) one of the identity assignments preceding a join node or (2) on all paths from the root to the use site the definition site is the last assignment node for that variable.

In case (2) all paths from the start node to the use site must pass through the definition site. Thus, if the use site is executable, then there must be a path from the root to the use site of executable nodes, and the definition site must be on all such paths. Moreover, the definition site must be the last assignment node for that variable on that path. Thus, in case (2) there is an executable path and there is a contradiction.

In case (1) the value at the join node becomes lower in ConditionalConstant than in Wegbreit's algorithms, but the values of the variable at the identity assignments preceding the join node are no lower. The identity assignments are executable in ConditionalConstant only if they are executable in Wegbreit's algorithm (by case 2). ConditionalConstant uses the same meet rules as Wegbreit's algorithm to compute the value at the join, and must be using the same or higher values. The meet rules for lattices assure that the meet of one set of values cannot be lower than the meet of a second set of values unless the first set contains a lower value to begin with. Thus, the result at the join may not be lower under ConditionalConstant. QED

10.0 Optimizations

There are two improvements that apply to all the algorithms given in this paper.

1. It is normally beneficial to propagate anything that is known to be ***bottom*** before propagating anything else. Since nothing is ever examined after it goes to ***bottom***, getting something in one step versus two will improve the performance. This improvement will not effect the asymptotic complexity of any algorithm.
2. Some constants can be inferred from conditional expressions. If the conditional expression test whether i is equal to 3, it can be assumed that i will have the value 3 if the true branch is taken. Thus, we can determine that i is a constant on some branches, even if we cannot determine that i is a constant at its birthpoint. There are many other optimizations similar in nature to this that have been compiled by Allen and Cocke in [AlCo72].

11.0 Areas for Future Research

- *Value Numbering* is a problem that is related to constant propagation. An example of value numbering is:

  > Consider a subroutine which is passed the argument i and which immediately contains the assignment $j \leftarrow i$. After the assignment we know that i and j have the same value, however we don't know what that value is.

  More generally, we can give a symbolic value to i and the other arguments (say k) at the beginning of the routine and compute symbolic values, thus we can later determine that $i + k$ is equal to $j + k$ because we fetch the symbolic value stored in j when we create the symbolic value corresponding to the sum.

[13] This notion is very similar to the notion of a *safe assignment* by Graham and Wegman [GrWe76]. In Graham and Wegman, an executable sequence could be any path, even if the branch conditions could be proven constant. They did not allow the branch taken by the path. Thus, a safe assignment is conservative but a conservative assignment is not necessarily safe.

This can all be done in a straight forward manner. However, to do a complete job involves a number of complications: if you can show that certain values must be equal, then you can determine that certain branches must be taken. Moreover, suppose there is a place in the program where two edges join and the value of a variable, say i cannot be determined. Then we can assign i at that location a new symbolic value and conclude that if in a later assignment $j \leftarrow i$ that the two values are identical. If we merely record that we know nothing about the value of i we will also know nothing about the value of j. Reif solved the problem of creating new symbolic values and propagating those values while only slowing the algorithm down by a factor of Ackerman's inverse. However, he did not take any conditional branches into account. We have reason to believe that by slowing the base algorithm down by a *log* factor that this can also be accounted for.

- In Figure 7, a very simple form of type propagation was performed. In LISP, it is thought to be sufficient to propagate the type of any assignment node forward. The information is killed according to rules that are the same in the constant propagation problem. To get good information in SETL, the problem is somewhat harder. The goal is to infer the type of the object from the way it is used. This problem was originally defined by Tenenbaum [Tene74]. SETL has only one primitive data type to program with, the set. Since sets are rather inefficient to implement, the SETL compiler attempts to pick a more efficient representation of an object based on the way the object is used.

  The problem is bidirectional. The information about the way that a variable is used must be propagated backward as well as forward. Tenenbaum's algorithm for doing this requires alternating forward and backward passes. He does each pass in a method similar to Kildall. It may be possible to use a variation of DefUseChains to represent the propagation space. The chains must be bidirectional. That is, an edge from the use to the definition in addition to an edge from the definition to the use. There are many other details that must be worked out, but it does appear that this is a fairly straightforward extension to the ideas presented here.

- In the *Range Propagation Problem* described by Harrison [Harr77], the goal is to propagate ranges of values in an attempt to fix the upper and lower bounds of variables. This is useful to remove subscript range checking from areas of programs that can be proven safe. This problem differs from simple constant propagation in that the lattice may have an infinite number of levels, rather than just three. There are subsets of this problem in which the number of lattice levels is small. Consider the problem of determining the possible values of a label variable in Fortran. The number of labels in a program is small and fixed. This type of problem should be easily solved by modifications to the algorithms presented here.

- Arrays are difficult for almost any data flow analysis problem. The simple solution that is used in almost all implementations of optimizing compilers is to treat any assignment to an array as an assignment of **bottom** unless that array is always indexed by constants. To do anything more sophisticated, may require a much more expensive symbolic evaluator such as the one given in the previous example.

- It is almost always desirable to integrate a function if all of the parameters are constant and the function references no global or free variables. In the cases where only some parameters are constant, the decision is not so clear. Some benefit can be gained by unrolling loops and recursion if the space and time can be controlled by good heuristics. This problem has been investigated by Wegbreit [Wegb75], Ershov [Ersh77], and Wegman [Wegm81].

- Each algorithm given in this paper has the restriction that only one value for each variable is kept at each join node. If Figure 13, the value of c cannot be determined because separate values are not maintained along each flow graph edge that reaches the expression. It is possible that the modified Global Value graph, combined with a carefully constructed node splitting could solve a large subset of this without the combinatorial explosion of both time and space normally associated with node spitting algorithms.

```
if ...
  then do
    a ← 2
    b ← 3
  end
  else do
    a ← 3
    b ← 2
  end
c ← a + b
```

Figure 13. Node Splitting Example

- In this paper we have managed to combine constant propagation with a form of dead code elimination and procedure integration. One of the open questions in compiler optimization is the proper order to apply the various optimizations. Some optimizations expose opportunities for other optimization techniques. We have eliminated the need to be concerned about the order of the optimizations we have combined and in the process created a more powerful algorithm. It would be interesting to see if other techniques could be integrated in a similar manner.

12.0 Conclusions

The work presented here is based on three fundamental results concerning the constant propagation problem. The first is Kildall's definition of the problem involving the three layered lattice. The second is Reif and Lewis's algorithm involving a sparse representation of propagation space. The last is Wegbreit's algorithm that used the result of conditional operations to improve the class of constants found.

We have added two relevant results of our own. The first result is that a careful ordering of propagation in concert with symbolically executing the conditional expressions can increase the number of constants found with no penalty in time or space. The second result is a new and different way of representing the propagation space that captures the notion of values flowing along program flow graph edges but still remains sparse.

We have used these five results to craft an algorithm that is efficient in both time and space, and yet finds a very broad class of constants.

13.0 Acknowledgements

We would like to thank our colleagues at Yorktown who originally investigated many of the problems which we addressed. As well, we would like to thank our colleagues who have made many suggestions on the paper itself. We would also like to thank John Reif who made a special trip to Yorktown to help us understand his work.

Bibliography

[ACFF80] Allen, F. E., Carter, J. L., Fabri, J., Ferrante, J., Harrison, W. H., Loewner, P. G., Trevillyan, L. H., The Experimental Compiling System. *IBM Journal of Research and Development*, Nov. 1980, vol. 24, no. 6, page 695-715.

[AhUl77] Aho, A. V., Ullman, J. D., *Principles of Compiler Design*, Addison-Wesley Publishing Company, 1977.

[AlCo72] Allen, F. E., Cocke, J., A catalogue of optimizing transformations. *Design and Optimization of Compilers*, Rustin, R. (Ed.), Printice-Hall, Englewood Cliffs, N. J., July 1972. This was also published as IBM Research Report No. RC3548.

[Alle70] Allen, F. E., Control Flow Analysis. *SIGPLAN Notices*, July 1970.

[Ball79] Ball, J. E., Predicting the effects of optimization on a procedure body. *Proceedings of the SIGPLAN Symposium on Compiler Construction*, Aug 1979, page 214-220.

[Ersh77] Ershov, A. P., On the essence of compilation. *IFIP Working Conference on Formal Description of Programming Concepts*, Aug. 1977.

[FeOt82] Ferrante, J., Ottenstein, K. J., A program form based on data dependency in predicate regions., Nov 1982, no. RC-9685.

[FuPr82] Furtney, M., Pratt, T. W., Kernal-control tailoring of sequential programs for parallel execution. *Proceedings of the 1982 International Conference on Parallel Processing*, Aug. 1982, page 245-247.

[GrWe76] Graham, S. L., Wegman, M., A fast and usually linear algorithm for global flow analysis. *Journal of the Association for Computing Machinery*, January 1976, vol. 23, no. 1, page 172-202.

[Harr77] Harrison, W. H., Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, May 1977, vol. SE-3, no. 3, page 243-250.

[KaUl76] Kam, J. B., Ullman, J. D., Global data flow analysis and iterative algorithms. *Journal of the Association for Computing Machinery*, January 1976, vol. 23, no. 1, page 158-171.

[Kild73] Kildall, G. A., A unified approach to global program optimization. *Conference Record of the First ACM Symposium on Principles of Programming Languages*, October 1973, page 194-206.

[Prat78] Pratt, T. W., Program analysis and optimization through kernal-control decomposition. *Acta Informatica Processing*, 1978, vol. 9, page 195-216.

[ReLe77] Reif, J. H.,Lewis, H. R., Symbolic evaluation and the global value graph. *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Jan 1977, page 104-118.

[ReLe82] Reif, J. H., Lewis, H. R., Symbolic evaluation and the global value graph. published by Harvard University, Aiken Computation Laboratory, 1982, no. TR-37-82.

[ReTa81] Reif, J. H., Tarjan, R. E., Symbolic program analysis in almost linear time. *SIAM Journal of Computing*, Feb. 1981, vol. 11, no. 1, page 81-93.

[Sche77] Scheifler, R. W., An analysis of inline substitution for a structured programming language. *Communications of the ACM*, Sept. 1977, vol. 20, no. 9, page 647-654.

[Tene74] Tenebaum, A. M., Type determination for very high level languages. PhD thesis, published by Courant Institute of Mathematical Sciences, Oct. 1974.

[Wegb75] Wegbreit, B., Property extraction in well-founded property sets. *IEEE Transactions on Software Engineering*, Sept. 1975, volume se-1, number 3, page 270-285.

[Wegm81] Wegman, M., General and Efficent Methods for Global Code Improvement. PhD thesis of the University of California at Berkeley, 1981.

[Wegm83] Wegman, M., Summarizing graphs by regular expressions. *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, January 1983, page 203-216.

[Zade84] Zadeck, F. K., Incremental dataflow analysis in a structure program editor. *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, June 1984, vol. 19, no. 6, page 132-143.