Weak Dynamic Single Assignment Form

Carl Offner and Kathleen Knobe

November 24, 2003

Abstract

A good intermediate representation is the key to a good compiler, because it determines both how we can think about optimizations and how effective those optimization can be. For example, classical static single assignment (SSA) form is very successful in serial compilers for both these reasons. However, SSA form is not very helpful for the analysis of loop and array based codes for parallel targets.

We present a new Dynamic Single Assignment (DSA) form, designed specifically for analysis and transformations of array and loop-based codes. We present the basic step needed to generate it from serial code.

Future work will address the use of this intermediate representation in the analysis and optimization of parallel code.

Contents

Introduction

This document describes a compiler intermediate form called *weak dynamic single assignment form* (weak DSA) and a sequence of transformation that produce this form from serial code. Programs in this form have the property that each location is assigned only a single value, although it may be assigned that value more than once. The importance of Weak DSA is that it constitutes almost all the work required to convert a program to *strong DSA* form. Programs in strong DSA form have the property that each location is assigned to only once. Strong DSA significantly simplifies analyses and improves optimizations. In particular, existing optimizations applied to a program in strong DSA form are more effective than those same optimizations applied to the program before conversion to strong DSA form. The class of transformations that motivated this work were those that increase the level of parallelism the program exhibits

A variant of weak DSA form was introduced in Knobe (1997). This current report improves on that work in two ways:

1) The use of classic static single assignment (SSA) in Knobe (1997) uncovered some deficiencies in that form. Knobe and Sarkar (1998) addressed these deficiencies in a new form called array SSA form. In particular, the classic SSA form views array updates as modifications to whole arrays. This means that transformations must maintain the order of array updates in the original program. Array SSA form provides a mechanism for managing updates at the element level. Therefore it is not as restricted and allows more aggressive reordering. The weak DSA form presented here uses this new array SSA instead of classic SSA and is therefore more optimizable.

2) This document is significantly more formal and precise than Knobe (1997). It introduces some new concepts (for instance, τ_v variables), gives precise definitions of valid iteration subspaces, valid sets of contributing dimensions, and the rewriting involved in turning an ordinary program into weak DSA form, and uses these definitions to give rigorous proofs of a number of properties, including the correctness of the main algorithm.

Current work not yet completed and not presented here includes optimizations within weak DSA form, conversion to strong DSA form, and conversion of strong DSA to TStreams, a new model of parallel computation.

Chapter 1

Array static single assignment form

1.1 Notation and definitions

1.1.1 Programs

The programs we analyze are programs that can be reduced to a particular highly structured form. Here is an informal description of the grammar of our restricted language:

- An **expression** is an ordinary arithmetic expression having no side effects. It is composed of literal constants, scalars, array references, and pure functions.

  The subscripts of arrays, as well as the arguments of pure functions, are themselves expressions as defined here.

  A reference to the array A may be denoted by $A(s_1, s_2, s_3)$, or by $A(k{:}l, m{:}n, 3)$ when we want to indicate a section.[1] A scalar array reference may simply be denoted by $A(\vec{k})$ when we don't care to specify the rank of the array. In such a case, $\vec{k}$ always refers to a single value, not to a general expression that may take on various values in the course of execution of the program.

- There are three kinds of statements:

  assignment statements The left-hand side of an assignment statement is a scalar or a scalar array reference. The right-hand side is an expression as defined above.

  We occasionally write an assignment statement of the form

  $$A(:,:) = \ldots B(:,:) \ldots$$

  This is just to be interpreted (as in Fortran) as the set of statements that assign—conceptually in parallel—corresponding values on the right-hand side to corresponding elements on the left. That is, the assignments happen as if every element on the right-hand side is evaluated before any element on the left-hand side is assigned.

[1] If $k{:}l$ is the entire extent of array A in the first dimension and $m{:}n$ is the entire extent in the second dimension, then $A(k{:}l, m{:}n, 3)$ can be written simply as $A(:,:,3)$.

output statements Output statements are of the form

print e

where e is an expression as above (in particular, having no side effects).

input statements Input statements are of the form

read R

where R is a scalar or an array reference, as above.

Our language does not include pointers.

- Other constructs (represented by interior nodes in the parse tree) consist of **sequence** nodes, **if-then** and **if-then-else** constructs, and loops, which are either Fortran **do** loops or **while** loops.

  sequence nodes represent non-terminals in the grammar and nodes in the intermediate representation, but "SEQUENCE" is not a keyword in the language. **sequence** nodes occur in three places:

  - The entire program is a **sequence**.
  - The body of a loop is a **sequence**.
  - Each alternative in a conditional construct is a **sequence**.

  Each **while** loop is converted to a **do** loop (with an indeterminate upper bound) by giving it an explicit loop index. A loop index cannot be assigned to within the body of the loop.

  Loop indices are assumed to increase from one iteration of the loop to the next. This is not really a restriction on the language. We can always rewrite the loop if necessary so that this is true. Or we can introduce a new loop index that has this property, making the original loop index a simple function of the new loop index.

We assume that each time a program is executed it has the same input. In particular, this means that the input, if any, to a program must be regarded as part of the program, and similarly for the values of any initialized variables and passed parameters.

For simplicity, we will regard a scalar as a 0-dimensional array. So if we say that A is an array, we include implicitly the possibility that A is actually a scalar.

Such a program can be represented as a tree in the usual manner, with statements as leaf nodes. The interior nodes in the program tree are loop, if-then-else, and sequence nodes.

We don't go into very much detail about the semantics of this language: it is quite straightforward. But we do mention that in a legal program it is inadmissible to refer to the value of a variable that has not been given a value. (Although such variables may be said to have the value $\perp$, this value may not be used in any statement in the program.)

We will consider how to extend the analysis presented here to programs with **goto** statements, programs with subroutines and function calls, and more general, unstructured programs, in future work.

Definition *Two programs are* structurally similar *or* have the same control structure *iff the following three conditions are all satisfied:*

1. *Their parse trees are structurally the same down to the statement level.*

So for instance an assignment statement in one program corresponds to an assignment statement in the other. The expressions in such statements, however, may be quite dissimilar. The next two items, though, place some constraints on such expressions.

2. *Corresponding loop variables have the same names.*

3. *Corresponding statements that assign to array elements (i.e., assignment statements and* **read** *statements) assign to arrays with the same name.*

Note that in particular, no restriction is placed on the values that are assigned—these may differ in the two programs. And as we will see later, arrays with the same name in the two programs may have different shapes. So in particular two programs that are structurally similar may still have very different looking paths of execution, and may compute very different results.

1.1.2 Data space

Each declared object in the program has an associated *data space* $\mathcal{D}$, which is just $\mathbf{Z}^n$ where n is the rank of the data object. So each element of $\mathcal{D}$ is an n-tuple, and we denote the j^{th} index of that n-tuple by d_j. (We could in principle simply denote it by j, but we want to distinguish between data space indices and iteration space indices.) d_j thus corresponds to one of the n standard basis vectors for $\mathcal{D}$, and by an abuse of notation we can identify it with this basis vector. The only subspaces of $\mathcal{D}$ that we are concerned with are those spanned by a subset of these standard basis vectors $\{d_i\}$. Therefore, it is convenient notationally to regard the data space of an object, and all its subspaces, simply as sets of elements $\{d_i\}$. We use this convention for all the vector spaces in this paper.

An array reference R consists of an array A of dimension k and a parenthesized list of k subscript expressions. We define a function σ mapping an array reference R and data space index d_i to the i^{th} subscript expression in R. In practice, R is always clear from the context, and so we omit the reference to R and write $\sigma(d_i)$ instead of $\sigma(R, d_i)$. For example, if R is the array reference $A(2*i-3, 7*j, i+j)$, then $\sigma(d_2) = 7*j$. An expression like $7*j$ is evaluated in the context of a program state, so it is convenient to view an expression like $7*j$ as a function from program states to integers.

Points in data space are represented as tuples delimited by parentheses—this fits in with our notation for array references. For instance, the array reference $A(2, 17, 1)$ refers to the point $(2, 17, 1)$ in the data space of A. The data space point of a k-dimensional array reference A is thus $\big(\sigma(d_1), \sigma(d_2), \ldots, \sigma(d_k)\big)$.

1.1.3 Iteration space

Each data reference also has an *iteration space*. The definition of this space is more subtle than that of data space, because it may differ for different references to the same data object, and also because there are two notions of iteration space that we need to distinguish:

1. Sometimes we want to label assignment statements with the indices of loops containing them, as in this program fragment:

```
do i = 1, n
   do j = 1, m
      a(i, j) = i − j        !! label = [i, j]
   end do
   do j = 1, m
      a(j, i) = i + j        !! label = [i, j]
   end do
end do
```

This enables us to reason about the order of executions of each statement. For instance, in the second assignment statement, $a(2,5)$ is assigned later than $a(3,4)$, because the labels associated with them are $[5,2]$ and $[4,3]$ respectively, and $[5,2]$ follows $[4,3]$ in the ordinary lexicographic order.

That is, we often want a way of distinguishing (and ordering) distinct dynamic references of the same static reference. For this purpose, we define the iteration space at a data reference to be the space of tuples of the indices of the loops nested above that reference. If there are no loops containing the reference, the iteration space contains just one point, which we denote by $\perp$. We call this the *local iteration space* of a data reference. When people talk about "iteration space", this is usually what they mean.

Thus, in the program fragment above, each assignment statement has a local iteration space $\mathbf{Z}^2$. Each local iteration space reflects the order of execution of its statement. Note that the reason the ordering works out correctly is that by our standing assumption, each loop index increases from one iteration of the loop to the next.

Points in local iteration space are denoted by ordered tuples delimited by square brackets, as in the examples $[i,j]$ and $[5,2]$ above.

2. Another kind of iteration space arises from the following consideration: we want to assign labels to each dynamic execution of *every* statement in the program, and use these labels to order all these dynamic statement executions. Figure 1.1 shows a way of doing this for a short and rather meaningless program. (The construction, however, is perfectly general.)

   The figure shows how we can construct a set of labels that reflect the relative order of distinct dynamic references that do not necessarily correspond to the same static reference. (For instance, local iteration space does not enable us to reason about the relative order of an instance of the first assignment statement and an instance of the second assignment statement in the earlier example above.) We call a data structure that enables us to do this *global iteration space*.

   The significant properties of a global iteration space (call it $\mathcal{G}$) are these:

   - $\mathcal{G}$ is a linearly ordered set.
   - Any two distinct dynamic computations of statements in the execution of our program are mapped to distinct elements of $\mathcal{G}$. This mapping preserves the ordering of the computations.

   Let us denote for the moment the subset that is the range of this mapping by $\mathcal{E}$. That is, $\mathcal{E}$ is the set of elements of $\mathcal{G}$ that correspond to dynamic computations in an execution of the program. $\mathcal{E}$ clearly has a first and a last element, corresponding to the dynamic statements executed first and last in the run of the program. And every other element of $\mathcal{E}$ has in $\mathcal{E}$ a next element and a preceding element.

	short form	*long form*
$a(:) = 0$	$[1_\lambda]$	$[1_\lambda, -, -, -, -, -]$
do $i = 1, n$	$[2_\lambda]$	$[2_\lambda, -, -, -, -, -]$
$a(i) = f(i)$	$[2_\lambda,\ i\ , 1_\lambda]$	$[2_\lambda,\ i\ , 1_\lambda, -, -, -]$
if $b(i) > 0$ **then**	$[2_\lambda,\ i\ , 2_\lambda]$	$[2_\lambda,\ i\ , 2_\lambda, -, -, -]$
do $j = 1, n$	$[2_\lambda,\ i\ , 2_\lambda, 1_\lambda]$	$[2_\lambda,\ i\ , 2_\lambda, 1_\lambda, -, -]$
$b(j) = b(i) - a(j)$	$[2_\lambda,\ i\ , 2_\lambda, 1_\lambda,\ j\ , 1_\lambda]$	$[2_\lambda,\ i\ , 2_\lambda, 1_\lambda,\ j\ , 1_\lambda]$
end do		
$a(i) = b(i)$	$[2_\lambda,\ i\ , 2_\lambda, 2_\lambda]$	$[2_\lambda,\ i\ , 2_\lambda, 2_\lambda, -, -]$
else		
$b(i) = a(i)$	$[2_\lambda,\ i\ , 2_\lambda, 1_\lambda]$	$[2_\lambda,\ i\ , 2_\lambda, 1_\lambda, -, -]$
$a(i) = b(i-1)$	$[2_\lambda,\ i\ , 2_\lambda, 2_\lambda]$	$[2_\lambda,\ i\ , 2_\lambda, 2_\lambda, -, -]$
end if		
end do		
print $b(3)$	$[3_\lambda]$	$[3_\lambda, -, -, -, -, -]$

Figure 1.1: Positions in global iteration space of the statements in a short sample program.

We can think of $\mathcal{G}$ as the set of possible times that might occur in the program, and we can think of the subset $\mathcal{E}$ as the set of actual times that do occur as the program is executed. So we call $\mathcal{E}$ the *executed global iteration space*[2].

The actual construction of our global iteration space can be specified like this: it is set of tuples whose rank is the height of the program tree. Associated with each level of the tree is an index. For a sequence, the index denotes the position in that sequence. For loop, it is the loop index. There is also a value (represented below by "$-$") that represents a non-existent index. If any coordinate has the value "$-$", then all subsequent coordinates do. Therefore we can suppress these coordinates when writing a point in global iteration space, as can be seen in the example below. Each legal dynamic statement corresponds to a unique tuple in this global iteration space, and the normal lexicographic order on tuples corresponds to execution order.

This is the construction we will use in this paper. It is natural, because it reflects the loop structure of the program. As in the case of local iteration space, the reason the ordering works out correctly is that by our standing assumption, each loop index increases from one iteration of the loop to the next.

Just as we use ordered tuples delimited by square brackets to denote points in local iteration space, we also use them to denote points in global iteration space.

In this example, we have used the convention that the subscript λ indicates a *lexical* position. So coordinates of a point in global iteration space that are not subscripted by λ refer to points in a local iteration space.

Note that statements on different branches of an **if** construct have identical coordinates. This works because of course in any execution of the program only one branch can be taken, so no point in global iteration space is encountered twice.

[2]The same sort of distinction could be made between possible and executed local iteration spaces, but it does not seem useful to make this distinction.

It is sometimes convenient to have an explicit mapping from global iteration space back into the program. If the program itself is denoted by P, we will use P also to denote the mapping. That is, if t is a point in the global iteration space of the program P, then $P(t)$ denotes the program statement that is dynamically executed at (global iteration) time t. If P is then transformed into a program P' say, with the same global iteration space, we could refer to a corresponding statement $P'(t)$ in the transformed program.

Speaking informally, we use the word "time" to refer to a point in global iteration space. And similarly, we use the phrase "at some point during the course of execution of the program" to refer either to a time t in global iteration space, or to $P(t)$—the dynamic instance of the statement corresponding to that point in global iteration space.

τ_{w} variables and τ_{v} variables (see the section on Array SSA form below) take their values in global iteration space. In many of our examples, we need to represent a point in global iteration space, but the only significant dimensions are the local iteration dimensions. In such a case, we will use the notation

$$[t_1, t_2, t_3]^g$$

to denote the point in global iteration space at the current lexical position whose local iteration coordinates are t_1, t_2, and t_3.

For instance, the point $[2_\lambda, i, 1_\lambda]$ in global iteration space in the example above might have been written as $[i]^g$. Of course if we did this, we would need to have some way of knowing what the lexical coordinates were.

Going in the other direction, if t is a point in global iteration space, we use the notation t^{loc} to denote the corresponding point in the local iteration space at that lexical position. That is, t^{loc} is the tuple consisting of the non-lexical components of t.

$\mathcal{I}$ denotes the local iteration space of a reference. A subspace of the local iteration space is determined by a corresponding set of loop indices.

For convenience, we denote the loop variable corresponding to the iteration space index i_j by $I(i_j)$. For instance in Figure 1.1 at the assignment to $b(j)$, $I(i_1) = i$ and $I(i_2) = j$.

Two programs that are structurally similar have the same global iteration spaces. The *executed* global iteration spaces, however, may well differ—that is, the actual paths of execution may differ. We say that two programs that are structurally the same are *iteratively similar* if the actual paths of execution are the same:

Definition *Two programs P and Q that are structurally similar are iteratively similar iff*

1. *The executed global iteration space of P is the same as that of Q.*
2. *The two maps from the executed global iteration spaces of P and Q into the lexical locations of P and Q take corresponding times into corresponding lexical locations.*

Further, we say that P and Q are iteratively similar *up to time t* iff the two conditions just enumerated hold up to and including time t, although possibly not after t.

The following technical lemma will be needed later.

1.1 Lemma *Let P be a program, and let there be an assignment to a variable A (which may be an array reference) at time t_1 in the execution of P. Let there be a later assignment to a variable B at time t_2. (B may or may not be the same as A.)*

Then there is a program Q which is iteratively similar to P up to time t_2 and such that

1. *All the variables in Q have the same shape as their corresponding variables in P.*
2. *Each array reference in Q assigned to up to and including time t_2 refers to the same array element as its corresponding reference in P assigned to at the same time.*
3. *The values assigned to each array reference in Q* before *time t_1 are the same as the corresponding values assigned in P.*
4. *The value assigned at time t_1 in Q is different from that assigned at time t_1 in P.*

PROOF. Q will be created by applying a finite number of simple transformations to P. The number of these transformations is no more than a small multiple of the number of steps in program execution between times t_1 and t_2.

First, let us define a pure function $f(s, s_0, e, c)$ as follows: s and s_0 are times (i.e., points in the global iteration space of P). e is an expression. c is a constant expression. f is defined as follows:

$$f(s, s_0, e, c) = \begin{cases} e & \text{if } s \neq s_0 \\ c & \text{if } s = s_0 \end{cases}$$

Now we begin our algorithm. Q starts out being the same as P. We step through the executions of P and Q in parallel until we arrive at time t_1. The statement executed at this time is either a **read** statement or a statement that assigns a value to A:

- If the statement is a **read** statement, change the input of the program Q by reading in a value different from that read in program P at time t_1.
- If the statement is an assignment statement, then since we are executing program P, we know what the value being assigned at time t_1 is. Let c be a different value. Let e be the right-hand side of the assignment statement. Replace the right-hand side of the assignment statement in Q by the function call $f(t, t_1, e, c)$, where t is a representation of the current time.

In either case, what we have done ensures that the value assigned at time t_1 in Q differs from that in P but does not change the behavior of Q in any respect before time t_1.

Now we continue the execution of programs P and Q in parallel. We consider each statement executed at time s, where s starts from the next time after t_1 and continues up to and including t_2. For each such statement, we perform the following actions:

- If the statement in P contains an expression that affects control flow—in our language these are only **if** guards, **while** conditions, and **do** loop bounds—then for each such expression e (note that e may have already been changed by a previous step), let c be the value of e in P. Replace e by the function call $f(t, s, e, c)$. This ensures that the control flow at time s in Q is the same as that in P, and does not change the behavior of Q at any time previous to time s.

- If the statement in P is an assignment statement—say to an element of an array G—then for each subscript expression e in G, let c be the value of e in P at time s. Replace e by $f(t, s, e, c)$ in program Q. Also replace the expression e which is the right-hand side of the assignment statement in the same way.
- In any other case, do nothing to the statement in Q and continue to the next statement.

It is evident that each of these actions preserves the conditions of the lemma, and so by induction those conditions are still true when the algorithm concludes at time t_2. □

1.1.4 SSA form

For the moment, let us assume that our program contains only scalars.

A program is in *static single assignment* form (SSA form) if each variable is the target of exactly one assignment in the program text (Cytron et al., 1991).

Figure 1.2 shows a simple program fragment and the result of translating it into SSA form.

```
a = 0                          a_0 = 0
do i = 1, n                    do i = 1, n
    print a                        a_1 = φ(a_0, a_2)
    a = a + 1                      print a_1
end do                             a_2 = a_1 + 1
                               end do
```

Figure 1.2: A program and its translation into SSA form.

Thus, programs are converted into SSA form by the introduction of ϕ assignments. A ϕ assignment (or ϕ statement) is an assignment such as the first statement in the body of the SSA program in Figure 1.2; its right-hand side is called a ϕ function. The purpose of a ϕ function is to choose which of its arguments should be returned and assigned to the left-hand side of the assignment. As we have written it here, the ϕ function is not actually a function—that is, there is no way for it to decide which of its arguments to pick. This has been the convention in standard SSA expositions, because SSA form is often not actually used for code generation but only for analysis, and the ϕ assignments are subsequently eliminated. When we come to Array SSA form below, we will see how ϕ functions can be turned into honest functions.

Without specifying a specific way of constructing SSA form, but following Cytron et al. (1991), we say simply that a (scalar program) P' is a *translation into SSA form* of an original source program P iff it satisfies the following conditions:

1. P' has the same control structure as P. The executed global iteration space of P' is the same as that of P with the addition of times corresponding to the added ϕ assignments, as described in item 3 below. We use this new executed global iteration space for P as well, by simply regarding the points in the program P that correspond to these new times as null statements. In this way, P and P' continue to have identical global iteration spaces.

2. For each variable V in the original program P, each assignment to V has been replaced by an assignment to a distinct variable V_i, thus leaving the program in SSA form.

   We will call these variables $\{V_i\}$ the *SSA variants* of the source variable V, or the *SSA variables* corresponding to the source variable V.

   Note that loop indices have no explicit assignments in the source program, and therefore have no SSA variants.

3. ϕ assignments are introduced. One way this can be done is as follows: let the flow graph for P be constructed with each statement in P being a node of the graph. If two non-null paths in the flow graph—$X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$—converge at a node Z, and in program P nodes X and Y are assignments to variables V_i and V_j (obtained from the original program variable V), then a new variable V_k is created, and a ϕ assignment $V_k \leftarrow \phi(V_i, V_j)$ has been inserted at Z in P'.

   We do not, however, presume that this particular way of introducing ϕ assignments has been followed. Whatever way is used, however, must maintain the semantics of the original program, as specified in the last item below.

4. Each use of V in the original program P has been replaced by a mention of one of these new variables V_i.

5. The value of any variable V_i encountered during execution of a source statement, or defined by a ϕ assignment, in the program P' (say, at a point t in the global iteration space of P') is the same as the corresponding value of V at the same point in global iteration space in the program P.

It is the last of these items that ensures that P' is semantically equivalent to P. This item as we have stated it is slightly more general than that stated in Cytron et al. (1991), but it is easily seen to hold for the construction in that paper.

There is an additional constraint that we place on the SSA construction algorithm: it says that the algorithm is insensitive to non-structural changes in the program:

If the source programs P and Q are structurally similar, and if P' and Q' are the transformed versions of P and Q generated by the SSA algorithm, then the numbering of SSA variants and the placement and names of the arguments of the ϕ functions is the same in P' and Q'.

This says slightly more than that P' and Q' are structurally similar: In particular, the arguments to the ϕ functions and the variants assigned to in those assignments are the same in P' and Q'. Further, if V is a variable in an expression evaluated in P, and if the same variable V occurs in an expression in the same lexical location in Q, then V is replaced by the same variant V_i in P' and Q'.

So for instance, if programs P and Q differ only in the right-hand side of one or more assignment statements, the algorithm applied to P and Q will produce programs having exactly the same variables V_i defined in exactly the same positions, and will have exactly the same ϕ assignments in exactly the same positions.

This constraint is easily seen to be satisfied for the construction in Cytron et al. (1991), because that construction depends only on the control flow structure of the source program and the positions of assignments to variables in the source program.

We do not specify a particular algorithm for the construction of SSA form, however. We simply agree that any program P' satisfying the above conditions and the above constraint is an SSA translation of the program P.

If a loop has a ϕ assignment on entrance (i.e., at the top of the loop body, as in Figure 1.2), we sometimes refer to this assignment as a *wrap-around* ϕ, because it "wraps around" a value computed on one iteration of the loop to the next.

For future reference, we point out a simple and very particular property of programs in SSA form. One could certainly list much stronger properties, but this is all we really will need below:

1.2 Lemma *If the source program P containing a loop L is transformed into the SSA form P' by our algorithm, if V is a variable in P, and if*

- *The transformed program P' contains a definition of a variant V_i of V at a ϕ assignment inside L.*
- *There is no source assignment in L of any variant of V whose value can reach the assignment to V_i by a path entirely contained within the body of L.*

then the value assigned to V_i at its definition is an invariant of the loop L.

PROOF. Since there is no source assignment inside L that reaches the definition of V_i on a path that remains in the body of L, the value assigned to V_i must be a value that was assigned in the original program P to V before the loop was entered, and by condition 5 above, it must be the *last* such value that was assigned to V before the loop was entered. This value certainly does not change as the loop is executed. □

This Lemma simply points out that our SSA algorithm never produces a program that looks like this:

$V_1 = 1$
$V_2 = 2$
do $i = 1, 100$
  $V_3 = \phi(V_1, V_4)$
  $V_4 = \phi(V_2, V_3)$
end do

1.2 Array SSA form

1.2.1 The naive Array SSA construction

We begin by converting our original source program to Array SSA form. This form was introduced by Knobe and Sarkar (1998). Rather than just referring to that paper, we will describe the construction because we need to be a little more precise about some issues in order to handle our subsequent conversion to dynamic single assignment form.

First, let us note that scalar SSA form can be adapted to apply to arrays, as discussed in Cytron et al. (1991, pages 460–461). Figure 1.3 shows a simple program involving an array, and its translation into SSA form.

```
a(:) = 0
do i = 1, n
    a(i) = a(f(i)) + 1
    print a(g(i))
end do
```

```
a_0(:) = 0
do i = 1, n
    a_1 = φ(a_0, a_3)
    a_2(i) = a_1(f(i)) + 1
    a_3 = φ(a_1, a_2)
    print a_3(g(i))
end do
```

Figure 1.3: A program with an array and its translation into SSA form.

In that example, the ϕ assignment defining a_1 is placed exactly where the ordinary scalar SSA algorithm (whichever one we are using) would place it. We call this a *control ϕ assignment*.

In addition, however, another ϕ assignment is introduced, to assign to a_3. The function of this assignment is to merge the single element assigned to $a_2(i)$ into the entire array. Such an assignment is not needed in a purely scalar program, since in such a program assignment to a scalar redefines the entire object. We call this assignment a *definition ϕ assignment*.

We know a priori that the correct value that must be assigned to each element of an array on the left-hand side of a ϕ assignment is one of the values of the corresponding elements of the arrays that are the arguments to the ϕ function. We need to find a way of making this selection explicit; that is, of turning the ϕ into an honest function.

Now for a given ϕ function $\phi(a_1, a_2, \ldots, a_n)$, say and a given element $\vec{k}$, the value returned by the ϕ function should be that value in the set $\{a_1(\vec{k}), a_2(\vec{k}), \ldots, a_n(\vec{k})\}$ which was assigned last. We can accomplish this by introducing, for each SSA variant a_i an auxiliary array $\tau_{\mathrm{v}} a_i$ such that at any point in program execution $\tau_{\mathrm{v}} a_i(\vec{k})$ holds the time in global iteration space at which the value currently held by $a_i(\vec{k})$ was assigned to that element. Think of τ_{v} as standing for "time of assignment to the variant", or simply as "variant time".

$\tau_{\mathrm{v}} a_i$ is easy to compute: After each assignment to an element $a_i(\vec{k})$ (e.g., at a source definition) we simply define $\tau_{\mathrm{v}} a_i(\vec{k})$ to be the current time. And after each assignment to the whole array a_i (e.g., at a ϕ assignment to a_i), we simply assign to every element of $\tau_{\mathrm{v}} a_i$ the current time. (See Figure 1.4.)

We then add in the arrays $\tau_{\mathrm{v}} a_i$ as additional arguments to the ϕ functions, as shown in that same figure. The code on the right-hand side of the same figure shows how the ϕ functions are interpreted.

Note that although we have introduced new assignment statements into the program to assign to the τ_{v} variables, we regard these statements as being executed simultaneously with their corresponding source statements. The reason we do this is that these variables (as well as the τ_{w} variables to be defined later) will for the most part not appear in the final code. Further, each τ_{v} variable (and later, τ_{w} variable as well) is so closely associated with the variable just defined at the preceding definition that it makes sense to view it as being defined at the same point in global iteration space. So no change is made to the global iteration space in adding these assignments.

The τ_{v} variables thus turn the ϕ functions into actual functions, whose values are uniquely determined by the values of their arguments. This can be useful in scalar SSA form—it is similar to

$a_0(:) = 0$
$\tau_v a_0(:) = [1_\lambda]$
do $i = 1, n$
  $a_1 = \phi(a_0, \tau_v a_0; a_3, \tau_v a_3)$
  $\tau_v a_1 = [2_\lambda, i, 1_\lambda]$

  $a_2(i) = a_1(f(i)) + 1$
  $\tau_v a_2(i) = [2_\lambda, i, 2_\lambda]$

  $a_3 = \phi(a_1, \tau_v a_1; a_2, \tau_v a_2)$
  $\tau_v a_3 = [2_\lambda, i, 3_\lambda]$

  print $a_3(g(i))$
end do

$a_0(:) = 0$
$\tau_v a_0(:) = [1_\lambda]$
do $i = 1, n$
  forall $(k = 1 : n)$
    $$a_1(k) = \begin{cases} a_0(k) & \text{if } \tau_v a_0(k) > \tau_v a_3(k) \\ a_3(k) & \text{otherwise} \end{cases}$$
  end forall
  $\tau_v a_1 = [2_\lambda, i, 1_\lambda]$

  $a_2(i) = a_1(f(i)) + 1$
  $\tau_v a_2(i) = [2_\lambda, i, 2_\lambda]$

  forall $(k = 1 : n)$
    $$a_3(k) = \begin{cases} a_1(k) & \text{if } \tau_v a_1(k) > \tau_v a_2(k) \\ a_2(k) & \text{otherwise} \end{cases}$$
  end forall
  $\tau_v a_3 = [2_\lambda, i, 3_\lambda]$

  print $a_3(g(i))$
end do

Figure 1.4: Semantics of the ϕ functions.

what has been called "gated SSA form" (Ballance et al., 1990)—but it is actually essential in what we will be doing.

We refer to this form of the program that we have just described as *naive Array SSA form*. It is not the final version of Array SSA form that we will construct. But it clearly preserves the semantics of the original program. To be precise, if P denotes the original source program and P' denotes the naive Array SSA program, then

1. P' has the same control structure as P and the same global iteration space.

2. With the exception of loop indices, each variable V in P has been replaced by a collection $\{V_i\}$ of variables in P' so that each V_i is assigned to in only one lexical location.

3. ϕ assignments are introduced, as in scalar SSA form, and in addition, definition ϕ assignments are introduced after each source definition. Each ϕ assignment whose arguments are variants of V defines a new variant of V.

4. Each use of V in the original program is replaced by a mention of one of these new variables V_i.

5. Corresponding to each variant V_i, a new array $\tau_v V_i$ is created. Each assignment to one or more elements of V_i assigns the current time to the corresponding elements of $\tau_v V_i$.

   The variables $\tau_v V_i$ are added as explicit arguments to the ϕ functions, and are used to specify the selection made by the ϕ function, as follows:

The ϕ function

$$V_{k_0} = \phi(V_{k_1}, \tau_{\mathrm{v}} V_{k_1}; \ldots; V_{k_n}, \tau_{\mathrm{v}} V_{k_n})$$

is interpreted as

forall $(\vec{k})$

$$V_{k_0}(\vec{k}) = \begin{cases} V_{k_1}(\vec{k}) & \text{if } \tau_{\mathrm{v}} V_{k_1}(\vec{k}) = \max\big(\tau_{\mathrm{v}} V_{k_1}(\vec{k}), \ldots, \tau_{\mathrm{v}} V_{k_n}(\vec{k})\big) \\ \ldots & \\ V_{k_n}(\vec{k}) & \text{if } \tau_{\mathrm{v}} V_{k_n}(\vec{k}) = \max\big(\tau_{\mathrm{v}} V_{k_1}(\vec{k}), \ldots, \tau_{\mathrm{v}} V_{k_n}(\vec{k})\big) \end{cases}$$

end forall

6. The value of any array element $V_i(\vec{k})$ encountered during execution of a source statement, or defined by a ϕ assignment, in the program P' (say, at a point t in the global iteration space of P') is the same as the corresponding value of $V(\vec{k})$ at the same point in global iteration space in the program P.

The expressions serving as **if** guards in item 5 above, and also in Figure 1.4 are called *Boolean choice expressions*.

Note that the way we have defined τ_{v} variables (with their values in global iteration space), we are guaranteed that the cases in the expression for the value of $V_{k_0}(\vec{k})$ are all disjoint. In particular, the value of a ϕ function is independent of the order of its arguments.

For clarity, we will refer to all the SSA variants of all source program variables as *ordinary variables*. This will enable us to distinguish them from τ_{v} variables (and later, from τ_{w} variables) in descriptive text. Note in particular that loop indices are not classified as ordinary variables, since they have no SSA variants.

τ_{v} variables (and τ_{w} variables, to be defined below) have one characteristic that is quite different from ordinary variables: as we remarked earlier, the value $\perp$ of an ordinary variable simply means that that variable has not been given a value in the program. Such a value can never be referenced in the program. For τ_{v} variables, however, the value $\perp$ is significant: it is actually used in comparisons in Boolean choice expressions, and so it has real meaning.

1.2.2 SSA graphs and webs

By the *SSA graph* for a source variable A we mean the directed graph G constructed as follows:

1. The nodes of G are all the lexical references to all the SSA variants of A.
2. The edges of G fall into two classes:
   (a) There is an edge from the definition of each SSA variant to each of its uses.
   (b) There is an edge from each argument of a ϕ function to the variant defined by that function.

It is sometimes also useful to consider what we will call the *collapsed SSA graph* for a source variable A. This is the SSA graph with only one node for each variant. So all references to each variant are identified, and all the edges of the first kind are deleted. That is,

1. The nodes of the collapsed SSA graph for a source variable A are all the SSA variants of A.
2. There is an edge of this graph from each A_i that is an argument of a ϕ function to the variant A_j defined by that function.

Note that these graphs only take account of the ordinary variables. τ_{v} variables are not represented in these graphs.

We want to talk about connectedness, and for this purpose only, we will regard SSA graphs and collapsed SSA graphs as undirected graphs—that is, we just ignore the sense of each edge.

The SSA graph is connected iff the collapsed SSA graph is connected. Since the collapsed graph is the smaller graph, it is the preferred data structure to use when dealing with connectedness. Of course it is quite likely that neither of these data structures will actually be constructed. It may, for instance, be quite possible to show that the SSA graph is connected without actually constructing it.

If A is an array, then the SSA graph for A is generally connected. If A is a scalar, however, the SSA graph might be disconnected—each component corresponds to disjoint lifetimes of a variable. Further, even if A is an array, it turns out that in some cases we can resolve some of the ϕ functions so that the SSA graph for A becomes disconnected.

Following Muchnick (1997), we call the set of SSA variables corresponding to the nodes of a connected component of an SSA graph a *web*. We use the notation $web(A_i)$ to denote the web of variables containing the SSA variable A_i.

1.2.3 τ_{w} variables: time of entry to the web

At any time in the course of execution of the program the variable $\tau_{\mathrm{v}}A_i(\vec{k})$ holds the time that the value currently in $A_i(\vec{k})$ was assigned to $A_i(\vec{k})$. There is another time that we can associate with $A_i(\vec{k})$, which is no later than $\tau_{\mathrm{v}}A_i(\vec{k})$ and often earlier. The variable that holds this new time will be denoted by $\tau_{\mathrm{w}}A_i$, and it is defined as follows:

Definition *For each ordinary variable A_i, there is an associated variable $\tau_{\mathrm{w}}A_i$. At any time t in the course of execution of the program, $\tau_{\mathrm{w}}A_i(\vec{k})$ holds the time at which the value currently in $A_i(\vec{k})$ was first assigned (at a source definition) to some element of the web containing A_i, subsequently reaching $A_i(\vec{k})$ by a chain of ϕ assignments.*

To help remember this notation, think of τ_{w} as standing for "time of entry to the web", or simply as "web time". τ_{w} variables were called @ variables in Knobe and Sarkar (1998).

Thus, if A_i is defined at a source definition, then $\tau_{\mathrm{w}}A_i(\vec{k}) = \tau_{\mathrm{v}}A_i(\vec{k})$ for all $\vec{k}$. But if A_i is defined at a ϕ assignment, then $\tau_{\mathrm{w}}A_i(\vec{k}) < \tau_{\mathrm{v}}A_i(\vec{k})$ for all $\vec{k}$.

Computing $\tau_{\mathrm{w}}A_i$ is straightforward:

- At a source definition, $\tau_{\mathrm{w}}A_i(\vec{k})$ is defined to be equal to $\tau_{\mathrm{v}}A_i(\vec{k})$.
- At a ϕ assignment

forall $(\vec{k})$

$$A_{k_0}(\vec{k}) = \begin{cases} A_{k_1}(\vec{k}) & \text{if } \tau_{\mathrm{v}} A_{k_1}(\vec{k}) = \max\bigl(\tau_{\mathrm{v}} A_{k_1}(\vec{k}), \ldots, \tau_{\mathrm{v}} A_{k_n}(\vec{k})\bigr) \\ \ldots & \\ A_{k_n}(\vec{k}) & \text{if } \tau_{\mathrm{v}} A_{k_n}(\vec{k}) = \max\bigl(\tau_{\mathrm{v}} A_{k_1}(\vec{k}), \ldots, \tau_{\mathrm{v}} A_{k_n}(\vec{k})\bigr) \end{cases}$$

end forall

we extend the ϕ assignment by adding a definition of $\tau_{\mathrm{w}} A_{k_0}$ to the **forall**, as follows:

forall $(\vec{k})$

$$A_{k_0}(\vec{k}) = \begin{cases} A_{k_1}(\vec{k}) & \text{if } \tau_{\mathrm{v}} A_{k_1}(\vec{k}) = \max\bigl(\tau_{\mathrm{v}} A_{k_1}(\vec{k}), \ldots, \tau_{\mathrm{v}} A_{k_n}(\vec{k})\bigr) \\ \ldots & \\ A_{k_n}(\vec{k}) & \text{if } \tau_{\mathrm{v}} A_{k_n}(\vec{k}) = \max\bigl(\tau_{\mathrm{v}} A_{k_1}(\vec{k}), \ldots, \tau_{\mathrm{v}} A_{k_n}(\vec{k})\bigr) \end{cases}$$

$$\tau_{\mathrm{w}} A_{k_0}(\vec{k}) = \begin{cases} \tau_{\mathrm{w}} A_{k_1}(\vec{k}) & \text{if } \tau_{\mathrm{v}} A_{k_1}(\vec{k}) = \max\bigl(\tau_{\mathrm{v}} A_{k_1}(\vec{k}), \ldots, \tau_{\mathrm{v}} A_{k_n}(\vec{k})\bigr) \\ \ldots & \\ \tau_{\mathrm{w}} A_{k_n}(\vec{k}) & \text{if } \tau_{\mathrm{v}} A_{k_n}(\vec{k}) = \max\bigl(\tau_{\mathrm{v}} A_{k_1}(\vec{k}), \ldots, \tau_{\mathrm{v}} A_{k_n}(\vec{k})\bigr) \end{cases}$$

end forall

We may abbreviate this simply by writing

$$\begin{aligned} A_{k_0} &= \phi(A_{k_1}, \tau_{\mathrm{v}} A_{k_1}; \ldots; A_{k_n}, \tau_{\mathrm{v}} A_{k_n}) \\ \tau_{\mathrm{w}} A_{k_0} &= \phi(\tau_{\mathrm{w}} A_{k_1}, \tau_{\mathrm{v}} A_{k_1}; \ldots; \tau_{\mathrm{w}} A_{k_n}, \tau_{\mathrm{v}} A_{k_n}) \end{aligned}$$

Figure 1.5 contains a simple scalar example that illustrates the difference between τ_{v} and τ_{w} variables.

$x = 0$
do $i = 1, 2$
  print x
  $x = 5$
end do

Figure 1.5: The value printed on iteration 2 is computed on iteration 1.

In this example, the value printed on iteration 2 is computed on iteration 1. If we convert this program to SSA form, we have the code in Figure 1.6. The assignments to the τ_{v} and τ_{w} variables on each iteration of the loop occur with the following values:

	value before loop	value ($i = 1$)	value ($i = 2$)
$(x_0, \tau_{\mathrm{w}} x_0, \tau_{\mathrm{v}} x_0)$	$(0, [1_\lambda], [1_\lambda])$		
$(x_1, \tau_{\mathrm{w}} x_1, \tau_{\mathrm{v}} x_1)$	$(\bot, \bot, \bot)$	$(0, [1_\lambda], [1]^g)$	$(5, [1]^g, [2]^g)$
$(x_2, \tau_{\mathrm{w}} x_2, \tau_{\mathrm{v}} x_2)$	$(\bot, \bot, \bot)$	$(5, [1]^g, [1]^g)$	$(5, [2]^g, [2]^g)$

Thus on iteration 2 of the loop, $\tau_{\mathrm{w}} x_1$ is assigned the value $[1]^g$, which denotes the iteration at which the value last assigned to x_1 (i.e., the value 5) was computed (and assigned to the variable x_2). However, this value was not actually stored into the variable x_1 until iteration 2 (that is, at time $\tau_{\mathrm{v}} x_1$).

$x_0 = 0$
$\tau_{\mathrm{v}} x_0 = [1_\lambda]$
$\tau_{\mathrm{w}} x_0 = [1_\lambda]$
do $i = 1, 2$
  $x_1 = \phi(x_0, \tau_{\mathrm{v}} x_0; x_2, \tau_{\mathrm{v}} x_2)$
  $\tau_{\mathrm{v}} x_1 = [2_\lambda, i, 1_\lambda]$
  $\tau_{\mathrm{w}} x_1 = \phi(\tau_{\mathrm{w}} x_0, \tau_{\mathrm{v}} x_0; \tau_{\mathrm{w}} x_2, \tau_{\mathrm{v}} x_2)$
  print x_1
  $x_2 = 5$
  $\tau_{\mathrm{v}} x_2 = [2_\lambda, i, 3_\lambda]$
  $\tau_{\mathrm{w}} x_2 = [2_\lambda, i, 3_\lambda]$
end do

Figure 1.6: The code from Figure 1.5 in SSA form, with the τ_{v} and τ_{w} variables added.

The reader may fear that we are suffering from variable pollution at this point. There are several things to keep in mind:

- These new variables are actually needed for our analysis.
- The τ_{v} variables will not appear in generated code.
- In typical cases most or all of the τ_{w} variables can be optimized away.

These points will be justified below.

As a matter of notation, by $\left(\tau_{\mathrm{v}} A_i(\vec{k})\right)_j$ we mean the j^{th} local iteration component of $\tau_{\mathrm{v}} A_i(\vec{k})$. Of course $\tau_{\mathrm{v}} A_i(\vec{k})$ also has lexical components, but we will not extract them by subscripting in this way. The same convention will be used for extracting iteration components of τ_{w} variables.

For example, if at a particular reference $\tau_{\mathrm{v}} A_i(\vec{k}) = [3_\lambda, i, 2_\lambda, j, 2_\lambda]$, then at that same reference $\left(\tau_{\mathrm{v}} A_i(\vec{k})\right)_2 = j$.

1.2.4 ϕ iteration indices

A ϕ assignment, which in naive Array SSA form has the form

$$A_6(:,:,:) = \phi(A_5(:,:,:), \tau_{\mathrm{v}} A_5(:,:,:); A_4(:,:,:), \tau_{\mathrm{v}} A_4(:,:,:))$$
$$\tau_{\mathrm{w}} A_6(:,:,:) = \phi(\tau_{\mathrm{w}} A_5(:,:,:), \tau_{\mathrm{v}} A_5(:,:,:); \tau_{\mathrm{w}} A_4(:,:,:), \tau_{\mathrm{v}} A_4(:,:,:))$$

can be written more explicitly like this:

forall (p, q, r)

$$A_6(p,q,r) = \begin{cases} A_5(p,q,r) & \text{if } \tau_{\mathrm{v}} A_5(p,q,r) > \tau_{\mathrm{v}} A_4(p,q,r) \\ A_4(p,q,r) & \text{otherwise} \end{cases}$$

$$\tau_{\mathrm{w}} A_6(p,q,r) = \begin{cases} \tau_{\mathrm{w}} A_5(p,q,r) & \text{if } \tau_{\mathrm{v}} A_5(p,q,r) > \tau_{\mathrm{v}} A_4(p,q,r) \\ \tau_{\mathrm{w}} A_4(p,q,r) & \text{otherwise} \end{cases}$$

end forall

Here we have introduced explicit iteration variables p, q, and r to express the computations inside the ϕ assignment. We will find it convenient to regard these as representing the values of additional coordinates in the local iteration space of the statements inside the **forall**. We call these coordinates *ϕ iteration coordinates*. Just as with other iteration dimensions, they are labeled by indices of the form i_j.

This has the effect of increasing the depth of the local iteration space (and similarly, the global iteration space) at each ϕ assignment.

1.2.5 ϕ iteration components of τ_{v} variables

There is a simple result that won't be needed until later but that is convenient to mention at this point: consider a ϕ assignment, say the one illustrated in Section 1.2.4 (page 18):

forall (p, q, r)

$$A_6(p,q,r) = \begin{cases} A_5(p,q,r) & \text{if } \tau_{\mathrm{v}} A_5(p,q,r) > \tau_{\mathrm{v}} A_4(p,q,r) \\ A_4(p,q,r) & \text{otherwise} \end{cases}$$

$$\tau_{\mathrm{w}} A_6(p,q,r) = \begin{cases} \tau_{\mathrm{w}} A_5(p,q,r) & \text{if } \tau_{\mathrm{v}} A_5(p,q,r) > \tau_{\mathrm{v}} A_4(p,q,r) \\ \tau_{\mathrm{w}} A_4(p,q,r) & \text{otherwise} \end{cases}$$

end forall

Since this **forall** implements a ϕ assignment, there is no control ϕ after it. Therefore it is possible to have a reference to A_6 subsequent to this **forall**. Say the reference is of the form $A_6(S_1, S_2, S_3)$ where the S_i are subscript expressions. Let us say that the control ϕ iteration variables p, q, and r are indexed by iteration indices i_7, i_8, and i_9, respectively. Then the value of $\left(\tau_{\mathrm{v}} A_6(S_1, S_2, S_3)\right)_7$ (which denotes the value in the "i_7-coordinate" of the point in the local iteration space of the definition of A_6 at which $A_6(S_1, S_2, S_3)$ was defined, must be S_1, since it occurs when $p = S_1$, and $I(i_7) = p$.

This point is quite immediate, but since we will need to refer to it explicitly below, we isolate it as a lemma:

1.3 Lemma *If*

- $A(S_1, \ldots, S_n)$ *is a use of an array that is defined at a ϕ assignment,*
- i_k *is a ϕ iteration index of that ϕ assignment, and d_q is the corresponding data index*

then $\left(\tau_{\mathrm{v}} A(S_1, \ldots, S_n)\right)_k = S_q$.

For example, if in the example above there was a subsequent reference of the form $A_6(5, 3*i, i-j)$, then $\big(\tau_{\mathrm{v}} A_6(5, 3*i, i-j)\big)_8 = 3*i$.

1.2.6 Extending the subscript map to iteration dimensions

For convenience, we extend the subscript map σ, which we defined earlier on page 4. As previously defined, σ takes as input an array reference (which is always implicit) and an index d_k into the data space of the array of the reference. We will extend this so that instead of d_k, we can pass an iteration index i_j, say. In such a case, the value of σ will be the j^{th} component of the τ_{v} variable associated with that reference.

For instance, suppose R is a reference to an element $A\big(\sigma(d_1), \sigma(d_2), \sigma(d_3)\big)$ of a 3-dimensional array A inside 4 nested loops. With respect to this reference, we have $\sigma(i_2) = \big(\tau_{\mathrm{v}} A(\sigma(d_1), \sigma(d_2), \sigma(d_3))\big)_2$. This is an expression which can be regarded as a function of the program state. If for instance, the actual array element is $A(2, 17, 1)$ and if this element was last assigned to at time $[4, 3, 1, 5]^g$, then the current value of $\sigma(i_2)$ with respect to this reference is 3.

In particular, if the reference is a definition, then $\sigma(i_j)$ is just the loop index corresponding to the iteration space dimension i_j.

Note that $\sigma(i_j)$ at a reference to an array A is defined only for those iteration components i_j of the local iteration space at the definition of A.

Note also that if the reference itself is simply a loop variable $I(i_j)$ corresponding to the i_j coordinate of a local iteration space, then $\sigma(i_j) = I(i_j)$. This is because in our language a loop variable is assigned once on entry to each iteration of the loop, and cannot be changed during that iteration.

The reason for this particular definition of $\sigma(i_j)$ will become clear in Section 2.3.1 (page 41), where we specify how to rewrite a program corresponding to a particular set of dimensions. The definition itself is used in the main algorithm in Chapter 2, and in particular in Lemma 2.8 (page 51).

1.2.7 Initial Array SSA form

It turns out that we can expose more opportunities for optimization if we can replace the times represented by τ_{v} variables in the Boolean choice expressions of the program in naive Array SSA form by earlier times. The particular way we will do this is to replace each τ_{v} variable in each Boolean choice expression in each ϕ function by its corresponding τ_{w} variable.

With this change, the program is in what we will call *initial Array SSA form*. There are two observations we need to make at this point:

1. The cases in a ϕ function may no longer be disjoint. It may be, for instance, that $\tau_{\mathrm{w}} A_i(\vec{k}) = \tau_{\mathrm{w}} A_j(\vec{k})$. (This could never happen with τ_{v} variables.) This is really not a problem, because if $\tau_{\mathrm{w}} A_i(\vec{k}) = \tau_{\mathrm{w}} A_j(\vec{k})$ then the values assigned at those two times (which are really the same time) are of course equal, and so the values currently in $A_i(\vec{k})$ and $A_j(\vec{k})$ are equal. Thus $A_i(\vec{k}) = A_j(\vec{k})$, so it really does not matter which of the two variants of A_i is chosen by the ϕ function. So while there may be some formal ambiguity in the ϕ function, it is harmless.

2. The reasoning that shows that the transformation to naive array SSA form preserves the semantics of the original program shows in much the same way that the transformation to initial Array SSA form preserves the semantics.

   However, we can also derive the correctness of the transformation to initial Array SSA form from that of naive Array SSA form by proving a somewhat more general result first. We will do this next.

1.2.8 Equivalence of the initial and naive Array SSA constructions

Let P be an original source program, and let P_{v} be the naive Array SSA form of P. We have already noted that P_{v} preserves the semantics of P.

Let P_{w} be the program in initial Array SSA form constructed from P_{v} as just specified. We will show that P_{w} is semantically equivalent to P_{v} by showing that the transformation from P_{v} to P_{w} leaves invariant the value of each Boolean choice expression. Since it is only those Boolean choice expressions that were modified, it follows immediately by induction that the control flow in P_{w} is the same as that in P_{v} and that each value computed in P_{w} is the same as the corresponding value in P_{v}. And that means that P_{w} is semantically equivalent to P_{v} and thus to P.

1.4 Lemma *If A and B are in the same web, and if at some point in the execution of the program $\tau_{\mathrm{w}}A(\vec{k}) < \tau_{\mathrm{w}}B(\vec{k})$ for some $\vec{k}$, then (at that same point)*

$$\tau_{\mathrm{w}}A(\vec{k}) \le \tau_{\mathrm{v}}A(\vec{k}) < \tau_{\mathrm{w}}B(\vec{k}) \le \tau_{\mathrm{v}}B(\vec{k})$$

PROOF. We know that the two outer inequalities are automatically true, so we only have to prove the inner one.

Suppose it is not true. There are two possibilities:

1. $\tau_{\mathrm{w}}B(\vec{k}) = \tau_{\mathrm{v}}A(\vec{k})$
2. $\tau_{\mathrm{w}}B(\vec{k}) < \tau_{\mathrm{v}}A(\vec{k})$

We consider these two cases separately.

Case 1. In this case we have

$$\tau_{\mathrm{w}}A(\vec{k}) < \tau_{\mathrm{w}}B(\vec{k}) = \tau_{\mathrm{v}}A(\vec{k})$$

Now $P\big(\tau_{\mathrm{v}}A(\vec{k})\big)$ must be a ϕ assignment, since $\tau_{\mathrm{w}}A(\vec{k}) < \tau_{\mathrm{v}}A(\vec{k})$. But this is impossible, since $P\big(\tau_{\mathrm{v}}A(\vec{k})\big) = P\big(\tau_{\mathrm{w}}B(\vec{k})\big)$, which must be a source statement. So Case 1 cannot happen.

Case 2. In this case we have

$$\tau_{\mathrm{w}}A(\vec{k}) < \tau_{\mathrm{w}}B(\vec{k}) < \tau_{\mathrm{v}}A(\vec{k}) \tag{1.1}$$

We must show that such a relation cannot hold. In doing this, we may assume that $\tau_{\mathrm{w}}B(\vec{k})$ is the last time before time $\tau_{\mathrm{v}}A(\vec{k})$ at which the $\vec{k}$ element of any member of $web(A)$ was assigned to at a source statement.

Further, we may assume that the value assigned at time $\tau_w B(\vec{k})$ is actually different from that assigned at time $\tau_w A(\vec{k})$. For if not, we can apply Lemma 1.1 to our original source program P, with t_1 in that lemma being $\tau_w B(\vec{k})$ and t_2 being $\tau_v A(\vec{k})$, to create a program Q in which the values assigned at time t_1 in P and Q are different. Because P and Q are structurally similar, we know that whatever SSA algorithm we are using gives Q_v the same SSA structure as P_v. That is, Q_v and P_v have the same SSA variants, numbered the same way, and the same placement of ϕ functions with the same arguments.

Further, the variables assigned to in Q_v are the same as those in P_v up through time t_2, and so the constraint of the preceding paragraph still holds, and in addition relation 1.1 continues to hold in Q_v. If we can show that this leads to a contradiction in Q_v, then there must have already been a contradiction in P_v. So we will simply assume that the value assigned at time $\tau_w B(\vec{k})$ is actually different from that assigned at time $\tau_w A(\vec{k})$.

Let us denote the variable in the original program P of which A and B are SSA variants by X. The inequality above says the following:

1. At time $\tau_w A(\vec{k})$ a value is assigned to the $\vec{k}$ element of some variant of X at a source statement. We know that at the same time in the original program $X(\vec{k})$ is assigned the same value by the corresponding source statement.
2. At the later time $\tau_w B(\vec{k})$ a different value is assigned to the $\vec{k}$ element of another variant of X at a source statement. At that same time in the original program $X(\vec{k})$ is overwritten by this different value, and this is the last time before time $\tau_v A(\vec{k})$ at which the element $X(\vec{k})$ is assigned to in P.
3. At the still later time $\tau_v A(\vec{k})$, $A(\vec{k})$ is assigned (by a ϕ assignment) the value that was originally computed at time $\tau_w A(\vec{k})$. Again, this value must be the value held by $X(\vec{k})$ at that time $\tau_v A(\vec{k})$. But this value in $X(\vec{k})$ is no longer the original value computed at time $\tau_w A(\vec{k})$, since it was overwritten at time $\tau_w B(\vec{k})$.

Since this is a contradiction, Case 2 cannot happen, and we are done. □

1.5 Theorem (Equivalence of the initial and naive Array SSA constructions) *The initial Array SSA program P_w is semantically equivalent to the original source program. That is, the value of any array element $A_i(\vec{k})$ encountered in P_w at time t during execution of a source statement or defined by a ϕ assignment is the same as the corresponding value $A(\vec{k})$ in the original source program at the same time t.*

PROOF. We know that P is semantically equivalent to P_v. Let

$$A_i = \phi(A_{i_1}, \tau_v A_{i_1}; \ldots; A_{i_k}, \tau_v A_{i_k})$$

be any ϕ assignment in P_v. For a given vector $\vec{k}$ of subscripts, let $A_{i_j}(\vec{k})$ be that element chosen by the ϕ function. That is, $\tau_v A_{i_j}(\vec{k})$ is the largest of the τ_v values occurring as arguments to the ϕ function corresponding to the subscript vector $\vec{k}$. Lemma 1.4 then shows that $\tau_w A_{i_j}(\vec{k})$ is at least as great as any other $\tau_w A_{i_k}(\vec{k})$ for all indices i_k occurring in the ϕ function. Thus, substituting for

each $\tau_{\mathrm{v}} A_{i_j}$ in the ϕ function the value $\tau_{\mathrm{w}} A_{i_j}$ does not change the value returned by the ϕ. (Recall that no harm is done if two values $\tau_{\mathrm{w}} A_{i_j}(\vec{k})$ and $\tau_{\mathrm{w}} A_{i_k}(\vec{k})$ turn out to be equal.) And doing this consistently throughout the program turns program P_{v} into P_{w} with no change in semantics, and thus concludes the proof. □

Note that once the program is in initial Array SSA form, the propagation of τ_{w} variables becomes formally simpler: in naive Array SSA form τ_{w} variables would be combined at a ϕ assignment like this:

forall $(\vec{k})$

$$\tau_{\mathrm{w}} A_{k_0}(\vec{k}) = \begin{cases} \tau_{\mathrm{w}} A_{k_1}(\vec{k}) & \text{if } \tau_{\mathrm{v}} A_{k_1}(\vec{k}) = \max\bigl(\tau_{\mathrm{v}} A_{k_1}(\vec{k}), \ldots, \tau_{\mathrm{v}} A_{k_n}(\vec{k})\bigr) \\ \ldots & \\ \tau_{\mathrm{w}} A_{k_n}(\vec{k}) & \text{if } \tau_{\mathrm{v}} A_{k_n}(\vec{k}) = \max\bigl(\tau_{\mathrm{v}} A_{k_1}(\vec{k}), \ldots, \tau_{\mathrm{v}} A_{k_n}(\vec{k})\bigr) \end{cases}$$

end forall

After replacing the τ_{v} variables in these Boolean choice expressions by the corresponding τ_{w} variables, thus putting the program in initial Array SSA form, this combination can be written more simply as

$$\tau_{\mathrm{w}} A_{k_0}(\vec{k}) = \max(\tau_{\mathrm{w}} A_{k_1}, \ldots, \tau_{\mathrm{w}} A_{k_n})$$

Note by the way that the same argument used in the proof of Theorem 1.5 yields the following statement, which we will identify for future reference:

1.6 Corollary *In the program P_{v}, the following properties hold:*

1. *If $A_i(\vec{k})$ is assigned at a source assignment, then the (conceptually simultaneous) assignment to $\tau_{\mathrm{w}} A_i(\vec{k})$ assigns a value in global iteration space greater than the current value of every other $\tau_{\mathrm{w}} A_j(\vec{k})$ for each A_j in web(A_i).*
2. *If on the other hand A_i is defined by a ϕ assignment, then for each execution of that assignment statement, the (conceptually simultaneous) assignment to $\tau_{\mathrm{w}} A_i$ causes $\tau_{\mathrm{w}} A_i$ to be elementwise at least as great as the current value of every other $\tau_{\mathrm{w}} A_j$ for each A_j in web(A_i).*

PROOF. This follows immediately from Lemma 1.4. □

1.7 Lemma *If P is a program in initial Array SSA form and L is a loop in P, and if*

- *P contains a definition of a variable V_i at a ϕ assignment inside L.*
- *There is no source assignment in L of any variable in web(V_i) whose value can reach the assignment to V_i by a path contained entirely in the body of L.*

then the value assigned to V_i at its definition is an invariant of the loop L.

PROOF. If V_i is an ordinary variable, this follows immediately from Lemma 1.2 (page 11).

If V_i is a τ_{w} variable—say V_i is $\tau_{\mathrm{w}} B_i$—then the ϕ assignment defining $\tau_{\mathrm{w}} B_i$ occurs in conjunction with a ϕ assignment defining B_i. Further, since there is no source assignment in L of any τ_{w} variable whose value can reach the assignment to $\tau_{\mathrm{w}} B_i$ by a path in L, there must similarly be no source

assignment in L of any variable in $web(B_i)$ whose value can reach B_i by a path in L. So the value assigned to $B_i(\vec{k})$ is an invariant of the loop L.

Now if $\tau_{\mathrm{w}} B_i(\vec{k})$ takes on two different values on two different loop iterations, then $B_i(\vec{k})$ must have received values that entered the web at two different times, both before the loop was entered. Using Lemma 1.1, we may assume that these two values are different. But we have just observed that they must be the same. □

1.3 Admissible program transformations

1.3.1 Array SSA form and true dependences

Putting a program in initial Array SSA form can lead to an overly conservative dependence analysis. For instance, suppose we start with the code

$A(:) = 0$
do $i = 1, n$
  $A(i) = A(i) + 1$
end do

We know that in this code there is no cycle of dependences around the loop. Nevertheless, if we insert ϕ statements in the "obvious" places, we get the following code:

$A_0(:) = 0$
$\tau_{\mathrm{w}} A_0(:) = [1_\lambda]$
do $i = 1, n$
  $A_1(:) = \phi\big(A_0(:), \tau_{\mathrm{w}} A_0(:); A_3(:), \tau_{\mathrm{w}} A_3(:)\big)$
  $\tau_{\mathrm{w}} A_1(:) = \max\big(\tau_{\mathrm{w}} A_0(:), \tau_{\mathrm{w}} A_3(:)\big)$

  $A_2(i) = A_1(i) + 1$
  $\tau_{\mathrm{w}} A_2(i) = [i]^g$
  $A_3(:) = \phi\big(A_2(:), \tau_{\mathrm{w}} A_2(:); A_1(:), \tau_{\mathrm{w}} A_1(:)\big)$
  $\tau_{\mathrm{w}} A_3(:) = \max\big(\tau_{\mathrm{w}} A_2(:), \tau_{\mathrm{w}} A_1(:)\big)$
end do

In this form, it appears that there is a cycle of dependences, for we have the sequence $A_2 \to A_3 \to A_1 \to A_2$.

Now a clever compiler might be able to see that there is really no cycle here. However, we do not have to make it work so hard. The only reason there appears to be a cycle is that we have inserted a back edge in the dependence graph by placing a ϕ assignment (defining A_1) at the top of the loop. This ϕ assignment is actually unnecessary, because its only function is to propagate values generated within the loop body back to the top of the next iteration. However, no such values are ever used in a subsequent iteration—that is to say, there is no loop-carried dependence involving any of the variables A_i. Therefore, the control ϕ assignment at the top of the loop did not need to be inserted. We could simply have written the code like this:

$A_0(:) = 0$
$\tau_w A_0(:) = [1_\lambda]$
do $i = 1, n$
  $A_1(i) = A_0(i) + 1$
  $\tau_w A_1(i) = [i]^g$
  $A_2(:) = \phi(A_1(:), \tau_w A_1(:); A_0(:), \tau_w A_0(:))$
  $\tau_w A_2(:) = \max(\tau_w A_1(:), \tau_w A_0(:))$
end do

and in this form, it is clear that there is no cycle of dependences in the loop.

Here is another example:

$A(:) = 0$
do $i = 1, n$
  $A(i) = \ldots$
  if $(i == 1)$ **then**
    $\cdots = \ldots A(i) \ldots$
  else
    $\cdots = \ldots A(i-1) \ldots$
  end if
end do

Again, inserting the initial Array SSA code can result in unnecessary complexity:

$A_0(:) = 0$
$\tau_w A_0(:) = [1_\lambda]$
do $i = 1, n$
  $A_1(:) = \phi(A_0(:), \tau_w A_0(:); A_3(:), \tau_w A_3(:))$
  $\tau_w A_1(:) = \max(\tau_w A_0(:), \tau_w A_3(:))$

  $A_2(i) = \ldots$
  $\tau_w A_2(i) = [i]^g$
  $A_3(:) = \phi(A_2(:), \tau_w A_2(:); A_1(:), \tau_w A_1(:))$
  $\tau_w A_3(:) = \max(\tau_w A_2(:), \tau_w A_1(:))$

  if $(i == 1)$ **then**
    $\cdots = \ldots A_3(i) \ldots$
  else
    $\cdots = \ldots A_3(i-1) \ldots$
  end if
end do

Now using a sophisticated form of dependence analysis, the compiler can determine that the reference to $A_3(i)$ is really a reference to $A_2(i)$, and similarly the reference to $A_3(i-1)$ is a really a reference to $A_2(i-1)$. After making these substitutions, A_1 and A_3 are unused except for defining each other. That is, they occur in a sink loop of the SSA graph, which can therefore be eliminated as dead code. The final code then is this:

$A_0(:) = 0$
$\tau_w A_0(:) = \langle$before the loop$\rangle$
do $i = 1, n$
  $A_2(i) = \ldots$
  $\tau_w A_2(i) = [i]^g$
  if $(i == 1)$ **then**
    $\cdots = \ldots A_2(i) \ldots$
  else
    $\cdots = \ldots A_2(i-1) \ldots$
  end if
end do

Of course there will be a ϕ assignment after this loop merging the values of A_0 with those of A_1, unless that assignment is also able to be eliminated (for instance, by the same sort of reasoning).

We perform this simplification systematically on the original Array SSA form: whenever a reference to an SSA variant A can be proved to be a reference to an element of $web(A)$ that is defined earlier than A is, the appropriate substitution is made. When this leads to one or more ϕ assignments being recognized as dead code, those assignments are then eliminated.

When this is done, it is still true of course that any true dependences in the original source program remain true dependences in the Array SSA form of the program. If we have done a good job, the converse should also be true (or approximately true, anyway)—all true dependence edges in the Array SSA program should now correspond to true dependences in the original program.

There is a certain circumstance (explained below in Section 2.4) in which a wrap-around ϕ assignment that has been removed in this way needs to be reinstated, and reinstating such a wrap-around ϕ assignment in turn may force other ϕ assignments to be reinstated. We won't explain *why* this needs to be done here, (other than to assert that it may be necessary to put the program in dynamic single assignment form) but we simply give an example of how it works. Suppose we have the original code

$A(:) = 0$
do $i = 1, n$
  $A\big(f(i)\big) = \ldots$
  $\cdots = \ldots A\big(f(i)\big) \ldots$
  $A\big(g(i)\big) = \ldots$
  $\cdots = \ldots A\big(g(i)\big) \ldots$
end do

After putting in the initial Array SSA code, we have

$A_0(:) = 0$
$\tau_{\mathrm{w}} A_0(:) = [1_\lambda]$
do $i = 1, n$
$A_1(:) = \phi\big(A_0(:), \tau_{\mathrm{w}} A_0(:); A_5(:), \tau_{\mathrm{w}} A_5(:)\big)$
$\tau_{\mathrm{w}} A_1(:) = \max\big(\tau_{\mathrm{w}} A_0(:), \tau_{\mathrm{w}} A_5(:)\big)$

$A_2\big(f(i)\big) = \ldots$
$\tau_{\mathrm{w}} A_2\big(f(i)\big) = [i]^g$
$A_3(:) = \phi\big(A_2(:), \tau_{\mathrm{w}} A_2(:); A_1(:), \tau_{\mathrm{w}} A_1(:)\big)$
$\tau_{\mathrm{w}} A_3(:) = \max\big(\tau_{\mathrm{w}} A_2(:), \tau_{\mathrm{w}} A_1(:)\big)$

$\cdots = \ldots A_3\big(f(i)\big) \ldots$

$A_4\big(g(i)\big) = \ldots$
$\tau_{\mathrm{w}} A_4\big(g(i)\big) = [i]^g$
$A_5(:) = \phi\big(A_4(:), \tau_{\mathrm{w}} A_4(:); A_3(:); \tau_{\mathrm{w}} A_3(:)\big)$
$\tau_{\mathrm{w}} A_5(:) = \max\big(\tau_{\mathrm{w}} A_4(:), \tau_{\mathrm{w}} A_3(:)\big)$

$\cdots = \ldots A_5\big(g(i)\big) \ldots$
end do

Now a good compiler can be expected to understand that the use of A_3 in the source assignment is really a use of A_2. Similarly, the use of A_5 in the source assignment is really a use of A_4.

This makes the assignments to A_1, A_3, and A_5 dead code, and so they can be eliminated, yielding the following code:

$A_0(:) = 0$
$\tau_{\mathrm{w}} A_0(:) = [1_\lambda]$
do $i = 1, n$
$A_2\big(f(i)\big) = \ldots$
$\tau_{\mathrm{w}} A_2\big(f(i)\big) = [i]^g$

$\cdots = \ldots A_2\big(f(i)\big) \ldots$

$A_4\big(g(i)\big) = \ldots$
$\tau_{\mathrm{w}} A_4\big(g(i)\big) = [i]^g$

$\cdots = \ldots A_4\big(g(i)\big) \ldots$
end do

Now (this will be because the function f is presumably not invertible), we will probably discover in the course of performing the algorithm described in Section 2.4 that we need to reinsert the wrap-around ϕ assignment for $web(A_2)$. This assignment defines A_1 in terms of A_0 and A_5, so we need also to reinsert the ϕ assignment defining A_5 in terms of A_4 and A_3, which in turn forces us to reinsert the ϕ assignment for A_3. We thus wind up with the following code:

$A_0(:) = 0$
$\tau_w A_0(:) = [1_\lambda]$
do $i = 1, n$
$A_1(:) = \phi(A_0(:), \tau_w A_0(:); A_5(:), \tau_w A_5(:))$
$\tau_w A_1(:) = \max(\tau_w A_0(:), \tau_w A_5(:))$

$A_2(f(i)) = \ldots$
$\tau_w A_2(f(i)) = [i]^g$
$A_3(:) = \phi(A_2(:), \tau_w A_2(:); A_1(:), \tau_w A_1(:))$
$\tau_w A_3(:) = \max(\tau_w A_2(:), \tau_w A_1(:))$

$\cdots = \ldots A_2(f(i)) \ldots$

$A_4(g(i)) = \ldots$
$\tau_w A_4(g(i)) = [i]^g$
$A_5(:) = \phi(A_4(:), \tau_w A_4(:); A_3(:); \tau_w A_3(:))$
$\tau_w A_5(:) = \max(\tau_w A_4(:), \tau_w A_3(:))$

$\cdots = \ldots A_4(g(i)) \ldots$
end do

Note that in this code, the use of A_3 in the source statement has still been replaced by a use of A_2, and similarly the source use of A_5 has been replaced by a use of A_4. The ϕ assignments that were reinserted add additional dependences to the Array SSA program that are not present in the original program. These additional dependences, however, will not degrade our analysis.

1.3.2 Simplification of if merges

The initial Array SSA code for a control ϕ merge after an **if** construct looks like this:

$\ldots$
$A_5(:) = \ldots$

if B **then**
$A_6(\vec{k}) = \ldots$
$\tau_w A_6(\vec{k}) = \ldots$
$A_7 = \phi(A_6, \tau_w A_6; A_5, \tau_w A_5)$
$\tau_w A_7 = \max(\tau_w A_6, \tau_w A_5)$
end if
forall $(\vec{j})$

$$A_8(\vec{j}) = \begin{cases} A_7(\vec{j}) & \text{if } \tau_w A_7(\vec{j}) > \tau_w A_5(\vec{j}) \\ A_5(\vec{j}) & \text{otherwise} \end{cases}$$

$$\tau_w A_8(\vec{j}) = \begin{cases} \tau_w A_7(\vec{j}) & \text{if } \tau_w A_7(\vec{j}) > \tau_w A_5(\vec{j}) \\ \tau_w A_5(\vec{j}) & \text{otherwise} \end{cases}$$

end forall

We can, however give an explicit form to the Boolean choice expression in the ϕ function:

...
$A_5(:) = \ldots$

if B **then**
$A_6(\vec{k}) = \ldots$
$\tau_{\mathrm{w}} A_6(\vec{k}) = \ldots$
$A_7 = \phi(A_6, \tau_{\mathrm{w}} A_6; A_5, \tau_{\mathrm{w}} A_5)$
$\tau_{\mathrm{w}} A_7 = \max(\tau_{\mathrm{w}} A_6, \tau_{\mathrm{w}} A_5)$
end if
forall $(\vec{j})$

$$A_8(\vec{j}) = \begin{cases} A_7(\vec{j}) & \text{if } B \\ A_5(\vec{j}) & \text{otherwise} \end{cases}$$

$$\tau_{\mathrm{w}} A_8(\vec{j}) = \begin{cases} \tau_{\mathrm{w}} A_7(\vec{j}) & \text{if } B \\ \tau_{\mathrm{w}} A_5(\vec{j}) & \text{otherwise} \end{cases}$$

end forall

This will be very useful to us; we assume this has been done systematically.

1.3.3 Admissible transformations

In preparation for performing the algorithm of the next chapter, we are going to need to apply some transformations, such as the ones considered above, to the program in order to simplify the SSA overhead as much as possible. In order that these transformations preserve the semantics of the program, we need them to preserve certain values in the program. The values that need to be preserved are the values that could conceivably affect the visible output of the program. We say that such values are *ultimately used*. Ultimate use is defined precisely as follows:

Given a program P, let us temporarily denote by G the directed graph defined as follows: Its nodes fall into two classes:

- the dynamic references in P. A dynamic reference (see also page 36 below) is a static reference together with the time at which the static reference occurs. So each static reference may correspond to more than one dynamic reference.

  There is one exception to this: a dynamic reference of an ordinary variable whose value can be proved to be $\perp$ (i.e., whose value has not been assigned within the program) is not a node in G. This is because $\perp$ values of ordinary variables can never be referenced in a legal program.

  We do not make this exception for τ_{w} variables: as we remarked earlier, values of $\perp$ for τ_{w} variables are meaningful and can be referenced in the program. We can't ignore them.

- expressions that can affect flow of control. In our language these expressions are of two types:

  - the Boolean guards in P. Such Boolean guards are the expressions used in **if** statements and the Boolean choice expressions in ϕ functions.
  - expressions for loop bounds and increments.

  These nodes are also considered dynamically. That is, they are also tagged by the time at which they are evaluated.

As in the previous item, a dynamic instance of a control flow expression whose value can be proved to be $\perp$ is not a node in G.

The edges of G are of the following forms:

- There is an edge from every definition to each of its uses.
- There is an edge from each use in an assignment statement to the reference being defined.
- There is an edge from each reference in a control flow expression to the expression itself.
- There is a (control) edge from each control flow expression to every reference defined or used subsidiary to that expression.

These edges are determined conservatively—a possible use of a definition always gives rise to an edge, for instance. On the other hand, if we can determine that a use is never reached, or that it definitely refers to a different element that that at the definition, then no edge is introduced.

Definition *The value held by the array element $A_i(\vec{k})$ at time t is* ultimately used *if there is a path in the graph G starting from the reference to $A_i(\vec{k})$ at time t and ending at a* ***print*** *statement.*

Of course as we expand the simple language we are considering, there are other conditions under which we will say that an array element is ultimately used. An assignment to an external variable, or a return value from a procedure, for instance, would each determine ultimate use.

It follows immediately from this definition that if some element $\tau_{\mathrm{w}} A_i(\vec{k})$ at time t is ultimately used, then the corresponding element $A(\vec{k})$ at time t is also ultimately used. This is because the only use of $\tau_{\mathrm{w}} A_i(\vec{k})$ is to guard references to $A(\vec{k})$.

As noted, this definition of ultimate use is conservative. A variable is considered to be ultimately used unless the compiler can prove that it is not.

Our notion of ultimate use is similar but not identical to the notion of live code as defined in Section 7.1 of Cytron et al. (1991).

Now the transformations we have just been considering have the following characterization:

1. Suppose A_i and A_j are ordinary variables in the same web and suppose that $A_i(\vec{k})$ is used in some statement at some time t.
   - If the value held by $A_i(\vec{k})$ at time t is ultimately used, and if it can be proved that at time t, $\tau_{\mathrm{w}} A_i(\vec{k}) = \tau_{\mathrm{w}} A_j(\vec{k})$ (and therefore that $A_i(\vec{k}) = A_j(\vec{k})$ as well), then the use of $A_i(\vec{k})$ may be replaced by a use of $A_j(\vec{k})$. When this is done, any corresponding use of $\tau_{\mathrm{w}} A_i(\vec{k})$ is also replaced by $\tau_{\mathrm{w}} A_j(\vec{k})$.
   - If the value held by $A_i(\vec{k})$ at time t is *not* ultimately used, then it is permissible to replace the use of $A_i(\vec{k})$ by a use of $A_j(\vec{k})$ and to replace any corresponding use of $\tau_{\mathrm{w}} A_i(\vec{k})$ by a corresponding use of $\tau_{\mathrm{w}} A_j(\vec{k})$ *provided* that $\tau_{\mathrm{w}} A_j(\vec{k}) \leq \tau_{\mathrm{w}} A_i(\vec{k})$ at time t.

2. A Boolean expression in a ϕ function may be replaced by another expression with the same value.

3. An argument to a ϕ function can be eliminated if it can be proved that doing so does not change the value of the ϕ function.

4. In conjunction with actions taken because of items 2, 3, and 4, dead code may be eliminated. When dead code is eliminated, it is replaced by null statements.

5. Any action as described above is permitted only if it preserves the property described in Lemma 1.7 (page 22).

6. Any action as described above is permitted only if it preserves the dynamic control flow of the program.

Definition *An* admissible transformation *is a program transformation described by any or all of the first 4 items above, and maintaining the constraints described in items 5 and 6.*

A program which is the result of applying only admissible transformations to a program in initial Array SSA form is called an admissible program*,* provided *that it also satisfies the following constraint:*

If the right-hand side of an assignment statement contains a τ_{w} variable (so in particular this is a statement that is a ϕ assignment), then it contains a τ_{w} variable for each ordinary variable on the right-hand side.

This last constraint is introduced only to help in the final proof in Chapter 2. It seems to be satisfied in every case we have seen so far.

In particular, a program in initial Array SSA form is an admissible program. (That is, the "null optimization" is trivially an admissible transformation.)

Note that an admissible transformation preserves the control flow graph of the program. In particular, no statements are moved from one place in the program to another, no loops are interchanged, and so on. In fact, constraint 6 above shows that an admissible program is iteratively similar to the original program.

The next lemma has as a consequence that that an admissible program is semantically equivalent to the initial Array SSA program from which it was derived, in the sense that it computes the same ultimately used values for every ordinary variable. As a result, the program has the same visible behavior. We continue to say that an admissible program is in Array SSA form.

1.8 Lemma *Let P be a program, and let P' be the program which is the result of applying an admissible transformation to P.*

If $\tau_{\mathrm{w}} A'(\vec{k})$ is the value of a τ_{w} variable for some element at time t in P', and if $\tau_{\mathrm{w}} A(\vec{k})$ is the corresponding value at the same time t in P, then

$$\tau_{\mathrm{w}} A'(\vec{k}) \leq \tau_{\mathrm{w}} A(\vec{k})$$

If the value held by $\tau_{\mathrm{w}} A(\vec{k})$ at time t is ultimately used, then

$$\tau_{\mathrm{w}} A'(\vec{k}) = \tau_{\mathrm{w}} A(\vec{k})$$

Similarly, if $A'(\vec{k})$ and $A(\vec{k})$ are values of ordinary variables in P' and P respectively at the same time t, and if the value held by $A(\vec{k})$ at time t is ultimately used, then

$$A'(\vec{k}) = A(\vec{k})$$

PROOF. It is sufficient to prove that this lemma is true when a transformation consisting of a single action described by one of items 2, 3, 4, or 5 is applied, for then it will remain true when additional such transformations are applied, by induction.

The proof for one such transformation proceeds by induction on the sequence of computations executed in the program.

Certainly for every array element at the start of the program (i.e., before any statements have been executed), $\tau_{\mathrm{w}} A'(\vec{k}) = \tau_{\mathrm{w}} A(\vec{k})$.

An assignment to $\tau_{\mathrm{w}} A(\vec{k})$ or to $A(\vec{k})$ in P may be dropped as dead code in P'. This cannot affect any other values in P'. Of course this cannot happen if $\tau_{\mathrm{w}} A(\vec{k})$ (respectively, $A(\vec{k})$) is ultimately used.

If then the assignment to $A(\vec{k})$ or to $\tau_{\mathrm{w}} A(\vec{k})$ is executed in P', then there are two possibilities:

The assignment is a source assignment If the assignment is a source assignment to $A(\vec{k})$, and if $A(\vec{k})$ is ultimately used, then all the references used in the assignment statement are ultimately used and hence by the inductive hypothesis have the same values in P' as they do in P.

If the assignment is a source assignment to $\tau_{\mathrm{w}} A(\vec{k})$ then $\tau_{\mathrm{w}} A(\vec{k})$ is set equal to the current time, which is the same in P' as in P, so $\tau_{\mathrm{w}} A'(\vec{k}) = \tau_{\mathrm{w}} A(\vec{k})$.

The assignment is a ϕ assignment If the assignment is a ϕ assignment to $\tau_{\mathrm{w}} A_i(\vec{k})$, let $\tau_{\mathrm{w}} A_j(\vec{k})$ be any one of the τ_{w} variable arguments to the ϕ function. One of the following must be true:

- $\tau_{\mathrm{w}} A'_j(\vec{k})$ occurs as an argument to the ϕ function in P'. In this case, by the inductive hypothesis, $\tau_{\mathrm{w}} A'_j(\vec{k}) \leq \tau_{\mathrm{w}} A_j(\vec{k})$. Further, if $\tau_{\mathrm{w}} A_j(\vec{k})$ is ultimately used, then again by the inductive hypothesis, $\tau_{\mathrm{w}} A'_j(\vec{k}) = \tau_{\mathrm{w}} A_j(\vec{k})$.
- The argument $\tau_{\mathrm{w}} A_j(\vec{k})$ is replaced in P' by $\tau_{\mathrm{w}} A'_k(\vec{k})$. When this transformation is made, we must have $\tau_{\mathrm{w}} A'_k(\vec{k}) \leq \tau_{\mathrm{w}} A_j(\vec{k})$, by item 2 in the definition of admissible transformations. Further, if $\tau_{\mathrm{w}} A_j(\vec{k})$ is ultimately used, then item 2 mandates that $\tau_{\mathrm{w}} A'_j(\vec{k}) = \tau_{\mathrm{w}} A_j(\vec{k})$.
- The argument $\tau_{\mathrm{w}} A_j(\vec{k})$ is dropped as an argument to the ϕ function, without replacement. Of course this cannot happen if $\tau_{\mathrm{w}} A_j(\vec{k})$ is ultimately used.

Since one of these three cases must apply to each argument of the ϕ function in P, the value of the ϕ function in P' must be $\leq$ its value in P. Further, if $\tau_{\mathrm{w}} A_i(\vec{k})$ is ultimately used, then at least one of its arguments (the argument that is selected) must be ultimately used, and so $\tau_{\mathrm{w}} A'_i(\vec{k}) = \tau_{\mathrm{w}} A_i(\vec{k})$.

If on the other hand the assignment is a ϕ assignment to $A_i(\vec{k})$, and if $A_i(\vec{k})$ is ultimately used, then it must be that the τ_{w} variable corresponding to the variant of $A_i(\vec{k})$ which is assigned to $A_i(\vec{k})$ by the ϕ function is also ultimately used. The same reasoning as above then shows that the same value will be picked for $A'_i(\vec{k})$ as for $A_i(\vec{k})$. □

1.4 General properties of τ_{w} and τ_{v} variables

The properties of τ_{w} and τ_{v} variables that we have derived above in Section 1.2.8, as well as some others, continue to hold in programs in initial Array SSA form, and indeed in admissible programs as well:

1.4.1 Properties that are true for programs in initial array SSA form

Lemma 1.4 and Corollary 1.6 state properties of τ_{v} and τ_{w} variables that hold true for naive Array SSA programs. These properties remain true when those programs are transformed into initial Array SSA form. This is simply because the values of the τ_{v} and τ_{w} variables are not changed in this transformation. For ease of reference we restate those results here:

1.9 Lemma *If A and B are in the same web, and if at some point in the execution of the program $\tau_{\mathrm{w}}A(\vec{k}) < \tau_{\mathrm{w}}B(\vec{k})$ for some $\vec{k}$, then (at that same point)*

$$\tau_{\mathrm{w}}A(\vec{k}) \leq \tau_{\mathrm{v}}A(\vec{k}) < \tau_{\mathrm{w}}B(\vec{k}) \leq \tau_{\mathrm{v}}B(\vec{k})$$

1.10 Lemma *In a program in initial Array SSA form, the following properties hold:*

1. *If $A_i(\vec{k})$ is assigned at a source assignment, then the (conceptually simultaneous) assignment to $\tau_{\mathrm{w}}A_i(\vec{k})$ assigns a value in global iteration space greater than the current value of every other $\tau_{\mathrm{w}}A_j(\vec{k})$ for each A_j in web(A_i).*
2. *If on the other hand A_i is defined by a ϕ assignment, then for each execution of that assignment statement, the (conceptually simultaneous) assignment to $\tau_{\mathrm{w}}A_i$ causes $\tau_{\mathrm{w}}A_i$ to be elementwise at least as great as the current value of every other $\tau_{\mathrm{w}}A_j$ for each A_j in web(A_i).*

1.4.2 Properties that are true for admissible programs

First, we have some properties of admissible programs that are true without qualification:

1.11 Lemma *1. For any ordinary array element $A_i(\vec{k})$, the values of $\tau_{\mathrm{v}}A_i(\vec{k})$ and $\tau_{\mathrm{w}}A_i(\vec{k})$ are both non-decreasing in time.*

2. At any point in the course of execution of the program, the value in $\tau_{\mathrm{w}}A_i(\vec{k})$ is $\leq$ the time at which the value currently in $A_i(\vec{k})$ was assigned. That is, at any point in the course of execution of the program,

$$\tau_{\mathrm{w}}A_i(\vec{k}) \leq \tau_{\mathrm{v}}A_i(\vec{k})$$

3. For any ordinary array element $A(\vec{k})$,

$$\tau_v \tau_w A(\vec{k}) \equiv \tau_v A(\vec{k})$$

PROOF. 1. That $\tau_v A_i(\vec{k})$ is non-decreasing follows immediately from the definition.

If $\tau_w A_i(\vec{k})$ is defined at a source statement, then successive assignments to it can only increase its value. If it is defined at a ϕ assignment, then it is defined as a max of other τ_w variables, and by induction it follows that its value is also non-decreasing.

2. This property is true by definition in initial Array SSA programs. No admissible transformation changes this relation.

3. This just says that $\tau_w A(\vec{k})$ was last assigned at the same time that $A(\vec{k})$ was, which is true by definition. □

In addition, as already noted, we always have $\tau_w A \leq \tau_v A$. We also note that

- $P(\tau_w A)$ must be a source statement (i.e., not a ϕ assignment).
- $\tau_w A = \tau_v A$ iff $P(\tau_v A)$ is a source statement.

The rest of the properties in this section are true with some qualification about one or more elements being ultimately used:

1.12 Theorem *Let P be an original source program, and P' be an admissible version of the Array SSA form of P. (That is, P' is the result of applying admissible transformations to the initial Array SSA form of P.)*

If A is a variable in P, and if A_i is an ordinary variable in P' which is in web(A), and if the value in P' held by $A_i(\vec{k})$ at time t is ultimately used, then it is equal to the value held in P at the same time t by $A(\vec{k})$.

PROOF. Defining P to be the original program P, as before, we have shown that this theorem is true for the program P_w in initial Array SSA form (even without the qualification of ultimate use).

That this property persists when admissible transformations are made to P_w then follows from Lemma 1.8 (page 30). □

Lemma 1.9 remains true for admissible programs provided that $\tau_w A(\vec{k})$ and $\tau_w B(\vec{k})$ are both ultimately used:

1.13 Theorem *If A and B are in the same web, and if at some point in the execution of the program $\tau_w A(\vec{k}) < \tau_w B(\vec{k})$ for some $\vec{k}$, where $\tau_w A(\vec{k})$ and $\tau_w B(\vec{k})$ are both ultimately used, then (at that same point)*

$$\tau_w A(\vec{k}) \leq \tau_v A(\vec{k}) < \tau_w B(\vec{k}) \leq \tau_v B(\vec{k})$$

PROOF. We know that each admissible transformation preserves the values of $\tau_w A(\vec{k})$ and $\tau_w B(\vec{k})$. Therefore the inequalities are true in the original program P. Further, such a transformation cannot increase the value of $\tau_v A(\vec{k})$, and by Lemma 1.11(2), the two outer inequalities are always true. So all the inequalities remain true in the transformed program P'. □

Lemma 1.10 remains true for admissible programs provided that in the second part we restrict ourselves to elements that are ultimately used:

1.14 Theorem *In an admissible program, the following properties hold:*

1. *If $A_i(\vec{k})$ is assigned at a source assignment, then the (conceptually simultaneous) assignment to $\tau_w A_i(\vec{k})$ assigns a value in global iteration space greater than every other $\tau_w A_j(\vec{k})$ for each A_j in $web(A_i)$.*
2. *If on the other hand A_i is defined by a ϕ assignment, then for each execution of that assignment statement, if $\tau_w A_i(\vec{k})$ is an element of $\tau_w A_i$ that is ultimately used, the (conceptually simultaneous) assignment to $\tau_w A_i(\vec{k})$ causes $\tau_w A_i(\vec{k})$ to be at least as great as every other $\tau_w A_j(\vec{k})$ for each A_j in $web(A_i)$.*

PROOF. 1. This remains true for the same reason as before: the value assigned to $\tau_w A_i(\vec{k})$ is the time of execution of the statement which is certainly greater than any previous time that could have been assigned or propagated in the program up to that point.

2. We know that this is true for programs in initial Array SSA form. We know by Lemma 1.8 (page 30) that since $\tau_w A_i(\vec{k})$ is ultimately used, its value is the same as the value it started out with in the original initial Array SSA form of the program. And by the same lemma, all the other values $\tau_w A_j(\vec{k})$ are no greater than they were in the original initial Array SSA form. Therefore the property persists in the transformed program. □

To see why we need the stipulation that $\tau_w A_i(\vec{k})$ be ultimately used in part 2 of the theorem, consider the code in Figure 1.7.

$A_0(:) = \ldots$
do $i = 1, 2 * n - 1, 2$
  $A_1(i) = \ldots$
  $A_2 = \phi(A_0, \tau_w A_0; A_1, \tau_w A_1)$
end do
$A_3 = \phi(A_2, \tau_w A_2; A_0; \tau_w A_0)$
$A_4(2) = 0$
$A_5 = \phi(A_4, \tau_w A_4; A_3, \tau_w A_3)$
do $i = 1, n$
  $\ldots$
  $\cdots = \ldots A_5(2 * i) \ldots$
  $\ldots$
end do

Figure 1.7: Code that illustrates why we need to consider the notion of ultimate use. The assignments to τ_w variables have been suppressed for clarity; they occur in the actual code.

We, and hopefully our compiler, can figure out that the use of $A_5(2 * i)$ must refer either to A_0 or to A_4—that is, the use of A_3 and $\tau_w A_3$ in the definition of A_5 are really resolved to uses of A_0 and $\tau_w A_0$. Thus, we can replace the definition of A_5 by

$$A_5 = \phi(A_4, \tau_w A_4; A_0, \tau_w A_0)$$

When we do this, note that subsequent to the definition of A_5, we have

$$\tau_w A_5(1) = \tau_w A_0(1) < \tau_w A_1(1)$$

which might seem like a counterexample to Theorem 1.14, except that (assuming there is no other code using A_5) $\tau_w A_5(1)$ is not ultimately used. (In fact, in this example, it's not used at all.)

Next we exhibit a relation between $\tau_v A$ and $\tau_w A$ that holds for ordinary variables A in any admissible program. As usual, we denote the program by P, and the mapping from global iteration space to program statements is thus denoted by $t \mapsto P(t)$. (We introduced this notation on page 7.) So $P(\tau_w A)$ is the statement at which the value ultimately assigned to A at time $\tau_v A$ enters $web(A)$.

1.15 Theorem *In an admissible program, the following properties hold:*

1. *Suppose S is a source statement that is executed at some time t. If the array element $A_i(\vec{k})$ occurs as a use in S at time t, and if the value held by $\tau_w A_i(\vec{k})$ at time t is ultimately used, then $\tau_w A_i(\vec{k})$ is greater than or equal to all other $\tau_w A_j(\vec{k})$ for all A_j in $web(A_i)$.*

2. *Suppose S is a ϕ assignment defining $\tau_w A_i$. If $\tau_w A_i(\vec{k})$ is defined by S at time t and is ultimately used, then one of the arguments to the ϕ function in S has a value that is maximal among all the values $\left\{\tau_w A_j(\vec{k}) : A_j \in web(A_i)\right\}$ at time t.*

PROOF. 1. First of all, this is true in a program in initial Array SSA form for the following reason: By Lemma 1.10, $\tau_w A_i(\vec{k})$ is the maximum value of $\tau_w A_j(\vec{k})$ for all A_j in $web(A_i)$.

Lemma 1.8 (page 30) then shows that $\tau_w A_i(\vec{k})$ maintains its value in any admissibly transformed program, while all other values $\tau_w A_j(\vec{k})$ are no greater than their values in the original initial Array SSA program, so this property persists in any admissible program.

2. This follows immediately from Theorem 1.14: since the assignment to $\tau_w A_i(\vec{k})$ causes $\tau_w A_i(\vec{k})$ to be at least as great as every other $\tau_w A_j(\vec{k})$ for each A_j in $web(A_i)$, some element $\tau_w A_j(\vec{k})$ in the web whose value is greatest just prior to execution of S must occur as an argument to the ϕ function in S. □

Chapter 2

Weak dynamic single assignment form

2.1 Overview

We start by stating the problem we are addressing, and giving an overview of our approach to it. Precise definitions come afterwards.

We start with an admissible program P in array static single-assignment form. (That is, admissible transformations may have been applied to P after it was put in initial Array SSA form.) Any computed value in the original program that is ultimately used is computed to be the same in P. Values that were not ultimately used in the original program may be computed differently (or not at all) in P. From now on, however, we can disregard this consideration. We simply take the admissible program P as our starting point, and we regard whatever it computes as essential.

In general, we will denote a reference to a scalar or an array element simply by R. However, there are two distinct kinds of references that we have to deal with:

- The *static reference* (i.e., the reference produced by putting the program in array static single assignment form). We sometimes refer to this as R_s when we need to be precise.
- A *dynamic reference* corresponding to this static reference. In general, the static reference R_s may occur dynamically more than once during the execution of the program. Each such dynamic reference can be referred to as R_d to be precise. Note that two dynamic references corresponding to the same static reference may refer to the same or to different array elements.

Definition *A program is in* dynamic single assignment form *if no scalar or array element is assigned to more than once in the course of execution of the program.*

Below we will define a notion of *weak* dynamic single assignment form. When we want to emphasize the distinction, we may refer to dynamic single assignment form as *strong* dynamic single assignment form.

Our ultimate aim is to find a transformed program P' such that

1. Each dynamic reference R_d in P has a well-defined corresponding dynamic reference R'_d in P'. (The map from R_d to R'_d may be many-to-one.)
2. P' is equivalent to P in the following sense: If a dynamic reference R_d in P to an ordinary array element or ordinary scalar variable has the value v, then the corresponding reference R'_d in P' has that same value.
3. ("P' is not too small.") P' has the dynamic single assignment (DSA) property.
4. ("P' is not too large.") P' is a minimal extension of P, in a sense to be described below.

We can always achieve the first three conditions by expanding each data reference by its iteration space dimensions. Then we have to find some way of making sure that values fetched in the transformed program correspond to the ones originally stored. We refer to this as the *maximal expansion.*

When we do this, however, there may well be dimensions that are redundant. For instance, suppose we start with the loop nest

do $j = 1, m$
  do $i = 1, n$
    $A(i) = i$
  end do
end do

(where we have suppressed the initial Array SSA overhead).

A maximal expansion would look like this:

do $j = 1, m$
  do $i = 1, n$
    $A(i)[i, j] = i$
  end do
end do

However, the added iteration dimensions are really not needed: Since each assignment to A assigns a value that depends only on i, the minimal dynamic single assignment form would really be one in which no additional dimensions were added and in which the j loop is eliminated entirely:

do $i = 1, n$
  $A(i) = i$
end do

However, we will not achieve this DSA form in one step. Rather, we will first perform a transformation that simply changes the shape of each data object to its "natural form", leaving the iteration structure of the program unchanged. This will leave the original loop nest unchanged. Now this is not in DSA form, because each array element $A(i)$ is assigned to m times. However, it does have the property that each assignment to $A(i)$ has the same value. We say that a program with this property is in *weak dynamic single assignment form*:

Definition *1. A program is in weak dynamic single assignment (weak DSA) form iff*

- *it is in SSA form*[1]*, and*

[1] Strictly speaking, this is not a requirement. For instance, code such as this

- *any two dynamic assignments to a scalar or an array element assign the same value.*

2. *A program is in weak dynamic single assignment form* up to time t *iff*

   (a) *it is in SSA form, and*

   (b) *any two dynamic assignments to a scalar or an array element occurring at or before time t assign the same value.*

It is important to note that ordinary variables act rather differently from τ_w variables in programs in weak DSA form. In either kind of program, an ordinary variable can be referenced only after it has been defined. Therefore, it can have only one value that is actually used in the program. For a τ_w variable, however, the value $\perp$ is meaningful, and so in a weak DSA program, a τ_w variable may have two values that are used in the program—in such a case, the first value must be $\perp$. We do not regard assignments of $\perp$ to a τ_w variable as violating the weak DSA condition. (And we don't regard "assignments" of $\perp$ to ordinary variables as assignments at all.)

The algorithm we describe in this chapter starts with a program in Array SSA form and produces a semantically equivalent program in weak DSA form. The transformed program P' produced by this algorithm satisfies the four conditions listed above with two changes:

- Condition 1 is strengthened: the map from R_d to R'_d is one-to-one. This is because we are not changing the iterative structure of the program—just the shape of the data objects in the program.
- Condition 3 is weakened: P' has the weak DSA property.

In future work we will show how to subsequently change this into a program in DSA form by restructuring the program and eliminating redundant loops.

In the example above, an iteration dimension, which might have been added, was found to be unnecessary. It is also possible that a data dimension (which is already present in the source program) may be deleted because it is redundant. For instance, if the original program were written like this:

```
do j = 1, n
   do i = 1, n
      A(i, j) = i
   end do
end do
```

then just as before, we could conclude that the data dimension d_2 is unnecessary, and delete it, again yielding the weak DSA form

```
do i = 1, n
   if b(i) then
      s(i) = 2
   else
      s(i) = 3
   end if
end do
```

is in dynamic single assignment form without being in SSA form, since each element of the array s is assigned to exactly once. Nevertheless, since our algorithm starts with a program in Array SSA form and maintains this property, there is no harm in asserting this.

do $j = 1, n$
  do $i = 1, n$
    $A(i) = i$
  end do
end do

In deciding whether a dimension is needed, we have to consider its role:

- Some data space dimensions are needed, as they are in the original program, to ensure that distinct values are stored in places from which they can be retrieved.

  In the examples above, the subscript expressions are so trivial that this consideration is hardly apparent; but it is easy to see that complex subscript expressions add meaning to a program that cannot simply be discarded.

  The key to dealing with this is the relation between a definition and its uses. For instance, (again ignoring the initial Array SSA code) if the original program looked like this:

  do $j = 1, n$
    do $i = 1, n$
      $A(i, i + j - 2) = i$
      $\cdots$
      $\cdots = \ldots A(i, j) \ldots$
    end do
  end do

  Then it's intuitively pretty clear that both dimensions of A are needed. On the other hand, if the original source looked like this:

  do $j = 1, n$
    do $i = 1, n$
      $A(i, i + j - 2) = i$
      $\cdots$
      $\cdots = \ldots A(i, i + j - 2) \ldots$
    end do
  end do

  then again only 1 dimension is needed[2].

- Some iteration space dimensions are needed to ensure that the program is in weak dynamic single-assignment form. That is, overwriting an element of an existing data object gives rise to an expansion of that object over an iteration space dimension. Equivalently, iteration space dimensions are used to distinguish different values written to the same data location at different times in the execution of the program.

We call the final set of dimensions that we arrive at for a data object the *set of contributing dimensions* of that object. We regard the set of contributing dimensions as describing the "natural shape" of the object. It takes into account essential dimensions in the shape originally given in the program P and also any reuse that may occur during execution of the program. That is, the set of contributing dimensions may include some (but possibly not all) of the original dimensions from

[2]Actually, our algorithm will give only one dimension to the defined object in either case. In the first example, however, the variable assigned to at the immediately following definition ϕ assignment will have two dimensions.

the declared rank of the object, and may also include some (but possibly not all) dimensions that come from the iteration space, which represent reuse of elements of the original object.

In many cases, even when a minimal transformed program P' can be effectively found, it may not be unique. For this reason, it is important to note that a dimension, in and of itself, is not a contributing dimension. There may be more than one set of contributing dimensions for an array, and so a particular dimension may belong to one set but not the other.

Our practice will be, when given a choice between data space and iteration space, to favor the data space. That is, we will keep a data space dimension rather than eliminating it and introducing an iteration space dimension. There are two reasons for this—a moral reason and a practical reason:

moral reason: This leaves the program looking as much as possible like the original program. The original shapes of arrays in the program had some meaning for the programmer. In general, it makes sense to try to preserve this meaning rather than to ignore it.

practical reason: Added iteration dimensions lead (see Section 2.3.1 below) to the introduction of τ_{v} variables into the code. We can always eliminate explicit references to τ_{v} variables (see Section 2.3.3), but it may be that we have to make the code more complicated in order to do this. So it is better to add as few iteration dimensions as possible.

These two reasons are actually quite closely related.

2.2 The space of values at a static reference

By $\mathcal{M}$ let us denote the direct sum of data space and local iteration space. Since we regard $\mathcal{D}$ and $\mathcal{I}$ simply as sets of basis elements, it is natural to write $\mathcal{M} = \mathcal{D} \cup \mathcal{I}$.

For each static reference R_s, the values of the subscripts together with the (dynamic) point in local iteration space determine the element referenced and its value. That is to say, the element referenced and its value at a dynamic reference corresponding to R_s are determined by its position in $\mathcal{D}$ and in $\mathcal{I}$, or simply in $\mathcal{M}$.

Sometimes we do not care about the elements being referenced by R_s; we care only about the set of values assumed by R_s during execution of the program. This set of values is completely parametrized by the local iteration space $\mathcal{I}$. And it may even be that a subspace of the local iteration space is all that is necessary to parametrize these values. For instance, in Figure 2.1, the local iteration space is $\{i_1, i_2\}$. But the set of values can be parametrized simply by $\{i_1\}$. For this reason, it is convenient for us to define the notion of a *valid iteration subspace* $\mathcal{S}$ at a (static) reference R; this will be a subspace of the local iteration space $\mathcal{I}$ at R such that the value of R is determined by the indices in $\mathcal{S}$.

Definition *$\mathcal{S}$ is a valid iteration subspace at a static reference R_s iff all dynamic references R_d at the static reference R_s having the same values of the indices $\{I(i) : i \in \mathcal{S}\}$ at that reference have the same value.*

More generally, $\mathcal{S}$ is a valid iteration subspace at an expression e iff all evaluations of e at times having the same values of the indices $\{I(i) : i \in \mathcal{S}\}$ have the same value.

So in particular, $\mathcal{I}$ is itself a valid iteration subspace.

```
do j = 1, n
   do i = 1, n
      A(i, j) = i
   end do
end do
```

Figure 2.1: A reference to a 2-dimensional array with a 1-dimensional valid iteration subspace.

In Figure 2.2, $\{i_1\}$ and $\{i_2\}$ are both valid iteration subspaces. This shows that a minimal valid iteration subspace is not necessarily unique.

```
do j = 1, n
   do i = 1, n
      ...
      if (i == j) then
         A(i, j) = i
      end if
      ...
   end do
end do
```

Figure 2.2: A reference to a 2-dimensional array with two different 1-dimensional valid iteration subspaces.

It is useful to keep this distinction in mind:

- The local iteration space at a static reference R_s parametrizes the dynamic references R_d corresponding to R_s.
- Each valid iteration subspace at a static reference R_s parametrizes the set of values assumed by the dynamic references R_d corresponding to R_s.

2.3 Contributing dimensions

2.3.1 Rewriting corresponding to an arbitrary set of dimensions

For each data object A, let $\mathcal{I}$ denote the local iteration space at the (static single) assignment to A. Say $\mathcal{I}$ has rank n. Let $\mathcal{D}$ be the data space of A, and say it has rank m.

A subspace of $\mathcal{M}$ can be denoted by $\mathcal{C} = \{d_{j_1}, \ldots, d_{j_p}, i_{k_1}, \ldots, i_{k_q}\}$, where $\{d_{j_r} : 1 \leq r \leq p\}$ is a subset of the data space dimensions $\mathcal{D}$ and $\{i_{k_s} : 1 \leq s \leq q\}$ is a subset of the local iteration space dimensions $\mathcal{I}$.

Each such subspace $\mathcal{C}$ determines a rewriting of the program P, creating a transformed program P', in the following way:

P' is the same as P with the exception that the array A is replaced consistently with an array $A^{\mathcal{C}}$, as follows:

1. $A^{\mathcal{C}}$ has rank $p+q$.
2. Each reference $A\big(\sigma(d_1),\ldots,\sigma(d_m)\big)$ to A is rewritten as

$$A^{\mathcal{C}}\big(\sigma(d_{j_1}),\ldots,\sigma(d_{j_p})\big)\big[\tau_{\mathrm{v}}A\big(\sigma(d_1),\ldots,\sigma(d_m)\big)_{k_1},\ldots,\tau_{\mathrm{v}}A\big(\sigma(d_1),\ldots,\sigma(d_m)\big)_{k_q}\big]$$

As usual, the notation

$$\tau_{\mathrm{v}}A\big(\sigma(d_1),\ldots,\sigma(d_m)\big)_{k_1}$$

refers to the k_1^{th} component of the q-tuple $\tau_{\mathrm{v}}A\big(\sigma(d_1),\ldots,\sigma(d_m)\big)^{\mathrm{loc}}$.

Note that because of the way we have defined $\sigma(i_j)$ (page 19), the rewritten reference to A can also be represented as

$$A^{\mathcal{C}}\big(\sigma(d_{j_1}),\ldots,\sigma(d_{j_p})\big)\big[\sigma(i_{k_1}),\ldots,\sigma(i_{k_q})\big]$$

This is the reason we defined σ on iteration indices the way we did.

In this way we replace A by a new array $A^{\mathcal{C}}$ whose data space is $\mathcal{C}$. This rewriting, while it changes the form of the references to A, does not change the control structure of the program.

More precisely, this rewriting changes the original Array SSA program only in the following respects:

- It may remove some data subscripts from array references.
- It may add some iteration subscripts to array references.

The rewriting leaves everything else in the program alone. In particular, no modifications are made to any of the following:

- The static forms of loops, loop nesting, if constructs, and in general anything that could change the control flow graph.
- Computations.
- Array names.

The term "rewriting", when used in this chapter, always has the meaning defined above.

While this rewriting may at first look somewhat strange and intimidating, it will become evident that it is really quite natural. On page 47 we give an example of this rewriting, and also exhibit the fact that in practice it is capable of being simplified quite a bit.

Note, by the way, that if A is actually a τ_{w} variable (say $A = \tau_{\mathrm{w}}B$), then the rewriting would introduce variables of the form $\tau_{\mathrm{v}}\tau_{\mathrm{w}}B$. As we have seen in Lemma 1.11 (page 32), such a variable can be simply written as $\tau_{\mathrm{v}}B$, and that is what we do.

Note that the rewritten program is structurally similar to the original program. The two programs may not be iteratively similar, however. If $\mathcal{C} = \varnothing$ for each array, for instance, then in the rewritten program each array becomes a scalar, and in any but the most trivial programs one could expect the actual path of execution to differ. This would obviously be true, for instance, if any Boolean expression in an **if** construct involved array references.

If P and P' *are* iteratively similar, however, there is a 1-1 correspondence between the dynamic references in P and P'.

2.3.2 Valid sets of contributing dimensions

By a *valid set of contributing dimensions* for a variable A we mean a set of dimensions that intuitively captures the natural shape of A:

Definition *A valid set of contributing dimensions for a variable A is a subspace*

$$\mathcal{C} = \{d_{j_1}, \ldots, d_{j_p}, i_{k_1}, \ldots, i_{k_q}\} \subseteq \mathcal{M}$$

such that when we rewrite the program P to get a program P' by replacing each reference to A by a reference to $A^{\mathcal{C}}$ as above,

Property A: *P' has the weak dynamic single assignment property for $A^{\mathcal{C}}$: Any two dynamic assignments to an element of $A^{\mathcal{C}}$ in P' assign the same value.*

Property B: *P' and P are iteratively similar.*

Property C: *Each dynamic reference in P to an element of an array has the same value as the corresponding dynamic reference in P'. (By "corresponding", here we mean "occurring at the same time in P'".)*

Note that this must be true for references to *all* the variables in P. Only references to A are rewritten, however, and for those references, this property can be stated as follows:

Each dynamic reference $A\big(\sigma(d_1), \ldots, \sigma(d_m)\big)$ in P at a point t in its local iteration space has the same value as the corresponding reference

$$A^{\mathcal{C}}\big(\sigma(d_{j_1}), \ldots, \sigma(d_{j_p})\big)\big[\tau_{\mathrm{v}}A\big(\sigma(d_1), \ldots, \sigma(d_m)\big)_{k_1}, \ldots, \tau_{\mathrm{v}}A\big(\sigma(d_1), \ldots, \sigma(d_m)\big)_{k_q}\big]$$

at the same point t in P'.

Thus, unlike a valid iteration subspace, which is an attribute of a reference, and which may differ at different references to the same object A, a valid set of contributing dimensions is an attribute of the object A itself.

In any case, given a candidate set of dimensions $\mathcal{C}$, we need in principle to check the above properties A, B, and C to see whether or not $\mathcal{C}$ is a valid set of contributing dimensions. It turns out, however, that these properties are not independent; in fact, A implies B and C for all ordinary variables, and A also implies B and C for τ_{w} variables under an additional condition. To prove this, we first prove a lemma which contains the kernel of the argument:

2.1 Lemma *Let program P be rewritten to become program P' using a set of dimensions $\mathcal{C}$ for the array A in P. Let us assume that under this rewriting,*

- *P' is iteratively similar to P up through some time t.*
- *Every assignment before time t assigns the same value in P' as in P.*
- *P' is in weak dynamic single assignment form with respect to the variable A for all times before t.*

Further, let R_{def} and R_{use} denote dynamic references to fixed elements of A in program P, and let R'_{def} and R'_{use} denote dynamic references to $A^{\mathcal{C}}$ in program P'. We assume that R_{use} and R'_{use} both occur at time t, and that the specific relations between these four references are given in Figure 2.3. Then under the conditions stated in that figure, R'_{use} has the same value as R'_{def}.

Program P	Program P'
R_{def}: a value is assigned to an element of A. No further assignment is made to that element of A before R_{use}	R'_{def}: a value is assigned to an element of $A^{\mathcal{C}}$. R'_{def} is at the same point in global iteration space in program P' as R_{def} is in program P.
R_{use}: retrieves the value of that element of A that was assigned at R_{def}. R_{use} is referenced at time t.	R'_{use}: this is the dynamic reference that is at the same point t in global iteration space in program P' as R_{use} is in program P. Since the executed global iteration space in program P' is the same as that in program P up through time t, we know that R'_{use} occurs later than R'_{def}.

Figure 2.3: Four dynamic references used in the proof of Lemma 2.1.

Remark This lemma is one step in the process of showing that actually the values of all four references are equal. Of course we already know by assumption that R_{use} has the same value as R_{def}.

PROOF. 1. We first show that the subscripts in R'_{use} have the same values as those in R'_{def}.

Let us use the notation $v(d_k)$ or $v(i_k)$ to refer to the value of $\sigma(d_k)$ or $\sigma(i_k)$ at a particular time. (The time will always be understood from the context.)

We can write

$$R_{\text{def}} = A\big(\sigma^{\text{def}}(d_1), \ldots, \sigma^{\text{def}}(d_m)\big)$$
$$R_{\text{use}} = A\big(\sigma^{\text{use}}(d_1), \ldots, \sigma^{\text{use}}(d_m)\big)$$

Since R_{def} and R_{use} refer to the same element of A, we know that the values of their corresponding subscripts are equal, and so we can write them simply as $v(d_i)$:

$$v(d_i) = v^{\text{def}}(d_i) = v^{\text{use}}(d_i) \quad \text{for all } i$$

Further, we can say that R_{def} occurs at the point $[v(i_1), \ldots, v(i_n)]$ in its local iteration space in P, and the value of $\tau_{\text{v}}A$ at R_{use} is

$$\tau_{\text{v}}A\big(v(d_1), \ldots, v(d_m)\big)^{\text{loc}} = \big[\tau_{\text{v}}A\big(v(d_1), \ldots, v(d_m)\big)_1, \ldots, \tau_{\text{v}}A\big(v(d_1), \ldots, v(d_m)\big)_n\big]$$

where for each $1 \le k \le n$, $\tau_{\text{v}}A\big(\sigma(d_1), \ldots, \sigma(d_m)\big)_k$ at R_{use} has the same value as $\sigma(i_k)$ at R_{def}. This is just because τ_{v} is defined to hold the value in global iteration space at which the associated array element was last assigned to.

Now the value of R'_{def} is

$$A^{\mathcal{C}}\big(v(d_{j_1}), \ldots, v(d_{j_p})\big)\big[v(i_{k_1}), \ldots, v(i_{k_q})\big]$$

Here the quantity in brackets is just the position of R_{def} in its local iteration space $\mathcal{I}$, projected onto the local iteration space component $\mathcal{C} \cap \mathcal{I}$ of $\mathcal{C}$ which is used to rewrite R'_{def}.

Similarly, the value of R'_{use} is

$$A^{\mathcal{C}}\big(v(d_{j_1}), \ldots, v(d_{j_p})\big)\big[\tau_{\text{v}}A\big(v(d_1), \ldots, v(d_m)\big)_{k_1}, \ldots, \tau_{\text{v}}A\big(v(d_1), \ldots, v(d_m)\big)_{k_q}\big]$$

Here the quantity in brackets is just the position $\tau_{\text{v}}A\big(v(d_1), \ldots, v(d_m)\big)^{\text{loc}}$ of R_{def} in its local iteration space $\mathcal{I}$, projected onto the local iteration space component $\mathcal{C} \cap \mathcal{I}$ of the same $\mathcal{C}$ which is used to rewrite R_{use}.

Thus, each subscript in R'_{use} has the same value as its corresponding subscript in R'_{def}.

2. It follows that the value of R'_{use} is one of the values assigned before time t at one of the dynamic references to $A^{\mathcal{C}}$ that assigns to the array element defined at R'_{def}. And one of these values is the value R'_{def} itself.

But by assumption, program P' has the weak dynamic single assignment property with respect to $A^{\mathcal{C}}$ for all times before time t. Therefore, all the values assigned to our fixed element of $A^{\mathcal{C}}$ at that set of dynamic references corresponding to the static reference at R'_{def} are the same. We therefore conclude that the value of R'_{use} is the same as the value of R'_{def}. □

2.2 Theorem *If program P is rewritten to become program P' using a set of dimensions $\mathcal{C}$ for the array A in P, and if for some time t in the global iteration space of P*

- *P' has the weak dynamic single assignment property for the transformed array $A^{\mathcal{C}}$ for all times before t, and*
- *either*
  - *A is an ordinary variable, or*
  - *A is a τ_{w} variable and has the property that if for any time $s \le t$, P and P' are iteratively similar up through time s and if R'_{use} is a use of $A^{\mathcal{C}}$ at time s in P' whose value is not $\perp$, then the corresponding use R_{use} in P is also not $\perp$,*

then Properties B and C hold up through time t.

PROOF. The proof is by induction: we take each dynamic statement, in execution order, and within each statement we consider the references in the order in which they would be evaluated. That is, within each statement we consider uses before defs, and we consider the references within the subscripts of an array reference R before we consider the reference R. We will prove that P' and P are iteratively similar up through time t and that at each time $s \leq t$ each reference R' in the transformed program has the same value as the corresponding reference R in the original program. So the inductive assumption, which is vacuously true at the start of the program, is that

- P and P' are iteratively similar up through time s, and
- each reference R' that has been considered at a time previous to s, as well as each reference R' that has been considered at time s previously to the current reference, has the same value as its corresponding reference R.

We consider the various kinds of references that can occur at time s:

Case I. The reference R is a definition of some object B in P, with a corresponding defining reference R' in P'. B may or may not be the same as A. By the inductive assumption, the values of the right-hand sides are identical, and so the value assigned to R equals the value assigned to R'.

That handles Case I, but there are some comments we need to make for future reference:

B **is different from** A **in** P**.** In this case, the shape of R is the same as that of R'—that is, they have the same number of subscripts. By the inductive assumption, we know therefore that the subscripts of R have the same values as the subscripts of R'. Therefore, the element of B (in P) that is being assigned to is the same element that is assigned to in P'.

B **is the same as** A **in** P**.** Here the elements of R and R' are not guaranteed to be the same, since A was rewritten and so the shape of R may differ from that of R'. But in any case, we note the following, which will be used below: If this defining reference R is the same as R_{def} in the notation used in the lemma, we have just shown (simply by the inductive assumption) that the value of R_{def} is the same as the value of R'_{def}.

Case II. The reference is to a use of some array B. There are two sub-cases to consider:

B **is different from** A **in** P**.** Since only A is rewritten, the subscripts of R (which are subscripts of B) correspond to the subscripts of R'. By the inductive assumption, the value of each subscript of R has the same value as the corresponding subscript of R'. Therefore R refers to the same element as R'. By Case I (first subcase), the values assigned to these two elements in P and P' are equal. Therefore, the values of R and R' are equal.

B **is the same as** A **in** P**.** Thus, R' is a use of $A^{\mathcal{C}}$ in P'. If A is a τ_{w} variable and the value of R is not $\perp$, then R is a use of an element of A defined at a previous definition R_{def}. By the lemma, R' has the same value as the corresponding definition R'_{def} in P', and therefore the value of R' is not $\perp$. To put this another way, if the value of R' is $\perp$, then the value of R must also be $\perp$, and we are done with this case.

In any other case (that is, when R' is not $\perp$), R must also not be $\perp$, for the following reasons:

- A reference to an ordinary variable that is actually reached in a program cannot be a reference to an unassigned value. So if R refers to an ordinary variable, then it automatically has a value different from $\perp$.
- If on the other hand R (and therefore also R') refers to a τ_{w} variable, then the assumption in the theorem guarantees that the value of R is not $\perp$, since the value of R' is now being assumed to be different from $\perp$.

Thus, we have the situation described in the lemma, and so we can regard R as R_{def} and R' as R'_{use}. We then need to show that the value of R'_{use} is the same as the value of R_{use}. Now we know by definition that the value of R_{use} is the same as the value of R_{def}. Case I (second sub-case) showed that the value of R_{def} is the same as that of R'_{def}. (These definitions must have been considered previously in the induction.) And we showed in the lemma that the value of R'_{def} is the same as that of R'_{use}. This proves the result.

This shows that property C is satisfied. If the references R_{use} and R'_{use} are part of Booleans used to determine control flow, this also shows that property B is satisfied.

Thus Properties B and C hold up through time t. □

2.3 Theorem *If program P is rewritten to become program P' using a set of dimensions $\mathcal{C}$ for the array A in P, and if*

- *P' has the weak dynamic single assignment property for the transformed array $A^{\mathcal{C}}$ and*
- *either*
  - *A is an ordinary variable, or*
  - *A is a τ_{w} variable and has the property that if for any time s, P and P' are iteratively similar up through time s and if R'_{use} is a use of $A^{\mathcal{C}}$ at time s in P' whose value is not $\perp$, then the corresponding use R_{use} in P is also not $\perp$,*

then $\mathcal{C}$ is a valid set of contributing dimensions for A.

Proof. This is an immediate corollary of Theorem 2.2. □

2.4 Corollary *For each ordinary variable A, any valid iteration subspace at the defining reference of A is a valid set of contributing dimensions for A.*

Proof. This is true simply because a valid iteration subspace at the defining reference of A parametrizes the values assigned to A at that lexical point in the program, and so rewriting the program using that valid iteration subspace for $\mathcal{C}$ puts the program in weak DSA form with respect to A. □

For example, consider the following source code:

do $i = 1, n$
  $A(f(i)) = i$
  ...
  $\cdots = \ldots A(g(i)) \ldots$
end do

The initial Array SSA form looks something like this, where the definition of $A(f(i))$ becomes a definition of the variant $A_5(f(i))$, and where A_4 is the most recently assigned version of A in $web(A_5)$ prior to the execution of the definition of A_5 in the loop:

do $i = 1, n$
  $A_4(:) = \ldots$
  $\tau_{\mathrm{w}} A_4(:) = \ldots$
  $A_5\big(f(i)\big) = i$
  $\tau_{\mathrm{w}} A_5\big(f(i)\big) = [i]^g$
  forall (k)

$$A_6(k) = \begin{cases} A_5(k) & \text{if } k = f(i) \\ A_4(k) & \text{otherwise} \end{cases}$$

$$\tau_{\mathrm{w}} A_6(k) = \begin{cases} \tau_{\mathrm{w}} A_5(k) & \text{if } k = f(i) \\ \tau_{\mathrm{w}} A_4(k) & \text{otherwise} \end{cases}$$

  end forall
  $\ldots$
  $\cdots = \ldots A_6(g(i)) \ldots$
end do

Certainly a valid iteration subspace at the definition of A_5 is $\{i_1\}$ (corresponding to the loop variable i). Similarly, a valid iteration subspace at the definition of A_6 is $\{i_1, i_2\}$, where again i_1 corresponds to the loop variable i, and i_2 corresponds to the **forall** variable k.

Rewriting the code using these two valid iteration subspaces to rewrite the references to A_5 and A_6 respectively yields the following:

do $i = 1, n$
  $A_4(:) = \ldots$
  $\tau_{\mathrm{w}} A_4(:) = \ldots$
  $A_5^{\mathcal{C}}[i] = i$! Substitution 1
  $\tau_{\mathrm{w}} A_5\big(f(i)\big) = [i]^g$
  forall (k)

$$A_6^{\mathcal{C}}[i,k] = \begin{cases} A_5^{\mathcal{C}}[i] & \text{if } k = f(i) \\ A_4(k) & \text{otherwise} \end{cases} \quad \text{! Substitution 2}$$

$$\tau_{\mathrm{w}} A_6(k) = \begin{cases} \tau_{\mathrm{w}} A_5(k) & \text{if } k = f(i) \\ \tau_{\mathrm{w}} A_4(k) & \text{otherwise} \end{cases}$$

  end forall
  $\ldots$
  $\cdots = \ldots A_6^{\mathcal{C}}\big[i, g(i)\big] \ldots$! Substitution 3
end do

where the indicated substitutions are made as follows:

Substitution 1: $\tau_{\mathrm{v}} A_5\big(f(i)\big) = [i]$, so $A_5\big(f(i)\big)$ is replaced by $A_5^{\mathcal{C}}\big[\tau_{\mathrm{v}} A_5\big(f(i)\big)\big] = A_5^{\mathcal{C}}[i]$.

Substitution 2: On the right-hand side, $\tau_{\mathrm{v}} A_5(k)$ is only needed when $k = f(i)$, and we know that

$\tau_{\mathrm{v}} A_5\big(f(i)\big) = [i]$.

On the left-hand side, $\tau_{\mathrm{v}} A_6(k) = [i, k]$.

Substitution 3: $\tau_{\mathrm{v}} A_6\big(g(i)\big) = \big[i, g(i)\big]$

It is evident that this rewritten code is semantically equivalent to the original code, and (with respect to the ordinary variables) is in weak DSA form. In fact, it is actually in DSA form with respect to the ordinary variables, because the minimal valid iteration subspace at the defining reference to A_5 is the entire local iteration space at that point, and similarly for A_6.

2.3.3 Computation of τ_{v} variables

In the rewriting of the last example, all explicit references to τ_{v} variables were eliminated. This will always be true. We give part of the explanation here.

Let us consider a reference to a variable A. A reference to $\tau_{\mathrm{v}} A$ will only occur in a rewritten reference to A where there is an iteration dimension (say i_k) in the set $\mathcal{C}(A)$ of contributing dimensions of A. In such a case, it is the i_k-coordinate of $\tau_{\mathrm{v}} A$ that appears.

We will show that if certain constraints are satisfied, the reference to $\tau_{\mathrm{v}} A(\vec{k})_{i_k}$ can be replaced by an explicit expression. Letting I denote the loop variable corresponding to the iteration dimension i_k, the constraints are these:

- For each path from the definition of an ultimately used array element to a lexically forward use within the I loop, the element used was assigned at the definition on the same iteration of the loop.
- If the use is lexically backward but still in the I loop, the element used was assigned at the definition on the previous iteration of the loop, if there was such an iteration. If there was no such iteration, the value of τ_{v} must be $\perp$.
- If the use is outside the I loop, the element used was assigned on the last iteration of the I loop the last time that loop was executed before the current use.

These constraints are automatically satisfied for programs in initial Array SSA form (even in the stronger form where we omit the qualification of ultimate use). There are admissible programs, however, for which they are not satisfied. We will see that in case these constraints are not satisfied, the algorithm itself (Section 2.4) will insure that $\tau_{\mathrm{v}} A(\vec{k})_{i_k}$ can be replaced.

So now let us assume that these constraints hold, and let us consider the different kinds of references there can be to A. For each such reference $A(\vec{k})$ we will examine $\tau_{\mathrm{v}} A(\vec{k})_{i_k}$.

If the reference is the definition of A (whether a source definition or a definition in a ϕ assignment), then the value of $\tau_{\mathrm{v}} A$ is just the current point in local iteration space. By our convention, if the definition is in a ϕ assignment, this local iteration space includes components for dimensions of the **forall** implementing the ϕ computation (i.e., the ϕ iteration coordinates).

So now we may assume that the reference is a use of A. We will further assume that the element of A is actually ultimately used, since otherwise we really don't care about the reference at all—in particular, we don't care which element it refers to.

If A was defined by a ϕ assignment, then the value of $\tau_{\mathrm{v}}A$ (which is a point in the global iteration space at the definition of A) includes components for the dimensions of the **forall** that implements the ϕ assignment. The values of those components are just the original (data space) coordinates of the reference A. (This is from Lemma 1.3.) In the last example, for instance, A_6 was defined by a ϕ assignment. The iteration variable of that ϕ assignment is k, and at the reference $A_6(g(i))$, the value of the data space coordinate corresponding to k is $g(i)$. This is where the $g(i)$ in the rewritten reference $A_6[i, g(i)]$ comes from.

Next, we show how to compute the remaining components in $\tau_{\mathrm{v}}A$. We enumerate the different kinds of use references of $A(\vec{k})$. Based on the three properties enumerated above, we immediately see the following:

1. If the reference occurs in the I loop lexically after the definition of A, then $\tau_{\mathrm{v}}A(\vec{k})_{i_k} = I$.
2. If the reference occurs in the I loop lexically before the definition of A, then

$$\tau_{\mathrm{v}}A(\vec{k})_{i_k} = \begin{cases} I - 1 & \text{if } I \text{ is greater than the initial bound of the loop} \\ \text{the value of the terminal bound of the loop the last time the loop was executed} & \text{if } I \text{ equals the initial bound of the loop, and if the loop was previously executed} \\ \perp & \text{in any other case. In fact, in such a case, } \tau_{\mathrm{v}} \text{ itself equals } \perp. \end{cases}$$

3. If the reference occurs outside the I loop, then $\tau_{\mathrm{v}}A(\vec{k})_{i_k}$ is the last value of I taken on in the last iteration of the loop.

Thus, while we use the notation τ_{v}, its value is always known (at least symbolically) at compile-time. It is important to note that this reasoning is possible only because of the restricted nature of our language. Adding arbitrary control flow, for instance, would make this problem much more difficult.

The preceding discussion can be summarized as follows:

2.5 Theorem *If $R = A(\sigma(d_1), \ldots, \sigma(d_m))$ is an ultimately used dynamic reference, and if the conditions at the top of this section are satisfied, then each component of $\tau_{\mathrm{v}}A(\sigma(d_1), \ldots, \sigma(d_m))$ is determined as follows:*

1. *If the component is in a ϕ iteration dimension, its value is one of the subscripts $\sigma(d_1)$. The subscript is determined by the iteration dimension.*
2. *If the component is in an iteration dimension belonging to a loop not reflected in the current local iteration space, then the value of the component is the last value of the loop variable of that loop.*
3. *Otherwise, if the index of the component is i_k, corresponding to the loop variable I_k, then the value of the component is one of the values*
   - I_k

- $I_k - 1$
- *the upper bound of the loop the previous time it was executed*
- $\perp$ *(in which case* $\tau_{\mathrm{v}} A\big(\sigma(d_1), \ldots, \sigma(d_m)\big)$ *is* $\perp$*)*

and a simple expression can be written to produce the correct value.

2.3.4 Some general properties

Here we collect some general properties of valid sets of contributing dimensions and valid iteration subspaces that are used below in constructing an algorithm for their computation.

In these lemmas, e denotes an expression. $\mathcal{S}(e)$ denotes a valid iteration subspace for e. A denotes an array, and $\mathcal{C}(A)$ denotes a valid set of contributing dimensions for A.

2.6 Lemma *1. If e is a literal constant, then $\varnothing$ is a valid iteration subspace for e.*

2. If $e = I(i_j)$ is a loop index in position j in the local iteration space, then $\{i_j\}$ is a valid iteration subspace for e.

PROOF. 1. No loop index can affect the value of e, so $\varnothing$ is a valid iteration subspace for e.

2. Since $I(i_j)$ cannot be assigned to within the body of its loop, its value is independent of any other loop index, so $\{i_j\}$ is a valid iteration subspace for e. □

2.7 Lemma *If e is a function of the set of expressions $\{e_1, e_2, \ldots, e_n\}$ and if $\mathcal{S}(e_i)$ is a valid iteration subspace of e_i for each i, then $\bigcup_{i=1}^{n} \mathcal{S}(e_i)$ is a valid iteration subspace of e.*

PROOF. The values $\{I(i_j) : i_j \in \bigcup_{i=1}^{n} \mathcal{S}(e_i)\}$ determine the values of $\{e_1, \ldots, e_n\}$, which in turn determine the value of e. □

Of course in applications of this lemma one would choose the set $\{e_1, \ldots, e_n\}$ so that e is not a function of any proper subset, in order to make the computed valid iteration subspace of e as small as possible.

2.8 Lemma *If e is any expression and if R runs over the references in e, then*

$$\bigcup_{R} \bigcup_{m} \{\mathcal{S}(\sigma(m)) : m \in \mathcal{C}(R)\}$$

is a valid iteration subspace for e.[3]

PROOF. Since e is a function of the references R, by Lemma 2.7 it is enough to show that for any reference R—say, to an array A—

$$\mathcal{S} \stackrel{\text{def}}{=} \bigcup_{m} \{\mathcal{S}(\sigma(m)) : m \in \mathcal{C}(R)\} \tag{2.1}$$

is a valid iteration subspace for R.

[3]By a harmless abuse of notation, we are writing $\mathcal{C}(R)$ where we really should have written $\mathcal{C}(A)$ where A is the array referred to in the reference R.

Now since $\mathcal{C}(A)$ is a valid set of contributing dimensions, rewriting A in terms of $\mathcal{C}$ puts the program in weak DSA form with respect to A. That is, any two dynamic references to

$$A^{\mathcal{C}}\big(\sigma(d_{j_1}),\ldots,\sigma(d_{j_p})\big)\big[\tau_{\mathrm{v}}A\big(\sigma(d_1),\ldots,\sigma(d_m)\big)_{k_1},\ldots,\tau_{\mathrm{v}}A\big(\sigma(d_1),\ldots,\sigma(d_m)\big)_{k_q}\big]$$

having the same subscripts have the same values. But these subscripts are just the values

$$\{\sigma(m) : m \in \mathcal{C}(R)\}$$

and these values are determined by $\mathcal{S}$. □

2.9 Lemma *If $LHS = RHS$ is an assignment statement and if $\mathcal{S}(RHS)$ is a valid iteration subspace of RHS, then $\mathcal{S}(RHS)$ is also a valid iteration subspace of LHS.*

Remark Note that the assignment statement in this lemma does not necessarily have to be a source assignment. It might be a ϕ assignment, for example.

PROOF. Any loop index that cannot affect the value of RHS cannot affect the value of LHS either. □

2.10 Lemma *If an object A_k is defined at a source definition, and if*

- *R is the defining reference, and*
- *R has data space $\mathcal{D}$ and iteration space $\mathcal{I}$, and*
- *$\mathcal{S} \subseteq \mathcal{I}$ is a valid iteration subspace for R, and*
- *$\mathcal{C}$ is any subset of $\mathcal{D} \cup \mathcal{I}$ such that for each $i \in \mathcal{S}$, $I(i)$ is a function of $\{\sigma(m) : m \in \mathcal{C}\}$*

then $\mathcal{C}$ is a valid set of contributing dimensions for A_k.

PROOF. First, since the set $\{\sigma(m) : m \in \mathcal{C}\}$ parametrizes the valid iteration subspace $\mathcal{S}$ at R, rewriting the references to A_k by means of this set of dimensions gives rise to a transformed program with the weak dynamic single assignment property.

Theorem 2.3 then shows that $\mathcal{C}$ is a valid set of contributing dimensions for A_k. □

2.4 An algorithm for computing contributing dimensions

We are now ready to present an algorithm that computes valid sets of contributing dimensions for each array in the program.

The algorithm is an increasing algorithm. Each expression (and sub-expression) is given an attribute $\mathcal{S}$ (for "valid iteration subspace") and each array name is given an attribute $\mathcal{C}$ (for "valid set of contributing dimensions"). These attributes are initialized in a particular fashion, and then an increasing iterative process is performed. We will prove that when this process terminates, each $\mathcal{S}$ at an array reference is a valid iteration subspace for that array reference and each $\mathcal{C}$ is a valid set of contributing dimensions for its array.

We first need some notation.

Definition *If e is an expression, a subexpression s of e is* primary with respect to e *iff both of the following conditions are satisfied:*

1. *s is one of the following kinds of expressions:*
   - *a literal constant*
   - *a loop variable*
   - *a scalar or array reference*
   - *a pure function*
2. *s is not a proper subexpression of any other subexpression of e that is primary with respect to e*

This definition does not preclude the possibility that e is itself primary with respect to e; i.e., that $s = e$. For instance, if e is a literal constant, then it is primary with respect to itself. We may omit the phrase "with respect to ..." if the context is clear. In fact, we will use the phrase "the primary expressions in e" as a shorthand for "the subexpressions of e that are primary with respect to e".

Every expression in our language is built up out of primary expressions using only the common arithmetic operations. The subscripts of an array reference themselves, as well as the actual arguments to pure functions, are expressions which are in turn made up from other primary expressions in the same way. For instance, in the expression

$$2 * \Big(A\big(3 + B(C(i+j)), f(j+x) - 11\big) + 3\Big) + E(i) + 5$$

which we assume is contained in loops on i and j, and in which A, B, and C are arrays and f is a pure function, the primary expressions are

- 2
- $A\big(3 + B(C(i+j)), f(j+x) - 11\big)$
- 3
- $E(i)$
- 5

The primary expressions in the first subscript of A are

- 3
- $B(C(i+j))$

The primary expressions in the second subscript of A are

- $f(j+x)$
- 11

The subscript of B consists of one primary expression:

- $C(i+j)$

The argument of f consists of two primary expressions:

- j—a loop variable
- x—a scalar reference

and so on.

Primary expressions have the following property: each path from the root of an expression tree for an expression e to its leaves reaches a subexpression s that is primary with respect to e. e is a (pure) function of the set of these primary subexpressions s.

Here now is the algorithm:

Initialization

- For each variable that is live on entry (e.g., a passed parameter to a subroutine), $\mathcal{C}$ is initialized to the data space of that variable.
- At a source definition, say of A, $\mathcal{C}(\tau_{\mathrm{w}}A)$ is initialized to the union of the data space of A with the local iteration space. (That is, it is given its maximal possible value.)
- For all other variables, $\mathcal{C}$ is initialized to $\varnothing$.
- At a **read** statement, $\mathcal{S}$ of the variable being read is initialized to the entire local iteration space.
- At a reference $I(i_j)$ to a loop index, $\mathcal{S}$ is initialized to $\{i_j\}$.
- At all other expressions, $\mathcal{S}$ is initialized to $\varnothing$.

Iterative Computation

The following actions are iterated until no further change takes place. RHS denotes the right-hand side of an arbitrary assignment statement in the program, and LHS denotes the corresponding left-hand side.

1. For each expression e on the right-hand side of an assignment statement, we compute $\mathcal{S}(e)$ by

$$\mathcal{S}(e) \leftarrow \bigcup_{P} \mathcal{S}(P)$$

   where P runs over the primary expressions composing e.

2. For each pure function reference f, we compute $\mathcal{S}(f)$ by

$$\mathcal{S}(f) \leftarrow \bigcup_{a} \mathcal{S}(a)$$

   where a runs over the actual arguments of f.
   Note in particular that a ϕ function is a pure function.

3. For each array reference R that is used (i.e., is not being defined), we compute $\mathcal{S}(R)$ by

$$\mathcal{S}(R) \leftarrow \bigcup_m \{\mathcal{S}(\sigma(m)) : m \in \mathcal{C}(R)\}$$

4. For each statement $LHS = RHS$, we compute $\mathcal{S}(LHS)$ by

$$\mathcal{S}(LHS) \leftarrow \mathcal{S}(RHS)$$

5. For each data object A other than variables live on entry, we compute $\mathcal{C}(A)$ at its defining reference R (which is either the left-hand side of an assignment statement or is the variable in a **read** statement) as follows: consider the family of sets $\{\mathcal{C}_1\}$ with $\mathcal{C}(A) \subseteq \mathcal{C}_1 \subseteq \mathcal{M}$ such that:
   - For each $i \in \mathcal{S}(A)$, $I(i)$ is a function of $\{\sigma(m) : m \in \mathcal{C}_1\}$.
   - In addition, the conclusion of Theorem 2.5 holds for each $i \in \mathcal{C}_1 \cap \mathcal{I}$.

   Note that there will always be at least one $\mathcal{C}_1$ that satisfies the first of these two conditions. The second condition is vacuously satisfied if the program is in initial Array SSA form. Subsequent transformations may have caused it no longer to be true.

   If there is no $\mathcal{C}_1$ satisfying both conditions, then (this is part of the algorithm) reinsert enough of the initial Array SSA machinery so that the conclusion of Theorem 2.5 holds for at least one $\mathcal{C}_1$ satisfying the first condition.

   Note that this may cause an additional iteration dimension to appear, because of the initialization of $\mathcal{S}$ of the right-hand side of an introduced wrap-around ϕ assignment.

   Of those sets $\mathcal{C}_1$ having minimal cardinality, choose one that has the least number of iteration dimensions. (N.B. This is the way we favor data space over iteration space, as mentioned on page 40.) Set $\mathcal{C}(A) \leftarrow \mathcal{C}_1$.

Remarks

- The reason for the maximal initialization of τ_w variables is somewhat subtle: From one point of view, it is simply what we need to make the proof of correctness of the algorithm work. There is, however, some intuition behind it as well:

  For an ordinary array variable, an element whose value is undefined at a certain point in the program has no significance—a legal program can never reference such an element until it has been given a value. For a τ_w variable, however, an element whose value has not been assigned in the program is understood to have the value $\perp$, and this is a meaningful value, because it really is used in Boolean choice expressions in ϕ functions.

  Therefore, any valid set of contributing dimensions has to be large enough to describe the entire array at any point in the program, even when it is only partially defined. This is what causes us to give $\mathcal{C}$ its maximal value for such variables.

  It is important, however, to realize that in typical cases, almost all of these variables can be optimized away. We have given some examples of this already, and we will discuss this in greater detail in later chapters. In addition, there are some other simplifications that can be made, the discussion of which we also defer.

- Action 1 can really be thought of as a special case of action 2.

- Action 2 corresponds to Lemma 2.7.
- Action 3 corresponds to Lemma 2.8.
- Action 4 corresponds to Lemma 2.9.
- Action 5 corresponds to Lemma 2.10.
- The algorithm is not deterministic: an arbitrary choice may be possible in action 5.
- The reason for the second condition in action 5 is to insure that when the program is ultimately rewritten in weak DSA form, the values of any necessary τ_{v} variables can be replaced by expressions not involving τ_{v} variables, thereby eliminating all τ_{v} variables from the program. If this were not possible, then the τ_{v} variables would themselves have to be rewritten in weak DSA form, and this could lead us into an endless recursion.

2.5 Proof of correctness of the algorithm

In this section P will be our admissible program, and P' will be P rewritten using the sets $\mathcal{C}$ of each array as computed by the algorithm of the last section.

2.11 Lemma *The algorithm of the previous section terminates in a finite number of steps.*

PROOF. This is simply because the algorithm is an increasing algorithm—at each step, each value of $\mathcal{S}$ and $\mathcal{C}$ either remains the same or becomes larger, and the set of possible values of each $\mathcal{S}$ and $\mathcal{C}$ is finite. □

2.12 Lemma *Suppose a variable A (which may be an ordinary variable or a τ_{w} variable) is defined at a ϕ assignment. (So each argument E of the ϕ function is either an ordinary variable or a τ_{w} variable.) At the conclusion of the algorithm $\mathcal{C}(A)$ contains*

- *all the data dimensions of each $\mathcal{C}(E)$ as E runs over all the arguments of the ϕ function, together with*
- *each iteration dimension i_j of each $\mathcal{C}(E)$ where E runs over those arguments of the ϕ function such that both E and A are defined within the loop corresponding to i_j.*

PROOF. The ϕ assignment looks like this:

forall $(k_1, k_2, \ldots, k_n)$
$\quad A(k_1, k_2, \ldots, k_n) = \phi\big(\ldots, E(k_1, k_2, \ldots, k_n), \ldots\big)$
end forall

By action 3, $\mathcal{S}(E)$ (by which we really mean $\mathcal{S}$ of the reference to E on the right-hand side) includes all the ϕ iteration coordinates that correspond to data dimensions in $\mathcal{C}(E)$.

Further, if E is an argument of the ϕ function such that both E and A are defined within a loop whose loop variable is $I(i_j)$, and if i_j is in $\mathcal{C}(E)$, then i_j is in $\mathcal{S}(E)$, for the following reason: by action 3, $\mathcal{S}\big(\sigma(i_j)\big)$ is contained in $\mathcal{S}(E)$. It is therefore enough to show that $\mathcal{S}\big(\sigma(i_j)\big) = \{i_j\}$.

Now by action 5 the conclusion of Theorem 2.5 holds. Therefore, $\sigma(i_j)$, which is $\tau_{\mathrm{v}}E(k_1, k_2, \ldots, k_n)_j$, is either $I(i_j)$ or $I(i_j) - 1$. (It can't be constant, because the loop corresponding to dimension i_j contains the ϕ assignment.) Therefore, $\mathcal{S}\big(\sigma(i_j)\big) = \{i_j\}$, so i_j is an element of $\mathcal{S}(E)$.

By actions 1, 2, and 4, $\mathcal{S}(A)$ (by which we really mean $\mathcal{S}$ of the reference to A on the left-hand side) includes the same set of iteration coordinates. By action 5, $\mathcal{C}(A)$ then must include all the data dimensions in $\mathcal{C}(E)$ together with the iteration dimension i_j. □

2.13 Lemma *If a variable $\tau_{\mathrm{w}}B_i$ is defined at a source definition and if there is a path in the (directed) SSA graph from the definition of $\tau_{\mathrm{w}}B_i$ to the definition of a variant $\tau_{\mathrm{w}}B_j$, then at the conclusion of the algorithm*

- *The data dimensions in $\mathcal{C}(\tau_{\mathrm{w}}B_j)$ include all the dimensions of $\tau_{\mathrm{w}}B_j$ (which are also all the declared dimensions of B_i).*
- *The iteration dimensions in $\mathcal{C}(\tau_{\mathrm{w}}B_j)$ include all the iteration dimensions at the definition of B_i corresponding to loops containing the entire path from the definition of B_i to the definition of B_j.*

PROOF. We know by the way the algorithm is initialized that $\mathcal{C}(\tau_{\mathrm{w}}B_i)$ includes all the data and iteration dimensions at the defintion of B_i.

Proceed by induction down the path in the SSA graph. Let us say that L is a loop containing the path from the definition of $\tau_{\mathrm{w}}B_i$ to the definition of $\tau_{\mathrm{w}}B_j$, and say L corresponds to the iteration dimension i_m.

Each edge in the path is one of the following two forms:

1. An edge from a definition of $\tau_{\mathrm{w}}B_p$ to a use of $\tau_{\mathrm{w}}B_p$. Nothing really happens on such an edge as far as $\mathcal{C}(\tau_{\mathrm{w}}B_p)$ is concerned, since $\mathcal{C}(\tau_{\mathrm{w}}B_p)$ is an attribute of the variable $\tau_{\mathrm{w}}B_p$, not of a particular reference to it.
2. An edge from a use of $\tau_{\mathrm{w}}B_p$ as an argument to a ϕ function defining $\tau_{\mathrm{w}}B_q$. Lemma 2.12 shows that all the data dimensions together with the iteration dimension i_m are propagated to $\mathcal{C}(\tau_{\mathrm{w}}B_q)$. □

2.14 Lemma *If*

- *P and P' are iteratively similar up through time t,*
- *each data object in P has the same value as its corresponding object in P' at each time before t,*
- *R is a dynamic use of a variable $\tau_{\mathrm{w}}B$ in P at time t,*
- *R' is the corresponding dynamic use of $\tau_{\mathrm{w}}B^{\mathcal{C}(\tau_{\mathrm{w}}B)}$ in P' at time t,*

then if the value of R' is not $\perp$, then the value of R is also not $\perp$.

PROOF. Suppose first that there is no path in the SSA graph from a source definition to the definition of $\tau_w B$. Then each element of $\tau_w B$ must always be $\perp$, and the same holds for $\tau_w B^{\mathcal{C}(\tau_w B)}$, so the lemma is vacuously true for this case.

Otherwise Lemma 2.13 shows that R' has all the data dimensions of $\tau_w B$. And by assumption the subscripts in those dimensions are the same in R' and in R. Since R' is not $\perp$, the element it refers to was defined. The corresponding element (with the iteration dimensions removed) in P was therefore also defined, at the same time, and R is a reference to that element. (R is a reference to that element because the data subscripts in R and R' are the same, their values having been determined before time t.) Therefore R is not $\perp$. □

2.15 Corollary *1. In order to show that the sets of dimensions $\mathcal{C}$ computed by the algorithm of the previous section are valid sets of contributing dimensions, it suffices to show that rewriting by means of these sets of dimensions puts the program in weak DSA form.*

2. If rewriting using the sets of dimensions $\mathcal{C}$ produced by the algorithm of the previous section puts the program in weak DSA form up to a certain time t in P, then properties B and C on page 43 also hold up to time t.

PROOF. Part 1 follows from Theorem 2.3 together with Lemma 2.14. Part 2 follows from Theorem 2.2 together with Lemma 2.14. □

2.16 Theorem *The algorithm of the last section correctly computes a valid iteration space and a valid set of contributing dimensions for each variable in the program.*

PROOF. First, the algorithm concludes after a finite number of steps by Lemma 2.11.

If when this has happened, the values $\mathcal{C}$ are valid sets of contributing dimensions for their corresponding arrays, then Lemmas 2.6, 2.7, 2.8, and 2.9 show that each $\mathcal{S}$ is a valid iteration subspace for its corresponding reference.

Therefore, we only need to prove that for each variable A (which may actually be a τ_w variable) $\mathcal{C}(A)$ is a valid set of contributing dimensions for A.

If this were not true, then by Corollary 2.15 there would be some array $A^{\mathcal{C}(A)}$ in the transformed program whose definition assigns at least two different values to the same element $A^{\mathcal{C}(A)}(\vec{a})$ in the course of execution of the program. (Note that $\vec{a}$ is a constant vector, not a possibly varying vector expression.) Say the first value is first assigned at time s, and say the second value is first assigned at time t; of course, s and t depend on the element being assigned to.

Of all such definitions of array elements, let $A^{\mathcal{C}(A)}(\vec{a})$ be that element whose t value is earliest.

For convenience, let us denote the defining statement for $A^{\mathcal{C}(A)}$ by

$$A^{\mathcal{C}(A)}(\vec{\alpha}) = \mathrm{expr}_{\mathrm{rhs}}$$

where $\vec{\alpha}$ is a vector of subscript expressions.

Note that none of the dimensions i_k at which s and t differ can be in $\mathcal{S}\big(A(\vec{\alpha})\big)$. For if i_k were, then $I(i_k)$ would be a function of the subscripts $\{\sigma(\kappa) : \kappa \in \mathcal{C}(A)\}$ at the definition of $A^{\mathcal{C}(A)}$ and therefore the elements of $A^{\mathcal{C}(A)}$ defined at s and t would be different, contrary to our assumption.

Now let us look at the right-hand side expr_{rhs} of the definition of A. None of the iteration indices i_k at which s and t differ can be in $\mathcal{S}(\text{expr}_{\text{rhs}})$, since if i_k were, action 4 of the algorithm would as before have propagated it into $\mathcal{S}\big(A(\vec{\alpha})\big)$.

On the other hand, by assumption, expr_{rhs} has different values at s and t.

Therefore there must be a primary expression X in expr_{rhs} which has different values at s and t. Order the primary expressions of expr_{rhs} in the order in which they are evaluated in the execution of the program. (This order may not be unique; any legitimate order will do.) Let X denote the first of these primary expressions.

X cannot be

- a loop variable corresponding to a dimension in which s and t differ, since this would automatically be incorporated by action 1 of the algorithm into $\mathcal{S}(\text{expr}_{\text{rhs}})$
- any other loop variable, since by definition these have the same values at s and t.
- a constant, trivially.

Hence X must be either a pure function or an array reference. Further, any element of $\mathcal{S}(X)$ is also a member of $\mathcal{S}\big(A(\vec{\alpha})\big)$, again by action 4 of the algorithm. So there are two possibilities:

1: X **is a pure function.** The only way it can have two different values at s and t is if some primary expression in some actual argument of X takes on different values at s and t.

2: X **is an array reference.** There are two ways X can have different values at s and t:

a) Some primary expression in some subscript of X takes on different values at s and t.

b) None of these primary expressions do, but some element of X itself takes on two different values at points s and t.
Note that X cannot be a variable that is live on entry, because for such a variable $\mathcal{C}(X) = \mathcal{D}(X)$, so the rewriting according to $\mathcal{C}(X)$ leaves X alone, and no element of X can change its value in the course of execution of the program.
Nor can X be any other ordinary variable, because of the way $A(a)$ was chosen ("...that element whose t value is earliest."). So any such X must be a τ_{w} variable.

Each time we arrive at an X in case 1 or 2a, we recurse (looking at primary expressions in its arguments or subscripts) until one of two things happens:

- We arrive at a situation in which there are no more pure functions or array references (or scalar references) to look at. The only things remaining can be loop variables, and constants. As we have already seen, this situation is impossible.
- We arrive at an array reference in case 2b. As we have seen, this must be a τ_{w} variable; say the left-hand side of its definition is $\tau_{\text{w}} B^{\mathcal{C}(\tau_{\text{w}} B)}(\vec{\beta})$.

So if we have not yet arrived at a contradiction, we have found an array element $\tau_{\text{w}} B^{\mathcal{C}(\tau_{\text{w}} B)}(\vec{b})$ (where $\vec{b}$ is a constant, not an expression) which at times s and t takes on two distinct values, and which is represented on the right-hand side of the definition of $A^{\mathcal{C}}$.

This is not in itself a contradiction—one of the values of $\tau_{\mathrm{w}}B^{\mathcal{C}(\tau_{\mathrm{w}}B)}(\vec{b})$ might be $\perp$. But it does show that the assignment to $A^{\mathcal{C}(A)}$ must be a ϕ assignment, since it has a τ_{w} variable on its right-hand side.

Let the definitions of $\tau_{\mathrm{w}}B^{\mathcal{C}(\tau_{\mathrm{w}}B)}(\vec{b})$ that reach the uses at times s and t occur at times s' and t' respectively. We must have $s' < s < t' < t$. Because of the way that $A^{\mathcal{C}(A)}(\vec{a})$ was chosen, s' must be the time of program entry, and the value of $\tau_{\mathrm{w}}B^{\mathcal{C}(\tau_{\mathrm{w}}B)}(\vec{b})$ at that time is of course $\perp$.

Now $P(t')$ must be a (static) point in the program inside the outermost loop L at which s and t differ, since $s < t' < t$.

There are two possibilities:

1: For some $\tau_{\mathrm{w}}B$ which is an argument to the ϕ function defining A (this may be *any* such argument, not necessarily the $\tau_{\mathrm{w}}B$ arrived at above), there is a source assignment to a member of $web(\tau_{\mathrm{w}}B)$ inside L that reaches the definition of $\tau_{\mathrm{w}}B$ by a path in the SSA graph that lies entirely within L. Say that source assignment is to $\tau_{\mathrm{w}}B_i$. Then $\mathcal{C}(\tau_{\mathrm{w}}B_i)$ includes the dimension corresponding to the loop L. By Lemma 2.13, this dimension is then propagated by the algorithm to $\mathcal{C}(\tau_{\mathrm{w}}B)$. By Lemma 2.12, this dimension is then propagated to $\mathcal{C}(A)$. This in turn shows that the elements of $A^{\mathcal{C}(A)}(\vec{k})$ referenced at times s and t are different, which contradicts our assumption.

2: There is no such source assignment. Because of the requirement that there be a τ_{w} variable for each ordinary variable in the ϕ assignment, it follows also that for each ordinary variable B which is an argument to the ϕ function defining A there is no source assignment to a member of $web(B)$ inside L that reaches the definition of B by a path in the SSA graph that lies entirely within L.

It then follows that there is no source assignment to a member of $web(A)$ inside L that reaches the definition of A by a path in the SSA graph that lies entirely within L. By the way we have defined admissible programs (see item 5 on page 30), this means that the values assigned to A at times s and t in program P must be identical.

On the other hand, by our construction, program P' satisfies the conditions of Theorem 2.2 for time t. Therefore the values assigned in P' up through time t are the same as those assigned in P up through time t. But the values assigned to $A^{\mathcal{C}}$ in P' at times s and t are by assumption different, and this is a contradiction.

These contradictions conclude the proof. □

Bibliography

Ballance, Robert A., Arthur B. Maccabe, and Karl J. Ottenstein. 1990. *The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages*, Proceedings of the Acm Sigplan '90 Conference on Programming Language Design and Implementation, ACM Press, pp. 257–271.

Cytron, Ron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. *Efficiently computing static single assignment form and the control dependence graph*, Transactions on Programming Languages and Systems **13**, 451–490.

Knobe, Kathleen and Vivek Sarkar. 1998. *Array SSA form and its use in parallelization*, Proceedings of the 25th Acm Sigplan-Sigact Symposium on Principles of Programming Languages (Popl '98), Association for Computing Machinery.

Knobe, Kathleen B. 1997. *The subspace model: Shape-based compilation for parallel systems*, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Muchnick, Steven S. 1997. *Advanced compiler design and implementation*, Morgan Kaufmann Publishers, San Francisco, [QA76.76.C65M8].

Index