Enabling Sparse Constant Propagation of Array Elements via Array SSA Form

Vivek Sarkar[1] and Kathleen Knobe[2]

[1] IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
vivek@watson.ibm.com

[2] Compaq Cambridge Research Laboratory
One Kendall Square, Building 700, Cambridge, MA 02139, USA
knobe@crl.dec.com

Abstract. We present a new static analysis technique based on *Array SSA form* [6]. Compared to traditional SSA form, the key enhancement in Array SSA form is that it deals with arrays at the element level instead of as monolithic objects. In addition, Array SSA form improves the ϕ function used for merging scalar or array variables in traditional SSA form. The computation of a ϕ function in traditional SSA form depends on the program's control flow in addition to the arguments of the ϕ function. Our improved ϕ function (referred to as a Φ function) includes the relevant control flow information explicitly as arguments through auxiliary variables that are called *@ variables*.
The @ variables and Φ functions were originally introduced as run-time computations in Array SSA form. In this paper, we use the element-level Φ functions in Array SSA form for enhanced static analysis. We use Array SSA form to extend past algorithms for Sparse Constant propagation (SC) and Sparse Conditional Constant propagation (SCC) by enabling constant propagation through array elements. In addition, our formulation of array constant propagation as a set of data flow equations enables integration with other analysis algorithms that are based on data flow equations.

Keywords: static single assignment (SSA) form, constant propagation, conditional constant propagation, Array SSA form, unreachable code elimination.

1 Introduction

The problems of *constant propagation* and *conditional constant propagation* (a combination of constant propagation and unreachable code elimination) have been studied for several years. However, past algorithms limited their attention to constant propagation of scalar variables only. In this paper, we introduce efficient new algorithms that perform constant propagation and conditional constant propagation through both scalar and array references.

One motivation for constant propagation of array variables is in optimization of scientific programs in which certain array elements can be identified as constant. For example, the SPEC95fp [3] benchmark 107.mgrid contains an array variable `A` that is initialized to four constant-valued elements as shown in Fig. 1. A significant amount of the time in this application is spent in the triply nested loop shown at the bottom of Fig. 1. Since constant propagation can determine that `A(2)` equals zero in the loop, an effective optimization is to eliminate the entire multiplicand of `A(2)` in the loop nest. Doing so eliminates 11 of the 23 floating-point additions in the loop nest thus leading to a significant speedup.

Another motivation is in analysis and optimization of field accesses of structure variables or objects in object-oriented languages such as Java and C++. A structure can be viewed as a fixed-size array, and a read/write operation of a structure field can be viewed as a read/write operation of an array element through a subscript that is a compile-time constant. This approach is more compact than an approach in which each field of a structure is modeled as a separate scalar variable. This technique for modeling structures as arrays directly extends to nested arrays and structures. For example, an array of rank n of some structure type can be modeled as an array of rank $n + 1$. Therefore, the constant propagation algorithms presented in this paper can be efficiently applied to structure variables and to arrays of structures. Extending these algorithms to analyze programs containing pointer aliasing is a subject for future research, however.

The best known algorithms for sparse constant propagation of scalar variables [8,2] are based on static single assignment (SSA) form [4]. However, traditional SSA form views arrays as monolithic objects, which is an inadequate view for analyzing and optimizing programs that contain reads and writes of individual array elements. In past work, we introduced Array SSA form [6] to address this deficiency. The primary application of Array SSA form in [6] was to enable parallelization of loops not previously parallelizable by making Array SSA form manifest at run-time. In this paper, we use Array SSA form as a basis for static analysis, which means that the Array SSA form structures can be removed after the program properties of interest have been discovered.

Array SSA form has two distinct advantages over traditional SSA form. First, the ϕ operator in traditional SSA form is not a pure function and returns different values for the same arguments depending on the control flow path that was taken. In contrast, the corresponding Φ operator in Array SSA form includes *@ variables* as extra arguments to capture the control information required *i.e.,* $x_3 := \phi(x_2, x_1)$ in traditional SSA form becomes $x_3 := \Phi(x_2, @x_2, x_1, @x_1)$ in

```
! Initialization of array A
REAL*8 A(0:3)
. . .
A(0) = -8.0D0/3.0D0
A(1) =  0.0D0
A(2) =  1.0D0/6.0D0
A(3) =  1.0D0/12.0D0

. . .

! Computation loop in subroutine RESID()

do i3 = 2, n-1
   do i2 = 2, n-1
      do i1 = 2, n-1
         R(i1,i2,i3)=V(i1,i2,i3)
         -A(0)*( U(i1,  i2,  i3  ) )
         -A(1)*( U(i1-1,i2,  i3  ) + U(i1+1,i2,  i3  )
                +  U(i1,  i2-1,i3  ) + U(i1,  i2+1,i3  )
                +  U(i1,  i2,  i3-1) + U(i1,  i2,  i3+1) )
         -A(2)*( U(i1-1,i2-1,i3  ) + U(i1+1,i2-1,i3  )
                +  U(i1-1,i2+1,i3  ) + U(i1+1,i2+1,i3  )
                +  U(i1,  i2-1,i3-1) + U(i1,  i2+1,i3-1)
                +  U(i1,  i2-1,i3+1) + U(i1,  i2+1,i3+1)
                +  U(i1-1,i2,  i3-1) + U(i1-1,i2,  i3+1)
                +  U(i1+1,i2,  i3-1) + U(i1+1,i2,  i3+1) )
         -A(3)*( U(i1-1,i2-1,i3-1) + U(i1+1,i2-1,i3-1)
                +  U(i1-1,i2+1,i3-1) + U(i1+1,i2+1,i3-1)
                +  U(i1-1,i2-1,i3+1) + U(i1+1,i2-1,i3+1)
                +  U(i1-1,i2+1,i3+1) + U(i1+1,i2+1,i3+1) )
      end do
   end do
end do
```

Fig. 1. Code fragments from the SPEC95fp 107.mgrid benchmark

Array SSA form. Second, Array SSA form operates on arrays at the element level rather than as monolithic objects. In particular, a Φ operator in Array SSA form, $A_3 := \Phi(A_2, @A_2, A_1, @A_1)$, represents an *element-level merge* of A_2 and A_1.

Both advantages of Array SSA form are significant for static analysis. The fact that Array SSA form operates at the element-level facilitates transfer of statically derived information across references to array elements. The fact that the Φ is a known pure function facilitates optimization and simplification of the Φ operations.

For convenience, we assume that all array operations in the input program are expressed as reads and writes of individual array elements. The extension to more complex array operations (*e.g.,* as in Fortran 90 array language) is straightforward, and omitted for the sake of brevity. Also, for simplicity, we will restrict constant propagation of array variables to cases in which both the subscript and the value of an array definition are constant *e.g.,* our algorithm mights recognize that a definition $A[k] := i$ is really $A[2] := 99$ and propagate this constant into a use of $A[2]$. The algorithms presented in this paper will not consider a definition to be constant if its subscript has a non-constant value *e.g.,* $A[m] := 99$ where m is not a constant. Performing constant propagation for such references (*e.g.,* propagating 99 into a use of $A[m]$ when legal to do so) is a subject of future work.

The rest of the paper is organized as follows. Section 2 reviews the Array SSA form introduced in [6]. Section 3 presents our extension to the Sparse Constant propagation (SC) algorithm from [8] that enables constant propagation through array elements. It describes how lattice values can be computed for array variables and Φ functions. Section 4 presents our extension to the Sparse Conditional Constant propagation (SCC) algorithm from [8] that enables constant propagation through array elements in conjunction with unreachable code elimination. Section 5 discusses related work, and Sect. 6 contains our conclusions.

2 Array SSA Form

In this section, we describe the Array SSA form introduced in [6]. The goal of Array SSA form is to provide the same benefits for arrays that traditional SSA provides for scalars but, as we will see, it has advantages over traditional SSA form for scalars as well. We first describe its use for scalar variables and then its use for array variables.

The salient properties of traditional SSA form are as follows:

1. Each definition is assigned a unique name.
2. At certain points in the program, new names are generated which combine the results from several definitions. This combining is performed by a ϕ function which determines which of several values to use, based on the flow path traversed.
3. Each use refers to exactly one name generated from either of the two rules above.

For example, traditional SSA form converts the code in Fig. 2 to that in Fig. 3. The $S_3 := \phi(S_1, S_2)$ statement defines S_3 as a new name that represents the merge of definitions S_1 and S_2. It is important to note that the ϕ function in traditional SSA form is not a pure function of S_1 and S_2 because its value depends on the path taken through the `if` statement. Notice that this path is unknown until runtime and may vary with each dynamic execution of this code.

In contrast to traditional SSA form, the semantics of a Φ function is defined to be a pure function in our Array SSA form. This is accomplished by introducing *@ variables* (pronounced “at variables”), and by rewriting a ϕ function in traditional SSA form such as $\phi(S_1, S_2)$ as a new kind of Φ function,

$\Phi(S_1, @S_1, S_2, @S_2)$. For each static definition S_k, its @ variable $@S_k$ identifies the most recent "time" *at* which S_k was modified by this definition.

For an *acyclic* control flow graph, a static definition S_k may execute either zero times or one time. These two cases can be simply encoded as $@S_k = \text{FALSE}$ and $@S_k = \text{TRUE}$ to indicate whether or not definition S_k was executed. For a control flow graph with *cycles* (loops), a static definition S_k may execute an arbitrary number of times. In general, we need more detailed information for the $@S_k = \text{TRUE}$ case so as to distinguish among different dynamic execution instances of static definition S_k. Therefore, $@S_k$ is set to contain the dynamic *iteration vector* at which the static definition S_k was last executed.

The *iteration vector* of a static definition S_k identifies a single iteration in the iteration space of the set of loops that enclose the definition. Let n be the number of loops that enclose a given definition. For convenience, we treat the outermost region of acyclic control flow in a procedure as a dummy outermost loop with a single iteration. Therefore $n \geq 1$ for each definition. A single point in the iteration space is specified by the iteration vector $\boldsymbol{i} = (i_1, \ldots, i_n)$, which is an n-tuple of iteration numbers one for each enclosing loop. We do not require that the surrounding loops be structured counted loops (*i.e.,* like Fortran DO loops) or that the surrounding loops be tightly nested. Our only assumption is that all loops are single-entry, or equivalently, that the control flow graph is *reducible* [5, 1]. For single-entry loops, we know that each def executes at most once in a given iteration of its surrounding loops. All structured loops (e.g., `do`, `while`, `repeat-until`) are single-entry even when they contain multiple exits; also, most unstructured loops (built out of `goto` statements) found in real programs are single-entry as well. A multiple-entry loop can be transformed into multiple single-entry loops by *node splitting* [5,1].

Array SSA form can be used either at run-time as discussed in [6] or for static analysis, as in the constant propagation algorithms presented in this paper. In this section, we explain the meaning of @ variables as if they are computed at run-time. We assume that all @ variables, $@S_k$, are initialized to the empty vector, $@S_k := (\ )$, at the start of program execution. For each real (non-Φ) definition, S_k, we assume that a statement of the form $@S_k := \boldsymbol{i}$ is inserted immediately after definition S_k[1], where $\boldsymbol{i}$ is the current iteration vector for all loops that surround S_k. Each Φ definition also has an associated @ variable. Its semantics will be defined shortly. All @ variables are initialized to the empty vector because the empty vector is the identity element for a lexicographic max operation *i.e.,* $\max((\ ), \boldsymbol{i}) = \boldsymbol{i}$, for any @ variable value $\boldsymbol{i}$.

As a simple example, Fig. 4 shows the Array SSA form for the program in Fig. 2. Note that @ variables $@S_1$ and $@S_2$ are explicit arguments of the Φ function. In this example of acyclic code, there are only two possible values

[1] It may appear that the @ variables do not satisfy the static single assignment property because each $@S_k$ variable has two static definitions, one in the initialization and one at the real definition of S_k. However, the initialization def is executed only once at the start of program execution and can be treated as a special-case initial value rather than as a separate definition.

for each @ variable — the empty vector ( ) and the unit vector (1) — which correspond to FALSE and TRUE respectively.

Figure 5 shows an example for-loop and its conversion to Array SSA form. Because of the presence of a loop, the set of possible values for an @ variable becomes unbounded *e.g.,* we may have $@S_1 = (100)$ on exit from the loop. However, $@S_1$ and $@S_2$ are still explicit arguments of the Φ function, and their iteration vector values are necessary for evaluating the Φ function at run-time.

The semantics of a Φ function can now be specified by a conditional expression that is a pure function of the arguments of the Φ. For example, the semantics of the Φ function, $S_3 := \Phi(S_2, @S_2, S_1, @S_1)$ in Fig. 5, can be expressed as a conditional expression as follows (where $\succeq$ denotes a lexicographic greater-than-or-equal comparison of iteration vectors):

$$S_3 = \begin{array}{ll} \textbf{if} & @S_2 \succeq @S_1 \ \textbf{then}\ S_2 \\ \textbf{else} & S_1 \\ \textbf{end if} & \end{array}$$

Following each Φ def, $S_3 := \Phi(S_2, @S_2, S_1, @S_1)$, there is the definition of the associated @ variable, $@S_3 = \max(@S_2, @S_1)$, where max represents a *lexicographic maximum* operation of iteration vector values $@S_2, @S_1$.

Consider, for example, a condition C in Fig. 5 that checks if the value of i is even. In this case, definition S_1 is executed in every iteration and definition S_2 is executed only in iterations $2, 4, 6, \ldots$. For this "even-value" branch condition, the final values of $@S_1$ and $@S_2$ are both equal to (100) if $m = 100$. Since these values satisfy the condition $@S_2 \succeq @S_1$, the conditional expression will yield $S_3 = S_2$.

Consider another execution of the for-loop in Fig. 5 in which condition C evaluates to FALSE in each iteration of the for loop. For this execution, the final values of $@S_2$ and $@S_1$ will be the empty vector ( ) and (100) respectively. Therefore, $S_2 \prec S_1$, and the conditional expression for the Φ function will yield $S_3 = S_1$ for this execution.

The above description outlines how @ variables and Φ functions can be computed at run-time. However, if Array SSA form is used for static analysis, then no run-time overhead is incurred due to the @ variables and Φ functions. Instead, the @ variables and Φ functions are inserted in the compiler intermediate representation prior to analysis, and then removed after the program properties of interest have been discovered by static analysis.

We now describe Array SSA form for array variables. Figure 6 shows an example program with an array variable, and the conversion of the program to Array SSA form as defined in [6]. The key differences between Array SSA form for array variables and Array SSA form for scalar variables are as follows:

1. **Array-valued @ variables:**
   The @ variable is an array of the same shape as the array variable with which it is associated, and each element of an @ array is initialized to the empty vector. For example, the statement $@A_1[k_1] := (1)$ is inserted after

```
if $(C)$ then
    $S := \ldots$
else
    $S := \ldots$
end if
```

Fig. 2. Control Flow with Scalar Definitions

```
if $(C)$ then
    $S_1 := \ldots$
else
    $S_2 := \ldots$
end if
$S_3 := \phi(S_1, S_2)$
```

Fig. 3. Traditional SSA form

```
$@S_1 := (\ )$
$@S_2 := (\ )$

if $(C)$ then
    $S_1 := \ldots$
    $@S_1 := (1)$
else
    $S_2 := \ldots$
    $@S_2 := (1)$
end if
$S_3 = \Phi(S_1, @S_1, S_2, @S_2)$
```

Fig. 4. After conversion of Fig. 2 to Array SSA form

Example for-loop:

```
$S := \ldots$
for $i := 1$ to $m$ do
    $S := \ldots$
    if $(C)$ then
        $S := \ldots$
    end if
end for
```

After conversion to Array SSA form:

```
$@S_1 := (\ )$ ; $@S_2 := (\ )$
$S := \ldots$
$@S := (1)$
for $i := 1$ to $m$ do
    $S_0 := \Phi(S_3, @S_3, S, @S)$
    $@S_0 := \max(@S_3, @S)$
    $S_1 := \ldots$
    $@S_1 := (i)$
    if $(C)$ then
        $S_2 := \ldots$
        $@S_2 := (i)$
    end if
    $S_3 := \Phi(S_2, @S_2, S_1, @S_1)$
    $@S_3 := \max(@S_2, @S_2)$
end for
```

Fig. 5. A for-loop and its conversion to Array SSA form

Example program with array variables:

```
n1:   $A[*]$ := initial value of $A$
      $i := 1$
      $C := i < n$
      if $C$ then
n2:     $k := 2 * i$
        $A[k] := i$
        print $A[k]$
      endif
n3:   print $A[2]$
```

After conversion to Array SSA form:

```
n1:   $@i := ()$ ; $@C := ()$ ; $@k := ()$ ; $@A_0[*] := ()$ ; $@A_1[*] := ()$

      $A_0[*]$ := initial value of $A$
      $@A_0[*] := (1)$
      $i := 1$
      $@i := (1)$
      $C := i < n$
      $@C := (1)$
      if $C$ then
n2:     $k := 2 * i$
        $@k := (1)$
        $A_1[k] := i$
        $@A_1[k] := (1)$
        $A_2 := d\Phi(A_1, @A_1, A_0, @A_0)$
        $@A_2 := \max(@A_1, @A_0)$
        print $A_2[k]$
      endif
n4:   $A_3 := \Phi(A_2, @A_2, A_0, @A_0)$
      $@A_3 := \max(@A_2, @A_0)$
      print $A_3[2]$
```

Fig. 6. Example program with an array variable, and its conversion to Array SSA form

statement $A_1[k_1] := i$ in Fig. 6. In general, $@A_1$ can record a separate iteration vector for each element that is assigned by definition A_1. This initialization is only required for @ arrays corresponding to real (non-Φ) definitions. No initialization is required for an @ array for a Φ definition (such as $@A_2$ and $@A_3$ in Fig. 6) because its value is completely determined by other @ arrays.

2. **Array-valued Φ functions:**
A Φ function for array variables returns an array value. For example, consider the Φ definition $A_3 := \Phi(A_2, @A_2, A_0, @A_0)$ in Fig. 6 which represents a merge of arrays A_2 and A_0. The semantics of the Φ function is specified by the following conditional expression for each element, $A_3[j]$:

$$A_3[j] = \begin{array}{ll} \textbf{if} & @A_2[j] \succeq @A_0[j] \ \textbf{then}\ A_2[j] \\ \textbf{else} & A_0[j] \\ \textbf{end if} & \end{array}$$

Note that this conditional expression uses a lexicographic comparison ($\succeq$) of @ values just as in the scalar case.

3. **Definition Φ's:**
The traditional placement of the Φ is at control merge points. We refer to this as a *control* Φ. A special new kind of Φ function is inserted immediately after each original program definition of an array variable that does not completely kill the array value. This *definition* Φ merges the value of the element modified in the definition with the values available immediately prior to the definition. Definition Φ's did not need to be inserted for definitions of scalar variables because a scalar definition completely kills the old value of the variable. We will use the notation $d\Phi$ when we want to distinguish a definition Φ function from a control Φ function.
For example, consider definition A_1 in Fig. 6. The $d\Phi$ function, $A_2 := d\Phi(A_1, @A_1, A_0, @A_0)$ is inserted immediately after the def of A_1 to represent an element-by-element merge of A_1 and A_0. Any subsequent use of the original program variable A (before an intervening def) will now refer to A_2 instead of A_1. The semantics of the $d\Phi$ function is specified by the following conditional expression for each element, $A_2[j]$:

$$A_2[j] = \begin{array}{ll} \textbf{if} & @A_1[j] \succeq @A_0[j] \ \textbf{then}\ A_1[j] \\ \textbf{else} & A_0[j] \\ \textbf{end if} & \end{array}$$

Note that this conditional expression is identical in structure to the conditional expression for the control Φ function in item 2 above.

3 Sparse Constant Propagation for Scalars and Array Elements

We now present our extension to the Sparse Constant propagation (SC) algorithm from [8] that enables constant propagation through array elements.

Section 3.1 contains our definitions of lattice elements for scalar and array variables; the main extension in this section is our modeling of lattice elements for array variables. Section 3.2 outlines our sparse constant propagation algorithm; the main extension in this section is the use of the definition Φ operator in Array SSA form to perform constant propagation through array elements.

3.1 Lattice Values for Scalar and Array Variables

Recall that a lattice consists of:

- $\mathcal{L}$, a set of lattice elements. A lattice element for a program variable v is written as $\mathcal{L}(v)$, and denotes $\textsc{set}(\mathcal{L}(v))$ = a set of possible values for variable v.
- $\top$ ("top") and $\perp$ ("bottom"), two distinguished elements of $\mathcal{L}$.
- A *meet* (or *join*) operator, $\sqcap$, such that for any lattice element e, $e \sqcap \top = e$ and $e \sqcap \perp = \perp$.
- A $\sqsupseteq$ operator such that $e \sqsupseteq f$ if and only if $e \sqcap f = f$, and a $\sqsupset$ operator such that $e \sqsupset f$ if and only if $e \sqsupseteq f$ and $e \neq f$.

The *height* H of lattice $\mathcal{L}$ is the length of the largest sequence of lattice elements $e_1, e_2, \dots, e_H$ such that $e_i \sqsupset e_{i+1}$ for all $1 \leq i < H$.

We use the same approach as in [8] for modeling lattice elements for scalar variables. Given a scalar variable S, the value of $\mathcal{L}(S)$ in our framework can be $\top$, *Constant* or $\perp$. When the value is *Constant* we also maintain the value of the constant. The sets denoted by these lattice elements are $\textsc{set}(\top) = \{\ \}$, $\textsc{set}(Constant) = \{Constant\}$, and $\textsc{set}(\perp) = \mathcal{U}^S$, where $\mathcal{U}^S$ is the universal set of values for variable S.

We now describe how lattice elements for array variables are represented in our framework. Let $\mathcal{U}^A_{ind}$ and $\mathcal{U}^A_{elem}$ be the universal set of *index values* and the universal set of array *element values* respectively for an array variable A in Array SSA form. For an array variable, the set denoted by lattice element $\mathcal{L}(A)$ is a subset of $\mathcal{U}^A_{ind} \times \mathcal{U}^A_{elem}$ *i.e.,* a set of index-element pairs. Since we restrict constant propagation of array variables to cases in which both the subscript and the value of an array definition are constant, there are only three kinds of lattice elements of interest:

1. $\mathcal{L}(A) = \top \quad \Rightarrow \quad \textsc{set}(\mathcal{L}(A)) = \{\ \}$
   This "top" case means that the possible values of A have yet to be determined *i.e.,* the set of possible index-element pairs that have been identified thus far for A is the empty set, $\{\ \}$.
2. $\mathcal{L}(A) = \langle (i_1, e_1), (i_2, e_2), \dots \rangle$
   $\Rightarrow \quad \textsc{set}(\mathcal{L}(A)) = \{(i_1, e_1), (i_2, e_2), \dots\} \cup (\mathcal{U}^A_{ind} - \{i_1, i_2, \dots\}) \times \mathcal{U}^A_{elem}$
   In general, the lattice value for this "constant" case is represented by a finite ordered list of index-element pairs, $\langle (i_1, e_1), (i_2, e_2), \dots \rangle$ where $i_1, e_1, i_2, e_2, \dots$ are all constant. The list is sorted in ascending order of the index values, $i_1, i_2, \dots$, and all the index values assumed to be distinct.

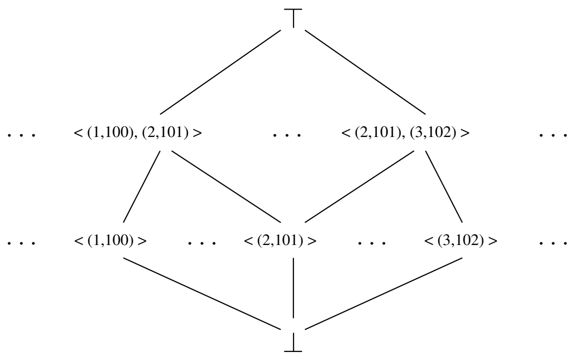


Fig. 7. Lattice elements of array values with maximum list size $Z = 2$

The meaning of this "constant" lattice value is that the current stage of analysis has determined some finite number of constant index-element pairs for array variable A, such that $A[i_1] = e_1$, $A[i_2] = e_2$, ... All other elements of A are assumed to be non-constant. These properties are captured by $\text{SET}(\mathcal{L}(A))$ defined above as the set denoted by lattice value $\mathcal{L}(A)$.

For the sake of efficiency, we will restrict these constant lattice values to ordered lists that are bounded in size by a small constant, $Z \geq 1$ *e.g.,* if $Z = 5$ then all constant lattice values will have ≤ 5 index-element pairs. Doing so ensures that the height of the lattice for array values is at most $(Z + 2)$. If any data flow equation yields a lattice value with $P > Z$ pairs, then this size constraint is obeyed by conservatively dropping any $(P - Z)$ index-element pairs from the ordered list.

Note that the lattice value for a real (non-Φ) definition, will contain at most one index-element pair, since we assumed that an array assignment only modifies a single element. Ordered lists with size > 1 can only appear as the output of Φ functions.

3. $\mathcal{L}(A) = \perp \quad \Rightarrow \quad \text{SET}(\mathcal{L}(A)) = \mathcal{U}^A_{ind} \times \mathcal{U}^A_{elem}$
This "bottom" case means that, according to the approximation in the current stage of analysis, array A may take on any value from the universal set of index-element pairs. Note that $\mathcal{L}(A) = \perp$ is equivalent to an empty ordered list, $\mathcal{L}(A) = \langle\ \rangle$.

The lattice ordering ($\sqsupseteq$) for these elements is determined by the subset relationship among the sets that they denote. The lattice structure for the $Z = 2$ case is shown in Fig. 7. This lattice has four levels. The second level (just below $\top$) contains all possible ordered lists that contain exactly two constant index-element pairs. The third level (just above $\perp$) contains all possible ordered lists that contain a single constant index-element pair. The lattice ordering is determined by the subset relationship among the sets denoted by lattice elements. For example, consider two lattice elements $\mathcal{L}_1 = \langle(1, 100), (2, 101)\rangle$

and $\mathcal{L}_2 = \langle(2, 101)\rangle$. The sets denoted by these lattice elements are:

$$\text{SET}(\mathcal{L}_1) = \{(1, 100), (2, 101)\} \cup (\mathcal{U}_{ind} - \{1, 2\}) \times \mathcal{U}_{elem}$$
$$\text{SET}(\mathcal{L}_2) = \{(2, 101)\} \cup (\mathcal{U}_{ind} - \{2\}) \times \mathcal{U}_{elem}$$

Therefore, $\text{SET}(\mathcal{L}_1)$ is a proper subset of $\text{SET}(\mathcal{L}_2)$ and we have $\mathcal{L}_1 \sqsupset \mathcal{L}_2$ *i.e.,* $\mathcal{L}_1$ is above $\mathcal{L}_2$ in the lattice in Fig. 7.

Finally, the meet operator ($\sqcap$) for two lattice elements, $\mathcal{L}_1$ and $\mathcal{L}_2$, for array variables is defined in Fig. 8 where $\mathcal{L}_1 \cap \mathcal{L}_2$ denotes an intersection of ordered lists $\mathcal{L}_1$ and $\mathcal{L}_2$.

$\mathcal{L}_3 = \mathcal{L}_1 \sqcap \mathcal{L}_2$	$\mathcal{L}_2 = \top$	$\mathcal{L}_2 = \langle(i_1, e_1), \ldots\rangle$	$\mathcal{L}_2 = \bot$
$\mathcal{L}_1 = \top$	$\top$	$\mathcal{L}_2$	$\bot$
$\mathcal{L}_1 = \langle(i'_1, e'_1), \ldots\rangle$	$\mathcal{L}_1$	$\mathcal{L}_1 \cap \mathcal{L}_2$	$\bot$
$\mathcal{L}_1 = \bot$	$\bot$	$\bot$	$\bot$

Fig. 8. Lattice computation for the meet operator, $\mathcal{L}_3 = \mathcal{L}_1 \sqcap \mathcal{L}_2$

3.2 The Algorithm

Recall that the @ variables defined in Sect. 2 were necessary for defining the full execution semantics of Array SSA form. For example, the semantics of a Φ operator, $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$, is defined by the following conditional expression:

$$A_2[j] = \begin{array}{ll} \textbf{if} & @A_1[j] \succeq @A_0[j] \textbf{ then } A_1[j] \\ \textbf{else} & A_0[j] \\ \textbf{end if} & \end{array}$$

The sparse constant propagation algorithm presented in this section is a static analysis that is based on conservative assumptions about runtime behavior. Let us first consider the case when the above Φ operator is a control Φ. Since algorithm in this section does not perform conditional constant propagation, the lattice computation of a control Φ can be simply defined as $\mathcal{L}(A_2) = \mathcal{L}(\Phi(A_1, @A_1, A_0, @A_0) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$ *i.e.,* as a join of the lattice values $\mathcal{L}(A_1)$ and $\mathcal{L}(A_0)$. Therefore, the lattice computation for A_2 does not depend on @ variables $@A_1$ and $@A_0$ for a control Φ operator.

Now, consider the case when the above Φ operator is a definition Φ. The lattice computation for a definition Φ is shown in Fig. 9. Since A_1 corresponds to a definition of a single array element, the ordered list for $\mathcal{L}(A_1)$ can contain at most one pair. The INSERT operation in Fig. 9 is assumed to return a new ordered list obtained by inserting (i', e') into $\langle(i_1, e_1), \ldots\rangle$ with the following adjustments if needed:

– If there exists an index-element pair (i_j, e_j) in $\langle (i_1, e_1), \ldots \rangle$ such that $i' = i_j$, then the INSERT operation just replaces (i_j, e_j) by (i', e').
– If the INSERT operation causes the size of the list to exceed the threshold size Z, then one of the pairs is dropped from the output list so as to satisfy the size constraint.

Interestingly, we again do not need @ variables for the lattice computation in Fig. 9. This is because the ordered list representation for array lattice values already contains all the subscript information of interest, and overlaps with the information that would have been provided by @ variables.

$\mathcal{L}(A_2)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle (i_1, e_1), \ldots \rangle$	$\mathcal{L}(A_0) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\top$	$\top$
$\mathcal{L}(A_1) = \langle (i', e') \rangle$	$\top$	INSERT$((i', e'), \langle (i_1, e_1), \ldots \rangle)$	$\langle (i', e') \rangle$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 9. Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{d\Phi}(\mathcal{L}(A_1), \mathcal{L}(A_0))$

Therefore, we do not need to analyze @ variables for the sparse constant propagation algorithm described in this section, because of our ordered list representation of lattice values for array variables. Instead, we can use a *partial* Array SSA form which is simply Array SSA form with all definitions and uses of @ variables removed, and with ϕ operators instead of Φ operators. If only constant propagation is being performed, then it would be more efficient to only build the partial Array SSA form. However, if other optimizations are being performed that use Array SSA form, then we can build full Array SSA form and simply ignore the @ variables for this particular analysis.

Our running example is shown in Fig. 10. The partial Array SSA form for this example is shown in Fig. 11. The partial Array SSA form does not contain any @ variables since @ variables are not necessary for the level of analysis performed by the constant propagation algorithms in this paper. The data flow equations for this example are shown in Fig. 12. Each assignment in the Array SSA form results in one data flow equation. The numbering S1 through S8 indicates the correspondence.

The argument to these equations are simply the current lattice values of the variables. The lattice operations are specific to the operations within the statement. Figures 13, 9, and 14 show the lattice computations for an assignment to an array element (as in S3 and S5), definition ϕ (as in S4 and S6), a reference to an array element (as in the RHS of S3 and S5). The lattice computation for a ϕ assignment (as in S7) $A_3 = \phi(A_2, A_1)$ is $\mathcal{L}(A_3) = \mathcal{L}_\phi(\mathcal{L}(A_2), \mathcal{L}(A_1)) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_2)$ where $\sqcap$ is shown in Fig. 8. Notice that we also include lattice computation for specific arithmetic computations such as the multiply in S3 and S5. This allows for constant computation as well as constant propagation. Tables for these arithmetic computations are straightforward and are not shown.

The simple data flow algorithm is shown in Fig. 15. Our example includes the propagation of values from a node to two successors (Y) and to a node from two predecessors (D). Equations S1 and S2 are evaluated because they are associated with the entry block. Element 3 of Y_2 is known to have the value 99 at this stage. As a result of the modification of Y_2 both S3 and S5 are inserted into the worklist (they reference the lattice value of Y_2). S3 uses the propagated constant 99 to compute and propagate the constant 198 to element 1 of D_1 and then, after evaluation of S4, to element 1 of D_2. Any subsequent references to D in the `then` block of the source become references D_2 in the Array SSA form and are known to have a constant value at element 1. Depending on the order of computations via the worklist, we may then compute either D_3 and then D_4 or, because D_2 has been modified, we may compute D_5. If we compute D_5 at this point, it appears to have a constant value. Subsequent evaluations D_3 and D_4 cause D_5 to be reevaluated and lowered from constant value to $\perp$ because the value along one path is not constant.

Notice that in this case, the reevaluation of D_5 could have been avoided by choosing an optimal ordering of processing. Processing of programs with cyclic control flow is no more complex but may involve recomputation that can not be removed by optimal reordering. In particular, the loop entry is a control flow merge point since control may enter from the top or come from the loop body. It will contain a ϕ which combines the value entering from the top with that returning after the loop. The lattice values for such a node may require multiple evaluations.

Also notice that in this example, if I in S5 is known to have the value 3, it will be recoded as a constant element. Upon evaluation of S7, the intersection of the sets associated with D_2 and D_4 will not be empty and element 1 of D_5 will be recorded as a constant.

```
$Y[3] := 99$
if $C$ then
    $D[1] := Y[3] * 2$
else
    $D[1] := Y[I] * 2$
endif
$Z := D[1]$
```

Fig. 10. Sparse Constant Propagation Example

```
     Y_0 and D_0 in effect here.
     ...
S1:  $Y_1[3] := 99$
S2:  $Y_2 := \phi(Y_1, Y_0)$
     if $C$ then
S3:    $D_1[1] := Y_2[3] * 2$
S4:    $D_2 := \phi(D_1, D_0)$
     else
S5:    $D_3[1] := Y_2[I] * 2$
S6:    $D_4 := \phi(D_3, D_0)$
     endif
S7:  $D_5 := \phi(D_2, D_4)$
S8:  $Z := D_5[1]$
```

Fig. 11. Array SSA form for the Sparse Constant Propagation Example

S1: $\mathcal{L}(Y_1) = <(3, 99)>$
S2: $\mathcal{L}(Y_2) = \mathcal{L}_{d\phi}(\mathcal{L}(Y_1), \mathcal{L}(Y_0))$
S3: $\mathcal{L}(D_1) = \mathcal{L}_{d[\,]}(\mathcal{L}_*(\mathcal{L}(Y_2[3]), 2))$
S4: $\mathcal{L}(D_2) = \mathcal{L}_{d\phi}(\mathcal{L}(D_1), \mathcal{L}(D_0))$
S5: $\mathcal{L}(D_3) = \mathcal{L}_{d[\,]}(\mathcal{L}_*(\mathcal{L}(Y_2[I]), 2))$
S6: $\mathcal{L}(D_4) = \mathcal{L}_{d\phi}(\mathcal{L}(D_3), \mathcal{L}(D_0))$
S7: $\mathcal{L}(D_5) = \mathcal{L}_{\phi}(\mathcal{L}(D_2), \mathcal{L}(D_4))$
S8: $\mathcal{L}(Z) = \mathcal{L}(D_5[1])$

Fig. 12. Data Flow Equations for the Sparse Constant Propagation Example

$\mathcal{L}(A_1)$	$\mathcal{L}(i) = \top$	$\mathcal{L}(i) = Constant$	$\mathcal{L}(i) = \bot$
$\mathcal{L}(k) = \top$	$\top$	$\top$	$\bot$
$\mathcal{L}(k) = Constant$	$\top$	$\langle(\mathcal{L}(k), \mathcal{L}(i))\rangle$	$\bot$
$\mathcal{L}(k) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 13. Lattice computation for array definition operator, $\mathcal{L}(A_1) = \mathcal{L}_{d[\,]}(\mathcal{L}(k), \mathcal{L}(i))$

$\mathcal{L}(A[k])$	$\mathcal{L}(k) = \top$	$\mathcal{L}(k) = Constant$	$\mathcal{L}(k) = \bot$
$\mathcal{L}(A) = \top$	$\top$	$\top$	$\bot$
$\mathcal{L}(A) = \langle(i_1, e_1), \ldots\rangle$	$\top$	e_j, if $\exists\ (i_j, e_j) \in \mathcal{L}(A)$ with $i_j = \mathcal{L}(k)$ $\bot$, otherwise	$\bot$
$\mathcal{L}(A) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 14. Lattice computation for array reference operator, $\mathcal{L}(A[k]) = \mathcal{L}_{[\,]}(\mathcal{L}(A), \mathcal{L}(k))$

```
Initialization:
    $\mathcal{L}(v) \leftarrow \top$ for all local variables, $v$.
    $insert(E_v, work\_list)$ for each equation $E_v$ defining $v$
          such that $v$ is assigned to in the entry block.

Body:
    while ($work\_list$ != $empty$)
        $E_v \leftarrow remove(work\_list)$
        $reevaluate(E_v)$
        $insert(E'_{v'}, work\_list)$ for each equation $E'_{v'}$ that uses $E_v$
    end while
```

Fig. 15. Algorithm for Sparse Constant propagation (SC) for array and scalar variables

4 Sparse Conditional Constant Propagation

4.1 Lattice Values of Executable Flags for Nodes and Edges

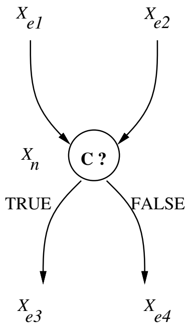
As in the SCC algorithm in [8], our array conditional constant propagation algorithm maintains executable flags associated with each node and each edge in the CFG. Flag X_{ni} indicates whether node n_i may be executed, and X_{ei} indicates whether edge e_i may be traversed. The lattice value of an execution flag is either NO or MAYBE, corresponding to unreachable code and reachable code respectively. The lattice value for an execution flag is initialized to NO, and can be lowered to MAYBE in the course of the constant propagation algorithm. In practice, control dependence identities can be used to reduce the number of executable flag variables in the data flow equations *e.g.,* a single flag can be used for all CFG nodes that are control equivalent. For the sake of simplicity, we ignore such optimizations in this paper.

The executable flag of a node is computed from the executable flags of its incoming edges. The executable flag of an edge is computed from the executable flag of its source node and knowledge of the branch condition variable used to determine the execution path from that node. These executable flag mappings are summarized in Fig. 16 for a node n with two incoming edges, $e1$ and $e2$, and two outgoing edges, $e3$ and $e4$. The first function table in Fig. 16 defines the *join operator* $\sqcap$ on executable flags such that $X_n = X_{e1} \sqcap X_{e2}$. We introduce a *true operator* $\mathcal{L}_\mathcal{T}$ and a *false operator* $\mathcal{L}_\mathcal{F}$ on lattice values such that $X_{e3} = \mathcal{L}_\mathcal{T}(X_n, \mathcal{L}(C))$ and $X_{e4} = \mathcal{L}_\mathcal{F}(X_n, \mathcal{L}(C))$. Complete function tables for the $\mathcal{L}_\mathcal{T}$ and $\mathcal{L}_\mathcal{F}$ operators are also shown in Fig. 16. Note that all three function tables are monotonic with respect to their inputs.

Other cases for mapping a node X_n value to the X_e values of its outgoing edges can be defined similarly. If n has exactly one outgoing edge e, then $X_e =$

Definition of join operator, $X_n = X_{e1} \sqcap X_{e2}$:

X_n	$X_{e2} =$ NO	$X_{e2} =$ MAYBE
$X_{e1} =$ NO	NO	MAYBE
$X_{e1} =$ MAYBE	MAYBE	MAYBE



Definition of true operator for branch condition C, $X_{e3} = \mathcal{L}_{\mathcal{T}}(X_n, \mathcal{L}(C))$:

X_{e3}	$\mathcal{L}(C) = \top$	$\mathcal{L}(C) =$ TRUE	$\mathcal{L}(C) =$ FALSE	$\mathcal{L}(C) = \bot$
$X_n =$ NO	NO	NO	NO	NO
$X_n =$ MAYBE	NO	MAYBE	NO	MAYBE

Definition of false operator for branch condition C, $X_{e4} = \mathcal{L}_{\mathcal{F}}(X_n, \mathcal{L}(C))$:

X_{e4}	$\mathcal{L}(C) = \top$	$\mathcal{L}(C) =$ TRUE	$\mathcal{L}(C) =$ FALSE	$\mathcal{L}(C) = \bot$
$X_n =$ NO	NO	NO	NO	NO
$X_n =$ MAYBE	NO	NO	MAYBE	MAYBE

Fig. 16. Executable flag mappings for join operator ($\sqcap$), true operator ($\mathcal{L}_{\mathcal{T}}$), and false operator ($\mathcal{L}_{\mathcal{F}}$)

$\mathcal{L}(k_3)$	$X_{e2} =$ NO	$X_{e2} =$ MAYBE
$X_{e1} =$ NO	$\top$	$\mathcal{L}(k_2)$
$\boldsymbol{X}_{e1} =$ MAYBE	$\mathcal{L}(k_1)$	$\mathcal{L}(k_1) \sqcap \mathcal{L}(k_2)$

Fig. 17. $k_3 := \Phi(k_1, X_{e1}, k_2, X_{e2})$, where execution flags X_{e1} and X_{e2} control the selection of k_1 and k_2 respectively

X_n. If node n has more than two outgoing edges then the mapping for each edge is similar to the $\mathcal{L}_\mathcal{T}$ and $\mathcal{L}_\mathcal{F}$ operators.

Recall that the $\mathcal{L}_\Phi$ function for a control Φ was defined in Sect. 3.2 by the meet function, $\mathcal{L}_\Phi(\mathcal{L}(A_2), \mathcal{L}(A_1)) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_2)$. For the extended SCC algorithm described in this section, we use the definition of $\mathcal{L}_\Phi$ shown in Fig. 17. This definition uses executable flags X_{e1} and X_{e2}, where $e1$ and $e2$ are the incoming control flow edges for the Φ function. Thus, the lattice values of the executable flags X_{e1} and X_{e2} are used as compile-time approximations of @ variables $@k_1$ and $@k_2$.

4.2 Sparse Conditional Constant Propagation Algorithm

We introduce our algorithm by first explaining how it works for acyclic scalar code. Consider the example program shown in Fig. 18. The basic blocks are labeled $n1$, $n2$, $n3$ and $n4$. Edges $e12$, $e13$, $e24$ and $e34$ connect nodes in the obvious way. The control flow following block $n1$ depends on the value of variable n.

The first step is to transform the example in Fig. 18 to partial Array SSA form (with no @ variables) as shown in Fig. 19. Note that since k had multiple assignments in the original program, a ϕ function is required to compute k_3 as a function of k_1 and k_2.

The second step is to use the partial Array SSA form to create a set of data flow equations on lattice values for use by our constant propagation algorithm. The conversion to equations is performed as follows. There is one equation created for each assignment statement in the program. There is one equation created for each node in the CFG. There is one equation created for each edge in the CFG. The equations for the assignments in our example are shown in Fig. 20. The equations for the nodes and edges in our example are found in Fig. 21 and 22 respectively.

The lattice operations $\mathcal{L}_<$, $\mathcal{L}_*$, and $\mathcal{L}_{max}$ use specific knowledge of their operation as well as the lattice values of their operands to compute resulting lattice operations. For example, $\mathcal{L}_*(\bot, \mathcal{L}(0))$ results in $\mathcal{L}(0)$ because the result of multiplying 0 by any number is 0.

Next, we employ a work list algorithm, shown in Fig. 23, that reevaluates the equations until there are no further changes. A solution to the data flow equations identifies lattice values for each variable in the Array SSA form of the program, and for each node executable flag and edge executable flag in the CFG. Reevaluation of an equation associated with an assignment may cause equations associated with other assignments to be inserted on the work list. If the value appears in a conditional expression, it may cause one of the equations associated with edges to be inserted on the work list. Reevaluation of an edge's executable flag may cause an equation for a destination node's executable flag to be inserted on the work list. If reevaluation of a node's executable flag indicates that the node may be evaluated, then the equations associated with assignments within that node to be added to the work list. When the algorithm terminates, the lattice values for variables identify the constants in the program, and the lattice values for executable flags of nodes identify unreachable code.

```
n1:   $i := 1$
      $C := i \; < \; n$
      if $C$ then
n2:       $k := 2 \; * \; i$
      else
n3:       $k := 2 \; * \; n$
      endif
n4:   print $k$
```

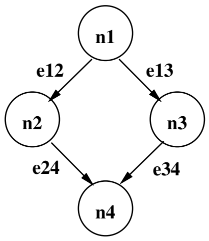


Fig. 18. Acyclic Scalar Example

```
n1:   $i := 1$
      $C := i \; < \; n$
      if $C$ then
n2:       $k_1 := 2 \; * \; i$
      else
n3:       $k_2 := 2 \; * \; n$
      endif
n4:   $k_3 := \phi(k_1, k_2)$
      print $k_3$
```

Fig. 19. Partial Array SSA form for the Acyclic Scalar Example

$$
\begin{aligned}
&\mathcal{L}(i) = \mathcal{L}(1)\\
&\mathcal{L}(C) = \mathcal{L}_{<}(\mathcal{L}(i), L(n))\\
&\mathcal{L}(k_1) = \mathcal{L}_{*}(\mathcal{L}(2), \mathcal{L}(i))\\
&\mathcal{L}(k_2) = \mathcal{L}_{*}(\mathcal{L}\texttt{(2)},\mathcal{L}\texttt{(n)})\\
&\mathcal{L}(k_3) = \mathcal{L}_{\Phi}(\mathcal{L}(k_1), X_{e24}, \mathcal{L}(k_2), X_{e34})
\end{aligned}
$$

Fig. 20. Equations for Assignments

$$
\begin{aligned}
&X_{n1} = \text{TRUE}\\
&X_{n2} = X_{e12}\\
&X_{n3} = X_{e13}\\
&X_{n4} = X_{e24} \sqcap X_{e34}
\end{aligned}
$$

Fig. 21. Equations for Nodes

$$X_{e12} = \mathcal{L}_{\mathcal{T}}(X_{n1}, \mathcal{L}(C))$$
$$X_{e13} = \mathcal{L}_{\mathcal{F}}(X_{n1}, \mathcal{L}(C))$$
$$X_{e24} = X_{n2}$$
$$X_{e34} = X_{n3}$$

Fig. 22. Equations for Edges

```
Initialization:
  $\mathcal{L}(v) \leftarrow \top$ for all local variables, $v$.
  $X_n \leftarrow$ MAYBE where $X_n$ is the executable flag for the entry node.
  $X_n \leftarrow$ NO where $X_n$ is the executable flag for any node other than the entry node.
  $X_e \leftarrow$ NO where $X_e$ is the executable flag for any edge.
  $insert(E_v, work\_list)$ for each equation $E_v$ defining $v$
        such that $v$ is assigned to in the entry block.

Body:
  while $(work\_list$ != $empty)$
     $E_v \leftarrow remove(work\_list)$
     $reevaluate(E_v)$
     $insert(E'_{v'}, work\_list)$ for each equation $E'_{v'}$ that uses $E_v$
     if $E_v$ defines the executable flag for some node $n$ then
        $insert(E'_{v'}, work\_list)$ for each equation $E'_{v'}$ defining $v'$
        such that $v'$ is assigned to in block $n$.
     end if
  end while
```

Fig. 23. Sparse Conditional Constant propagation (SCC) algorithm for scalar and array variables

Even though we assumed an acyclic CFG in the above discussion, the algorithm in Fig. 23 can be used unchanged for performing constant propagation analysis on a CFG that may have cycles. The only difference is that the CFG may now contain back edges. Each back edge will be evaluated when its source node is modified. The evaluation of this back edge may result in the reevaluation of its target node.

As in past work, it is easy to show that the algorithm must take at most $O(Q)$ time, where Q is the number of data flow equations, assuming that the maximum arity of a function is constant and the maximum height of the lattice is constant.

As an example with array variables, Fig. 24 lists the data flow equations for the assignment statements in the Array SSA program in Fig. 6 (the data flow equations for nodes and edges follow the CFG structure as in Figs. 21 and 22). Given the definition of lattice elements for array variables from Sect. 3.1, the conditional constant propagation algorithm in Fig. 23 can also be used unchanged for array variables.

$$\begin{aligned}
\mathcal{L}(A_0) &= \bot \\
\mathcal{L}(i) &= \mathcal{L}(1) \\
\mathcal{L}(C) &= \mathcal{L}_{<}(\mathcal{L}(i), \mathcal{L}(n)) \\
\mathcal{L}(k) &= \mathcal{L}_{*}(\mathcal{L}(2), \mathcal{L}(i)) \\
\mathcal{L}(A_1) &= \mathcal{L}_{d[\,]}(\mathcal{L}(k), \mathcal{L}(i)) \\
\mathcal{L}(A_2) &= \mathcal{L}_{d\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0)) \\
\mathcal{L}(A_3) &= \mathcal{L}_{\Phi}(\mathcal{L}(A_2), X_{e24}, \mathcal{L}(A_0), X_{e14})
\end{aligned}$$

Fig. 24. Equations for Assignments from Fig. 6

5 Related Work

Static single assignment (SSA) form for scalar variables has been a significant advance. It has simplified the design of some optimizations and has made other optimizations more effective. The popularity of SSA form surged after an efficient algorithm for computing SSA form was made available [4]. SSA form is now a standard representation used in modern optimizing compilers in both industry and academia.

However, it has been widely recognized that SSA form is much less effective for array variables than for scalar variables. The approach recommended in [4] is to treat an entire array like a single scalar variable in SSA form. The most serious limitation of this approach is that it lacks precise data flow information on a per-element basis. Array SSA form addresses this limitation by providing Φ functions that can combine array values on a per-element basis. The constant propagation algorithm described in this paper can propagate lattice values through Φ functions in Array SSA form, just like any other operation/function in the input program.

The problem of conditional constant propagation for scalar variables has been studied for several years. Wegbreit [7] provided a general algorithm for solving data flow equations; his algorithm can be used to perform conditional constant propagation and more general combinations of program analyses. However, his algorithm was too slow to be practical for use on large programs. Wegman and Zadeck [8] introduced a Sparse Conditional Constant (SCC) propagation algorithm that is as precise as the conditional constant propagation obtained by Wegbreit's algorithm, but runs faster than Wegbreit's algorithm by a speedup factor that is at least $O(V)$, where V is the number of variables in the program. The improved efficiency of the SCC algorithm made it practical to perform conditional constant propagation on large programs, even in the context of industry-strength product compilers. The main limitation of the SCC algorithm is a conceptual one — the algorithm operates on two “worklists” (one containing edges in the SSA graph and another containing edges from the control flow graph) rather than on data flow equations. The lack of data flow equations makes it hard to combine the algorithm in [8] with other program analyses. The problem of combining different program analyses based on scalar SSA form has been addressed by Click and Cooper in [2], where they present a framework for combining constant propagation, unreachable-code elimination, and value numbering that explicitly uses data flow equations.

Of the conditional constant propagation algorithms mentioned above, our work is most closely related to that of [2] with two significant differences. First, our algorithm performs conditional constant propagation through both scalar and array references, while the algorithm in [2] is limited only to scalar variables. Second, the framework in [2] uses control flow predicates instead of execution flags. It wasn't clear from the description in [2] how their framework deals with predicates that are logical combinations of multiple branch conditions; it appears that they must either allow the possibility of an arbitrary size predicate expression appearing in a data flow equation (which would increase the worst-case execution time complexity of their algorithm) or they must sacrifice precision by working with an approximation of the predicate expression.

6 Conclusions

We have presented a new sparse conditional constant propagation algorithm for scalar and array references based on Array SSA form [6]. Array SSA form has two advantages: It is designed to support analysis of arrays at the element level and it employs a new Φ function that is a pure function of its operands, and can be manipulated by the compiler just like any other operator in the input program.

The original sparse conditional constant propagation algorithm in [8] dealt with control flow and data flow separately by maintaining two distinct work lists. Our algorithm uses a single set of data flow equations and is therefore conceptually simpler. In addition to being simpler, the algorithm presented in this paper is more powerful than its predecessors in that it handles constant propagation through array elements. It is also more effective because its use of data flow equations allows it to be totally integrated with other data flow algorithms, thus making it easier to combine other analyses with conditional constant propagation.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. Cliff Click and Keith D. Cooper. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.
3. The Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. http://open.specbench.org/osg/cpu95/, 1997.

4. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
5. Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, Inc., 1977.
6. Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in Parallelization. *Conf. Rec. Twenty-fifth ACM Symposium on Principles of Programming Languages, San Diego, California*, January 1998.
7. B. Wegbreit. Property Extraction in Well-Founded Property Sets. *IEEE Transactions on Software Engineering*, 1:270–285, 1975.
8. Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.