



## Weak Dynamic Single Assignment Form

Carl Offner, Kathleen Knobe  
Cambridge Research Laboratory  
HP Laboratories Cambridge  
HPL-2003-169(R.1)  
August 1, 2005\*

dynamic single  
assignment, static  
single assignment,  
compiler  
optimizations,  
parallelism,  
intermediate  
representations

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.

# Weak Dynamic Single Assignment Form

Carl Offner and Kathleen Knobe

July 15, 2005



## **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

<b>Introduction</b>	<b>1</b>
<b>1 Array static single assignment form</b>	<b>2</b>
1.1 Notation and definitions	2
1.1.1 Programs	2
1.1.2 Data space	4
1.1.3 Iteration space	4
1.1.4 SSA form	9
1.2 Array SSA form	11
1.2.1 The naive Array SSA construction	11
1.2.2 SSA graphs and webs	14
1.2.3 $\tau_w$ variables: time of entry to the web	15
1.2.4 $\phi$ iteration indices	18
1.2.5 $\phi$ iteration components of $\tau_v$ variables	18
1.2.6 Extending the subscript map to iteration dimensions	19
1.2.7 Initial Array SSA form	19
1.2.8 Equivalence of the initial and naive Array SSA constructions	20
1.3 Admissible program transformations	23
1.3.1 Array SSA form and true dependences	23
1.3.2 Simplification of <b>if</b> merges	27
1.3.3 Admissible transformations	28
1.4 General properties of $\tau_w$ and $\tau_v$ variables	37
1.4.1 Properties that are true for programs in initial array SSA form	37
1.4.2 Properties that are true for admissible programs	38
<b>2 Weak dynamic single assignment form</b>	<b>41</b>
2.1 Overview	41
2.2 The space of values at a static reference	45
2.3 Contributing dimensions	46
2.3.1 Rewriting corresponding to an arbitrary set of dimensions	46
2.3.2 Valid sets of contributing dimensions	48
2.3.3 Computation of $\tau_v$ variables	54
2.3.4 Some general properties	56
2.4 An algorithm for computing contributing dimensions	58
2.5 Proof of correctness of the algorithm	62
<b>Bibliography</b>	<b>68</b>
<b>Index</b>	<b>69</b>



# 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,  $\tau_v$  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.<sup>1</sup> 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

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

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.

---

<sup>1</sup>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.

**do** loop indices are regarded as local to the loop: they have no meaningful value outside the loop body.

**do** 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  $j^{\text{th}}$  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  $\sigma$  mapping an array reference  $R$  and data space index  $d_i$  to the  $i^{\text{th}}$  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  $(\sigma(d_1), \sigma(d_2), \dots, \sigma(d_k))$ .

### 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.

---

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

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*<sup>2</sup>.

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 Figure 1.1. 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  $\lambda$  indicates a *lexical* position. So coordinates of a point in global iteration space that are not subscripted by  $\lambda$  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.

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.

---

<sup>2</sup>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.

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.

$\tau_w$  variables and  $\tau_v$  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  $t^{\text{loc}}$  to denote the corresponding point in the local iteration space at that lexical position. That is,  $t^{\text{loc}}$  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  $t_2$ .  $\square$

#### 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.

---

<pre> a = 0 do i = 1, n   print a   a = a + 1 end do </pre>	<pre> a<sub>0</sub> = 0 do i = 1, n   ▷a<sub>1</sub> = φ(a<sub>0</sub>, a<sub>2</sub>)   print a<sub>1</sub>   a<sub>2</sub> = a<sub>1</sub> + 1 end do </pre>
---	--

Figure 1.2: A program and its translation into SSA form. The new statement that is added when the program is put into SSA form is prefixed by a triangle and typeset in a slightly smaller font.

---

Thus, programs are converted into SSA form by the introduction of  $\phi$  assignments. A  $\phi$  assignment (or  $\phi$  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  $\phi$  function. The purpose of a  $\phi$  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  $\phi$  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  $\phi$  assignments are subsequently eliminated. When we come to Array SSA form below, we will see how  $\phi$  functions can be turned into honest functions.

Note that this  $\phi$  assignment is prefixed with a triangle and typeset in a slightly smaller font. We do this consistently in our code in this paper simply as a way of making the code somewhat easier to read.

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  $\phi$  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.  $\phi$  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— $X \xrightarrow{\pm} Z$  and  $Y \xrightarrow{\pm} Z$ —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  $\phi$  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  $\phi$  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  $\phi$  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  $\phi$  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  $\phi$  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  $\phi$  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.

This constraint is used, in conjunction with Lemma 1.1 in the proof below of Case 2 of Lemma 1.4. 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  $\phi$  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*  $\phi$ , 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  $\phi$  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.  $\square$

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

```

V1 = 1
V2 = 2
do i = 1, 100
  ▸V3 =  $\phi(V_1, V_4)$ 
  ▸V4 =  $\phi(V_2, V_3)$ 
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.

---

<pre> a(:) = 0 do i = 1, n   a(i) = a(f(i)) + 1   print a(g(i)) end do </pre>	<pre> a0(:) = 0 do i = 1, n   &gt;a1 = φ(a0, a3)   a2(i) = a1(f(i)) + 1   &gt;a3 = φ(a1, a2)   print a3(g(i)) end do </pre>
---	---

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

---

In that example, the  $\phi$  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  $\phi$  assignment*.

In addition, however, another  $\phi$  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  $\phi$  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  $\phi$  assignment is one of the values of the corresponding elements of the arrays that are the arguments to the  $\phi$  function. We need to find a way of making this selection explicit; that is, of turning the  $\phi$  into an honest function.

Now for a given  $\phi$  function  $\phi(a_1, a_2, \dots, a_n)$ , say and a given element  $\vec{k}$ , the value returned by the  $\phi$  function should be that value in the set  $\{a_1(\vec{k}), a_2(\vec{k}), \dots, a_n(\vec{k})\}$  which was assigned last. We can accomplish this by introducing, for each SSA variant  $a_i$  an auxiliary array  $\tau_v a_i$  such that at any point in program execution  $\tau_v a_i(\vec{k})$  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  $\tau_v$  as standing for “time of assignment to the variant”, or simply as “variant time”.

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

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

Note that although we have introduced new assignment statements into the program to assign to the  $\tau_v$  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  $\tau_w$  variables to be defined later) will for the most part not appear in the final code. Further, each  $\tau_v$  variable (and later,  $\tau_w$  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  $\tau_v$  variables thus turn the  $\phi$  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

---

$a_0(\cdot) = 0$ $\triangleright \tau_v a_0(\cdot) = [1_\lambda]$ <b>do</b> $i = 1, n$ $\triangleright a_1 = \phi(a_0, \tau_v a_0; a_3, \tau_v a_3)$ $\triangleright \tau_v a_1 = [2_\lambda, i, 1_\lambda]$  $a_2(i) = a_1(f(i)) + 1$ $\triangleright \tau_v a_2(i) = [2_\lambda, i, 2_\lambda]$  $\triangleright a_3 = \phi(a_1, \tau_v a_1; a_2, \tau_v a_2)$ $\triangleright \tau_v a_3 = [2_\lambda, i, 3_\lambda]$  <b>print</b> $a_3(g(i))$ <b>end do</b>	$a_0(\cdot) = 0$ $\triangleright \tau_v a_0(\cdot) = [1_\lambda]$ <b>do</b> $i = 1, n$ $\triangleright$ <b>forall</b> $(k = 1 : n)$ $\triangleright a_1(k) = \begin{cases} a_0(k) & \text{if } \tau_v a_0(k) > \tau_v a_3(k) \\ a_3(k) & \text{otherwise} \end{cases}$ $\triangleright$ <b>end forall</b> $\triangleright \tau_v a_1 = [2_\lambda, i, 1_\lambda]$  $a_2(i) = a_1(f(i)) + 1$ $\triangleright \tau_v a_2(i) = [2_\lambda, i, 2_\lambda]$  $\triangleright$ <b>forall</b> $(k = 1 : n)$ $\triangleright a_3(k) = \begin{cases} a_1(k) & \text{if } \tau_v a_1(k) > \tau_v a_2(k) \\ a_2(k) & \text{otherwise} \end{cases}$ $\triangleright$ <b>end forall</b> $\triangleright \tau_v a_3 = [2_\lambda, i, 3_\lambda]$  <b>print</b> $a_3(g(i))$ <b>end do</b>
---	---

Figure 1.4: Semantics of the  $\phi$  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.  $\phi$  assignments are introduced, as in scalar SSA form, and in addition, definition  $\phi$  assignments are introduced after each source definition. Each  $\phi$  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  $\phi$  functions, and are used to specify the selection made by the  $\phi$  function, as follows:

The  $\phi$  function

$$V_{k_0} = \phi(V_{k_1}, \tau_v V_{k_1}; \dots; V_{k_n}, \tau_v V_{k_n})$$

is interpreted as

$$\begin{array}{l} \mathbf{forall} (\vec{k}) \\ V_{k_0}(\vec{k}) = \begin{cases} V_{k_1}(\vec{k}) & \text{if } \tau_v V_{k_1}(\vec{k}) = \max(\tau_v V_{k_1}(\vec{k}), \dots, \tau_v V_{k_n}(\vec{k})) \\ \dots \\ V_{k_n}(\vec{k}) & \text{if } \tau_v V_{k_n}(\vec{k}) = \max(\tau_v V_{k_1}(\vec{k}), \dots, \tau_v V_{k_n}(\vec{k})) \end{cases} \\ \mathbf{end forall} \end{array}$$

6. The value of any array element  $V_i(\vec{k})$  encountered during execution of a source statement, or defined by a  $\phi$  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  $\tau_v$  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  $\phi$  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  $\tau_v$  variables (and later, from  $\tau_w$  variables) in descriptive text. Note in particular that loop indices are not classified as ordinary variables, since they have no SSA variants.

$\tau_v$  variables (and  $\tau_w$  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  $\tau_v$  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  $\phi$  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  $\phi$  function to the variant  $A_j$  defined by that function.

Note that these graphs only take account of the ordinary variables.  $\tau_v$  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  $\phi$  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 $\tau_w$ variables: time of entry to the web

At any time in the course of execution of the program the variable  $\tau_v A_i(\vec{k})$  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  $\tau_v A_i(\vec{k})$  and often earlier. The variable that holds this new time will be denoted by  $\tau_w A_i$ , and it is defined as follows:

**Definition** *For each ordinary variable  $A_i$ , there is an associated variable  $\tau_w A_i$ . At any time  $t$  in the course of execution of the program,  $\tau_w A_i(\vec{k})$  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  $\phi$  assignments.*

To help remember this notation, think of  $\tau_w$  as standing for “time of entry to the web”, or simply as “web time”.  $\tau_w$  variables were called @ variables in Knobe and Sarkar (1998).

Thus, if  $A_i$  is defined at a source definition, then  $\tau_w A_i(\vec{k}) = \tau_v A_i(\vec{k})$  for all  $\vec{k}$ . But if  $A_i$  is defined at a  $\phi$  assignment, then  $\tau_w A_i(\vec{k}) < \tau_v A_i(\vec{k})$  for all  $\vec{k}$ .

We see just by the definitions that

- $\tau_w A_i(\vec{k})$  is the *earliest* time at which the value currently in  $A_i(\vec{k})$  was assigned to some element of the web containing  $A_i$ , subsequently reaching  $A_i(\vec{k})$  by a chain of  $\phi$  assignments.

- $\tau_v A_i(\vec{k})$  is the *latest* such time. Of course in this case there are no  $\phi$  assignments at all, and the value is assigned to  $A_i(\vec{k})$  directly.

Further, and again just by its definition,  $\tau_w A_i(\vec{k})$  is the time in the original source program at which the value currently in  $A_i(\vec{k})$  was computed and assigned to  $A(\vec{k})$ .

Computing  $\tau_w A_i$  is straightforward:

- At a source definition,  $\tau_w A_i(\vec{k})$  is defined to be equal to  $\tau_v A_i(\vec{k})$ .
- At a  $\phi$  assignment

**forall** ( $\vec{k}$ )

$$A_{k_0}(\vec{k}) = \begin{cases} A_{k_1}(\vec{k}) & \text{if } \tau_v A_{k_1}(\vec{k}) = \max(\tau_v A_{k_1}(\vec{k}), \dots, \tau_v A_{k_n}(\vec{k})) \\ \dots & \\ A_{k_n}(\vec{k}) & \text{if } \tau_v A_{k_n}(\vec{k}) = \max(\tau_v A_{k_1}(\vec{k}), \dots, \tau_v A_{k_n}(\vec{k})) \end{cases}$$

**end forall**

we extend the  $\phi$  assignment by adding a definition of  $\tau_w A_{k_0}$  to the **forall**, as follows:

**forall** ( $\vec{k}$ )

$$A_{k_0}(\vec{k}) = \begin{cases} A_{k_1}(\vec{k}) & \text{if } \tau_v A_{k_1}(\vec{k}) = \max(\tau_v A_{k_1}(\vec{k}), \dots, \tau_v A_{k_n}(\vec{k})) \\ \dots & \\ A_{k_n}(\vec{k}) & \text{if } \tau_v A_{k_n}(\vec{k}) = \max(\tau_v A_{k_1}(\vec{k}), \dots, \tau_v A_{k_n}(\vec{k})) \end{cases}$$

$$\tau_w A_{k_0}(\vec{k}) = \begin{cases} \tau_w A_{k_1}(\vec{k}) & \text{if } \tau_v A_{k_1}(\vec{k}) = \max(\tau_v A_{k_1}(\vec{k}), \dots, \tau_v A_{k_n}(\vec{k})) \\ \dots & \\ \tau_w A_{k_n}(\vec{k}) & \text{if } \tau_v A_{k_n}(\vec{k}) = \max(\tau_v A_{k_1}(\vec{k}), \dots, \tau_v A_{k_n}(\vec{k})) \end{cases}$$

**end forall**

We may abbreviate this simply by writing

$$A_{k_0} = \phi( A_{k_1}, \tau_v A_{k_1}; \dots; A_{k_n}, \tau_v A_{k_n} )$$

$$\tau_w A_{k_0} = \phi(\tau_w A_{k_1}, \tau_v A_{k_1}; \dots; \tau_w A_{k_n}, \tau_v A_{k_n} )$$

Figure 1.5 contains a simple scalar example that illustrates the difference between  $\tau_v$  and  $\tau_w$  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  $\tau_v$  and  $\tau_w$  variables

---

```

 $x_0 = 0$ 
 $\triangleright \tau_v x_0 = [1_\lambda]$ 
 $\triangleright \tau_w x_0 = [1_\lambda]$ 
do  $i = 1, 2$ 
   $\triangleright x_1 = \phi(x_0, \tau_v x_0; x_2, \tau_v x_2)$ 
   $\triangleright \tau_v x_1 = [2_\lambda, i, 1_\lambda]$ 
   $\triangleright \tau_w x_1 = \phi(\tau_w x_0, \tau_v x_0; \tau_w x_2, \tau_v x_2)$ 
  print  $x_1$ 
   $x_2 = 5$ 
   $\triangleright \tau_v x_2 = [2_\lambda, i, 3_\lambda]$ 
   $\triangleright \tau_w x_2 = [2_\lambda, i, 3_\lambda]$ 
end do

```

Figure 1.6: The code from Figure 1.5 in SSA form, with the  $\tau_v$  and  $\tau_w$  variables added.

---

on each iteration of the loop occur with the following values:

	value before loop	value ( $i = 1$ )	value ( $i = 2$ )
$(x_0, \tau_w x_0, \tau_v x_0)$	$(0, [1_\lambda], [1_\lambda])$		
$(x_1, \tau_w x_1, \tau_v x_1)$	$(\perp, \perp, \perp)$	$(0, [1_\lambda], [1]^\theta)$	$(5, [1]^\theta, [2]^\theta)$
$(x_2, \tau_w x_2, \tau_v x_2)$	$(\perp, \perp, \perp)$	$(5, [1]^\theta, [1]^\theta)$	$(5, [2]^\theta, [2]^\theta)$

Thus on iteration 2 of the loop,  $\tau_w x_1$  is assigned the value  $[1]^\theta$ , 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  $\tau_v x_1$ ).

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  $\tau_v$  variables will not appear in generated code.
- In typical cases most or all of the  $\tau_w$  variables can be optimized away.

These points will be justified below.

As a matter of notation, by  $(\tau_v A_i(\vec{k}))_j$  we mean the  $j^{\text{th}}$  local iteration component of  $\tau_v A_i(\vec{k})$ . Of course  $\tau_v A_i(\vec{k})$  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  $\tau_w$  variables.

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



### 1.2.4 $\phi$ iteration indices

A  $\phi$  assignment, which in naive Array SSA form has the form

$$\begin{aligned} A_6(:, :, :) &= \phi(A_5(:, :, :), \tau_v A_5(:, :, :); A_4(:, :, :), \tau_v A_4(:, :, :)) \\ \tau_w A_6(:, :, :) &= \phi(\tau_w A_5(:, :, :), \tau_v A_5(:, :, :); \tau_w A_4(:, :, :), \tau_v A_4(:, :, :)) \end{aligned}$$

can be written more explicitly like this:

$$\begin{aligned} &\mathbf{forall} (p, q, r) \\ &A_6(p, q, r) = \begin{cases} A_5(p, q, r) & \text{if } \tau_v A_5(p, q, r) > \tau_v A_4(p, q, r) \\ A_4(p, q, r) & \text{otherwise} \end{cases} \\ &\tau_w A_6(p, q, r) = \begin{cases} \tau_w A_5(p, q, r) & \text{if } \tau_v A_5(p, q, r) > \tau_v A_4(p, q, r) \\ \tau_w A_4(p, q, r) & \text{otherwise} \end{cases} \\ &\mathbf{end forall} \end{aligned}$$

Here we have introduced explicit iteration variables  $p$ ,  $q$ , and  $r$  to express the computations inside the  $\phi$  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  *$\phi$  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  $\phi$  assignment.

### 1.2.5 $\phi$ iteration components of $\tau_v$ variables

There is a simple result that won't be needed until later but that is convenient to mention at this point: consider a  $\phi$  assignment, say the one illustrated just above in Section 1.2.4.

$$\begin{aligned} &\mathbf{forall} (p, q, r) \\ &A_6(p, q, r) = \begin{cases} A_5(p, q, r) & \text{if } \tau_v A_5(p, q, r) > \tau_v A_4(p, q, r) \\ A_4(p, q, r) & \text{otherwise} \end{cases} \\ &\tau_w A_6(p, q, r) = \begin{cases} \tau_w A_5(p, q, r) & \text{if } \tau_v A_5(p, q, r) > \tau_v A_4(p, q, r) \\ \tau_w A_4(p, q, r) & \text{otherwise} \end{cases} \\ &\mathbf{end forall} \end{aligned}$$

Since this **forall** implements a  $\phi$  assignment, there is no control  $\phi$  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  $\phi$  iteration variables  $p$ ,  $q$ , and  $r$  are indexed by iteration indices  $i_7$ ,  $i_8$ , and  $i_9$ , respectively. Then the value of  $(\tau_v A_6(S_1, S_2, S_3))_{i_7}$  (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*

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

then  $(\tau_v A(S_1, \dots, S_n))_k = S_q$ .

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

### 1.2.6 Extending the subscript map to iteration dimensions

For convenience, we extend the subscript map  $\sigma$ , which we defined earlier on page 4. As previously defined,  $\sigma$  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,  $\sigma(i_j)$  will denote the  $j^{\text{th}}$  component of the  $\tau_v$  variable associated with that reference.

For instance, suppose  $R$  is a reference to an element  $A(\sigma(d_1), \sigma(d_2), \sigma(d_3))$  of a 3-dimensional array  $A$  inside 4 nested loops. With respect to this reference, we have  $\sigma(i_2) = (\tau_v A(\sigma(d_1), \sigma(d_2), \sigma(d_3)))_2$ . 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 the expression denoted by  $\sigma(i_2)$  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 46), 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 57).

### 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  $\tau_v$  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  $\tau_v$  variable in each Boolean choice expression in each  $\phi$  function by its corresponding  $\tau_w$  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  $\phi$  function may no longer be disjoint. It may be, for instance, that  $\tau_w A_i(\vec{k}) = \tau_w A_j(\vec{k})$ . (This could never happen with  $\tau_v$  variables.) This is really not a problem, because

if  $\tau_w A_i(\vec{k}) = \tau_w A_j(\vec{k})$  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  $\phi$  function. So while there may be some formal ambiguity in the  $\phi$  function, it is harmless.

2. The reasoning above on page 12 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<sup>3</sup>.

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  $\tau_w A(\vec{k}) < \tau_w B(\vec{k})$  for some  $\vec{k}$ , 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 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.  $\tau_w B(\vec{k}) = \tau_v A(\vec{k})$
2.  $\tau_w B(\vec{k}) < \tau_v A(\vec{k})$

We consider these two cases separately.

---

<sup>3</sup>We only have to replace the statement

Now for a given  $\phi$  function  $\phi(a_1, a_2, \dots, a_n)$ , say and a given element  $\vec{k}$ , the value returned by the  $\phi$  function should be that value in the set  $\{a_1(\vec{k}), a_2(\vec{k}), \dots, a_n(\vec{k})\}$  which was assigned last.

by the statement

Now for a given  $\phi$  function  $\phi(a_1, a_2, \dots, a_n)$ , say and a given element  $\vec{k}$ , the value returned by the  $\phi$  function should be that value in the set  $\{a_1(\vec{k}), a_2(\vec{k}), \dots, a_n(\vec{k})\}$  which entered the web last.

**Case 1.** In this case we have

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

Now  $P(\tau_v A(\vec{k}))$  must be a  $\phi$  assignment, since  $\tau_w A(\vec{k}) < \tau_v A(\vec{k})$ . But this is impossible, since  $P(\tau_v A(\vec{k})) = P(\tau_w B(\vec{k}))$ , which must be a source statement. So Case 1 cannot happen.

**Case 2.** In this case we have

$$(1.1) \quad \tau_w A(\vec{k}) < \tau_w B(\vec{k}) < \tau_v A(\vec{k})$$

We must show that such a relation cannot hold. In doing this, we may make the following assumption:

$$(1.2) \quad \tau_w B(\vec{k}) \text{ is the last time before time } \tau_v A(\vec{k}) \text{ at which the } \vec{k} \text{ element of any member of } \text{web}(A) \text{ was assigned to at a source statement.}$$

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  $\phi$  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 assumption (1.2) 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  $P_v$ . (That is, (1.1) must be false in  $P$ .) 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  $\phi$  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.  $\square$

**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  $\phi$  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

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

be any  $\phi$  assignment in  $P_v$ . For a given vector  $\vec{k}$  of subscripts, let  $A_{i_j}(\vec{k})$  be that element chosen by the  $\phi$  function. That is,  $\tau_v A_{i_j}(\vec{k})$  is the largest of the  $\tau_v$  values occurring as arguments to the  $\phi$  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  $\phi$  function. Thus, substituting for each  $\tau_v A_{i_j}$  in the  $\phi$  function the value  $\tau_w A_{i_j}$  does not change the value returned by the  $\phi$ . (Recall that no harm is done if two values  $\tau_w A_{i_j}(\vec{k})$  and  $\tau_w A_{i_k}(\vec{k})$  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.  $\square$

Note that once the program is in initial Array SSA form, the propagation of  $\tau_w$  variables becomes formally simpler: in naive Array SSA form  $\tau_w$  variables would be combined at a  $\phi$  assignment like this:

$$\begin{array}{l} \mathbf{forall}(\vec{k}) \\ \tau_w A_{k_0}(\vec{k}) = \begin{cases} \tau_w A_{k_1}(\vec{k}) & \text{if } \tau_v A_{k_1}(\vec{k}) = \max(\tau_v A_{k_1}(\vec{k}), \dots, \tau_v A_{k_n}(\vec{k})) \\ \dots \\ \tau_w A_{k_n}(\vec{k}) & \text{if } \tau_v A_{k_n}(\vec{k}) = \max(\tau_v A_{k_1}(\vec{k}), \dots, \tau_v A_{k_n}(\vec{k})) \end{cases} \\ \mathbf{end forall} \end{array}$$

After replacing the  $\tau_v$  variables in these Boolean choice expressions by the corresponding  $\tau_w$  variables, thus putting the program in initial Array SSA form, this combination can be written more simply as

$$\tau_w A_{k_0}(\vec{k}) = \max(\tau_w A_{k_1}, \dots, \tau_w A_{k_n})$$

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

PROOF. This follows immediately from Lemma 1.4.  $\square$

**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  $\phi$  assignment inside  $L$ .*
- *There is no source assignment in  $L$  of any variable in  $\text{web}(V_i)$  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  $\tau_w$  variable—say  $V_i$  is  $\tau_w B_i$ —then the  $\phi$  assignment defining  $\tau_w B_i$  occurs in conjunction with a  $\phi$  assignment defining  $B_i$ . Further, since there is no source assignment in  $L$  of any  $\tau_w$  variable whose value can reach the assignment to  $\tau_w B_i$  by a path in  $L$ , there must similarly be no source assignment in  $L$  of any variable in  $\text{web}(B_i)$  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  $\tau_w B_i(\vec{k})$  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.  $\square$

## 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. Here we give some examples and show how they can be simplified.

#### Example: spurious cycles of dependences

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  $\phi$  statements in the “obvious” places, we get the following code:

```

A0(:) = 0
▷τwA0(:) = [1λ]
do i = 1, n
  ▷A1(:) = φ(A0(:), τwA0(:); A3(:), τwA3(:))
  ▷τwA1(:) = max(τwA0(:), τwA3(:))

  A2(i) = A1(i) + 1
  ▷τwA2(i) = [i]g
  ▷A3(:) = φ(A2(:), τwA2(:); A1(:), τwA1(:))
  ▷τwA3(:) = max(τwA2(:), τwA1(:))
end do

```

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

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  $\phi$  assignment (defining  $A_1$ ) at the top of the loop. This  $\phi$  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  $\phi$  assignment at the top of the loop did not need to be inserted. We could simply have written the code like this:

```

A0(:) = 0
▷τwA0(:) = [1λ]
do i = 1, n
  A1(i) = A0(i) + 1
  ▷τwA1(i) = [i]g
  ▷A2(:) = φ(A1(:), τwA1(:); A0(:), τwA0(:))
  ▷τwA2(:) = max(τwA1(:), τwA0(:))
end do

```

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

### Example: dead code

Here is another example:

```

A(:) = 0
do i = 1, n
  A(i) = ...
  if (i == 1) then
    ... = ... A(i) ...
  else
    ... = ... A(i - 1) ...
  end if
end do

```

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

```

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

   $A_2(i) = \dots$ 
   $\triangleright \tau_w A_2(i) = [i]^g$ 
   $\triangleright A_3(\cdot) = \phi(A_2(\cdot), \tau_w A_2(\cdot); A_1(\cdot), \tau_w A_1(\cdot))$ 
   $\triangleright \tau_w A_3(\cdot) = \max(\tau_w A_2(\cdot), \tau_w A_1(\cdot))$ 

  if  $(i == 1)$  then
     $\dots = \dots A_3(i) \dots$ 
  else
     $\dots = \dots A_3(i - 1) \dots$ 
  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 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:

```

 $A_0(\cdot) = 0$ 
 $\triangleright \tau_w A_0(\cdot) = \langle \text{before the loop} \rangle$ 
do  $i = 1, n$ 
   $A_2(i) = \dots$ 
   $\triangleright \tau_w A_2(i) = [i]^g$ 

  if  $(i == 1)$  then
     $\dots = \dots A_2(i) \dots$ 
  else
     $\dots = \dots A_2(i - 1) \dots$ 
  end if
end do

```

Of course there will be a  $\phi$  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  $\phi$  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.



**Example: reinstating deleted code**

There is a certain circumstance (explained below in Section 2.4) in which a wrap-around  $\phi$  assignment that has been removed in this way needs to be reinstated, and reinstating such a wrap-around  $\phi$  assignment in turn may force other  $\phi$  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(f(i)) = ...
  ... = ... A(f(i)) ...
  A(g(i)) = ...
  ... = ... A(g(i)) ...
end do

```

After putting in the initial Array SSA code, we have

```

A0(:) = 0
▷τwA0(:) = [1,λ]
do i = 1, n
  ▷A1(:) = φ(A0(:), τwA0(:); A5(:), τwA5(:))
  ▷τwA1(:) = max(τwA0(:), τwA5(:))

  A2(f(i)) = ...
  ▷τwA2(f(i)) = [i]g
  ▷A3(:) = φ(A2(:), τwA2(:); A1(:), τwA1(:))
  ▷τwA3(:) = max(τwA2(:), τwA1(:))

  ... = ... A3(f(i)) ...

  A4(g(i)) = ...
  ▷τwA4(g(i)) = [i]g
  ▷A5(:) = φ(A4(:), τwA4(:); A3(:), τwA3(:))
  ▷τwA5(:) = max(τwA4(:), τwA3(:))

  ... = ... A5(g(i)) ...
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:

```

 $A_0(\cdot) = 0$ 
 $\triangleright \tau_w A_0(\cdot) = [1_\lambda]$ 
do  $i = 1, n$ 
   $A_2(f(i)) = \dots$ 
   $\triangleright \tau_w A_2(f(i)) = [i]^g$ 
   $\dots = \dots A_2(f(i)) \dots$ 

   $A_4(g(i)) = \dots$ 
   $\triangleright \tau_w A_4(g(i)) = [i]^g$ 
   $\dots = \dots A_4(g(i)) \dots$ 
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  $\phi$  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  $\phi$  assignment defining  $A_5$  in terms of  $A_4$  and  $A_3$ , which in turn forces us to reinsert the  $\phi$  assignment for  $A_3$ . We thus wind up with the following code:

```

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

   $A_2(f(i)) = \dots$ 
   $\triangleright \tau_w A_2(f(i)) = [i]^g$ 
   $\triangleright A_3(\cdot) = \phi(A_2(\cdot), \tau_w A_2(\cdot); A_1(\cdot), \tau_w A_1(\cdot))$ 
   $\triangleright \tau_w A_3(\cdot) = \max(\tau_w A_2(\cdot), \tau_w A_1(\cdot))$ 
   $\dots = \dots A_2(f(i)) \dots$ 

   $A_4(g(i)) = \dots$ 
   $\triangleright \tau_w A_4(g(i)) = [i]^g$ 
   $\triangleright A_5(\cdot) = \phi(A_4(\cdot), \tau_w A_4(\cdot); A_3(\cdot), \tau_w A_3(\cdot))$ 
   $\triangleright \tau_w A_5(\cdot) = \max(\tau_w A_4(\cdot), \tau_w A_3(\cdot))$ 
   $\dots = \dots A_4(g(i)) \dots$ 
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  $\phi$  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  $\phi$  merge after an **if** construct looks like this:

```

...
A5(:) = ...

if  $B$  then
  A6( $\vec{k}$ ) = ...
  ▷ $\tau_w A_6(\vec{k})$  = ...
  ▷ $A_7 = \phi(A_6, \tau_w A_6; A_5, \tau_w A_5)$ 
  ▷ $\tau_w A_7 = \max(\tau_w A_6, \tau_w A_5)$ 
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  $\phi$  function:

```

...
A5(:) = ...

if  $B$  then
  A6( $\vec{k}$ ) = ...
  ▷ $\tau_w A_6(\vec{k})$  = ...
  ▷ $A_7 = \phi(A_6, \tau_w A_6; A_5, \tau_w A_5)$ 
  ▷ $\tau_w A_7 = \max(\tau_w A_6, \tau_w A_5)$ 
end if
▷forall ( $\vec{j}$ )
  ▷ $A_8(\vec{j}) = \begin{cases} A_7(\vec{j}) & \text{if } B \\ A_5(\vec{j}) & \text{otherwise} \end{cases}$ 
  ▷ $\tau_w A_8(\vec{j}) = \begin{cases} \tau_w A_7(\vec{j}) & \text{if } B \\ \tau_w A_5(\vec{j}) & \text{otherwise} \end{cases}$ 
▷end forall

```

Of course we can only do this if we ensure that the value of  $B$  introduced in the Boolean choice expression in the  $\phi$  function is the same as that which it had in the **if** guard. We can always arrange this by saving the value in a temporary if necessary.

This will be very useful to us; we assume this has been done systematically. The significance of this transformation is that the expression  $B$  contains no  $\tau_w$  variables. This allows the algorithm in Chapter 2 to propagate non-trivial values.

### 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. So first we give a precise definition of what has to be preserved:

Given a program  $P$ , let us temporarily denote by  $G_{\text{dynamic}}$  the directed graph defined as follows: Its nodes fall into two classes:

- the dynamic references in  $P$ . A dynamic reference (see also page 41 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  $G_{\text{dynamic}}$ . This is because  $\perp$  values of ordinary variables can never be referenced in a legal program.

We do not make this exception for  $\tau_w$  variables: as we remarked earlier, values of  $\perp$  for  $\tau_w$  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  $\phi$  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  $G_{\text{dynamic}}$ .

The edges of  $G_{\text{dynamic}}$  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 a source assignment statement to the reference being defined.
- There is an edge from each use of an ordinary variable in a  $\phi$  assignment which could be assigned to the reference being defined. There is also an edge from each corresponding  $\tau_w$  variable to the reference being defined.

(In general, this means that there is an edge from every argument in a  $\phi$  function to the reference being defined. However, in the course of performing program transformations, it may become clear that one or more of these edges can be eliminated.)

- 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 than 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  $G_{\text{dynamic}}$  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 constitute ultimate use.

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

As noted, this definition of ultimate use is conservative. A value 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).

Certainly any program transformation that we want to allow should preserve values that are ultimately used. There is a somewhat wider class of values and statements that actually need to be preserved for our purposes—we use the word *essential* to describe this class:

**Definition** *The value held by the array element  $A_i(\vec{k})$  at time  $t$  is essential if either*

- *the value is ultimately used, or*
- *there is a path in the graph  $G_{dynamic}$  starting from the reference to  $A_i(\vec{k})$  at time  $t$  and ending at an expression in a loop header or **if** guard.*

*A dynamic instance of a statement (that is, a statement  $S$  together with a time  $t$  at which it is executed) is essential iff either*

- *it is a **read** statement, or*
- *it contains a value that is essential.*

*(Even if a **read** statement does not assign to a variable that is essential, we regard it as essential, because it may affect what is read by subsequent **read** statements.)*

**Definition** *We say that a statement  $S$  in the program is dead code in any of the following cases:*

- *$S$  is never executed.*
- *$S$  is an empty **if** construct or loop—that is, there are no statements guarded by the **if** guard or in the body of the loop.*
- *$S$  is an assignment statement which assigns to an object none of the values of whose dynamic instances are essential.*

*As in the definition of essential statements above, we never characterize a **read** statement as dead code, even if it defines a variable that is never used. This is simply because we want to make sure that it is not deleted, thereby affecting the values read by any subsequent **read** statements.*

Now each transformation we have considered in Sections 1.3.1 and 1.3.2 falls into one of the following categories:

- T1.** 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 essential, and if it can be proved that at time  $t$ ,  $\tau_w A_i(\vec{k}) = \tau_w A_j(\vec{k})$  (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  $\tau_w A_i(\vec{k})$  in the same statement is also replaced by  $\tau_w A_j(\vec{k})$ . (This replacement of  $\tau_w A_i(\vec{k})$  only happens if the statement is a  $\phi$  assignment.)
  - If the value held by  $A_i(\vec{k})$  at time  $t$  is *not* essential, then
    - If the statement is not a  $\phi$  assignment, it is permissible to replace the use of  $A_i(\vec{k})$  by a use of any  $A_j(\vec{k})$ , provided that
      - \* this substitution does not lead to a run-time exception either at time  $t$  or later.
    - If the statement is a  $\phi$  assignment, it is permissible to replace the use of  $A_i(\vec{k})$  by a use of any  $A_j(\vec{k})$  and to replace the corresponding use of  $\tau_w A_i(\vec{k})$  in the same statement by a corresponding use of  $\tau_w A_j(\vec{k})$  provided that
      - \*  $\tau_w A_j(\vec{k}) \leq \tau_w A_i(\vec{k})$  at time  $t$ , and
      - \* this substitution does not lead to a run-time exception either at time  $t$  or later.
 (This substitution in a  $\phi$  assignment cannot lead to a run-time exception at time  $t$ , but it still might lead to one later.)

In applying this criterion, remember that in accordance with our convention,  $\vec{k}$  does not refer to an expression; it refers to a particular value. In general, of course, we are interested in *all* the possible values that occur in the course of execution of the program, because we usually want to replace an entire expression textually. However, our criterion is expressed in terms of single values to allow more flexibility.

- T2.** A Boolean expression in a  $\phi$  function after an **if** construct may be replaced, as in Section 1.3.2, by the Boolean expression which is the **if** guard (or by a temporary containing that value).
- T3.** An argument to a  $\phi$  function can be eliminated if it can be proved that doing so does not change the value of the  $\phi$  function.
- T4.** In conjunction with actions taken because of items 1, 2, and 3, dead code (in the sense defined above) may be eliminated. When dead code is eliminated,
- The corresponding elements of the executed global iteration space are also eliminated.
  - If the eliminated code is an assignment statement assigning to an ordinary variable, then any original source statement containing a use of that variable must also be dead code, and is also eliminated.

**Definition** An admissible transformation is a program transformation of one of these four forms.

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

1. The executed global iteration space of  $P'$  is a subset of that of  $P$ .

2. Every dynamic instance of a **read** or **print** statement in  $P$  occurs also in  $P'$  and conversely.

PROOF. 1. Since no admissible transformation can change the value of any expression in a loop header or **if** guard, the executed global iteration space of  $P'$  must be a subset of that of  $P$ . In fact, it is identical to that of  $P$  unless dead code is eliminated.

2. It is also clear that since **read** and **print** statements are essential, they cannot be deleted (and no construct including them, such as loops or **if** constructs, can be deleted) by any admissible transformation.  $\square$

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

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

$$\tau_w A'(\vec{k}) \leq \tau_w A(\vec{k})$$

If the value held by  $\tau_w A(\vec{k})$  at time  $t$  is essential, then

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

2. 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 essential, then

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

PROOF. By Theorem 1.8 the executed global iteration space of  $P'$  is a subset of that of  $P$ . It therefore suffices to walk through the sequence of computations executed in  $P$  and compare them to what (if anything) happens in  $P'$  at the corresponding times.

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

We therefore let  $t$  run over the times in the executed global iteration space of  $P$  and we make the inductive assumption that at all times before time  $t$  the conditions of the theorem are satisfied.

Let  $S$  be the statement executed in  $P$  at time  $t$ . Either  $S$  is dead code which is deleted by transformation **T4**, in which case it is not represented in  $P'$  at all, or  $S$  corresponds to a statement  $S'$  in  $P'$  which is either the same as  $S$  or is derived from  $S$  by an admissible transformation.

Suppose first that  $S$  is dead code that is deleted in  $P'$ . There are three possibilities:

**$S$  is never executed in  $P$ .** Certainly deleting  $S$  in  $P'$  then has no effect whatsoever, so the statements of the theorem remain true.

**$S$  is an empty loop or an empty if construct.** In such a case, clearly no variable changes its value when  $S$  is deleted, so the statements of the theorem remain true.

**$S$  is an assignment statement.** In such a case, the value of the variable defined by  $S$  at time  $t$  (whether ordinary or  $\tau_w$ ) cannot be essential. When the statement is deleted, that value becomes  $\perp$ . It is immediate that the statements of the theorem remain true in this case as well.

So we may henceforth assume that  $S$  has not been deleted and so corresponds as above to a statement  $S'$  in  $P'$ . There are five possibilities:

**$S$  is a print statement.** By the inductive hypothesis, the value of the expression being printed is the same in  $P'$  as in  $P$ .

**$S$  is a read statement.** Since none of the admissible transformations apply to a **read** statement, and since all previously executed **read** statements in  $P$  have been executed in  $P'$  as well, the statement reads the same value in  $P'$  as in  $P$  and so assigns the same value to its variable.

**$S$  is a source assignment.** There are two possibilities:

**$S$  assigns to an ordinary variable  $A(\vec{k})$ .** If the value of  $A(\vec{k})$  at time  $t$  is essential, then all the references used in the assignment statement are essential and hence by the inductive hypothesis have the same values in  $P'$  as they do in  $P$ . The only possible transformation that could be applied to  $S$  is **T1**, and this would again preserve all the values used in  $S$ . So  $S'$  produces the same value as  $S$ .

**$S$  assigns to a variable  $\tau_w A(\vec{k})$ .** In such a case,  $\tau_w A(\vec{k})$  is set equal to the current time, which is the same in  $P'$  as in  $P$ , so  $\tau_w A'(\vec{k}) = \tau_w A(\vec{k})$ .

**$S$  is a  $\phi$  assignment.** There are two possibilities:

**$S$  assigns to a variable  $\tau_w A_i(\vec{k})$ .** If transformation **T2** is applied to  $S$ , then the same value is automatically produced for  $\tau_w A'_i(\vec{k})$  in  $P'$  as for  $\tau_w A_i(\vec{k})$  in  $P$ .

In any other case, let  $\tau_w A_j(\vec{k})$  be any one of the  $\tau_w$  variable arguments to the  $\phi$  function. One of the following must be true:

- No transformation is applied that affects  $\tau_w A_j(\vec{k})$ . So  $\tau_w A'_j(\vec{k})$  occurs as an argument to the  $\phi$  function in  $P'$ . In this case, by the inductive hypothesis,  $\tau_w A'_j(\vec{k}) \leq \tau_w A_j(\vec{k})$ . Further, if  $\tau_w A_j(\vec{k})$  is essential, then again by the inductive hypothesis,  $\tau_w A'_j(\vec{k}) = \tau_w A_j(\vec{k})$ .
- Transformation **T1** is used to replace the argument  $\tau_w A_j(\vec{k})$  by  $\tau_w A'_k(\vec{k})$ . When this transformation is made, we must have  $\tau_w A'_k(\vec{k}) \leq \tau_w A_j(\vec{k})$ . Further, if  $\tau_w A_j(\vec{k})$  is essential, then we must have  $\tau_w A'_k(\vec{k}) = \tau_w A_j(\vec{k})$ .
- Transformation **T3** is used to drop the argument  $\tau_w A_j(\vec{k})$  from the  $\phi$  function, without replacement. Of course this cannot happen if  $\tau_w A_j(\vec{k})$  is essential.

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



$S$  assigns to an ordinary variable  $A_i(\vec{k})$ . We only have to consider the case in which  $A_i(\vec{k})$  is essential.

If transformation **T2** is used, then as above, the value assigned to  $A_i(\vec{k})$  is automatically preserved in  $P'$ .

In any other case the  $\tau_w$  variable corresponding to the variant of  $A_i(\vec{k})$  which is assigned to  $A_i(\vec{k})$  by the  $\phi$  function must also be essential. The same reasoning as above then shows that the same value will be picked for  $A'_i(\vec{k})$  as for  $A_i(\vec{k})$ .

**S is a loop or an if construct.** We have already noted that these statements are unchanged by any transformation (unless they are deleted, and such deletion is harmless).

The assertions of the theorem therefore remain true at time  $t$ , and hence by induction remain true throughout the execution of  $P'$ .  $\square$

**1.10 Theorem** *Let  $P$  be an original source program, let  $P_w$  be the initial Array SSA form of  $P$ , and let  $P'$  be a program which is the result of applying a sequence of admissible transformations to  $P_w$ .*

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

*If  $\tau_w A_i$  is a variable in  $P'$  and if its value at time  $t$  is essential, then the value is equal to that in  $P_w$  at time  $t$ .*

PROOF. Let  $P_j$  ( $0 \leq j \leq n$ ) be a sequence of programs such that

- $P_0 = P_w$ .
- $P_{j+1}$  is produced from  $P_j$  by the action of one admissible transformation.
- $P_n = P'$ .

We already know that the result is true for  $P_0$  (that is, when  $P' = P_0$ ), even without the qualification that the value of  $A_i(\vec{k})$  at time  $t$  is essential.

We proceed by induction. Suppose the result is true for  $P_j$ , and suppose that the value of  $A_i(\vec{k})$  at time  $t$  in  $P_{j+1}$  is essential.

First let us note that if we can show that if the value of  $A_i(\vec{k})$  was already essential at time  $t$  in  $P_{j+1}$ , then by the inductive hypothesis together with Theorem 1.9(2),  $A_i(\vec{k})$  has the same value in  $P'$  at time  $t$  as it does in  $P$ .

The transformation that was applied to  $P_j$  to produce  $P_{j+1}$  was one of the four admissible transformations. We consider them separately:

**T3 or T4:** No transformation of either of these two forms can change an inessential value into an essential one. Therefore since the value of  $A_i(\vec{k})$  at time  $t$  is essential in  $P'$ , it must already have been essential in  $P$ . The result then follows as above.

**T2:** There are three possibilities:

1.  $A_i(\vec{k})$  is in neither the original Boolean expression or the new one. In this case, the value of  $A_i(\vec{k})$  at time  $t$  must have been essential in  $P_j$ , since the transformation did nothing that could have affected this property.
2.  $A_i(\vec{k})$  is in the original Boolean expression but not in the new one. Since the value of  $A_i(\vec{k})$  at time  $t$  is essential in  $P'$ , it must also have been essential in  $P$  because the additional use of  $A_i(\vec{k})$  in  $P$  cannot have made the value inessential.
3.  $A_i(\vec{k})$  is not in the original Boolean expression but is in the new one. In this case, there are two possibilities:
  - (a) The fact that the value of  $A_i(\vec{k})$  at time  $t$  is essential in  $P'$  is independent of its use in the Boolean expression. In this case, the value must have been essential in  $P'$ , since nothing else changed.
  - (b) The value of  $A_i(\vec{k})$  at time  $t$  in  $P'$  occurs on a path from the Boolean expression to a **print** statement, a loop header, or an **if** guard.  
We know, in this case, that  $A_i(\vec{k})$  is also in the **if** guard which became that new Boolean expression. And there must be a similar path from that original **if** guard that makes every value in it essential. So again the value  $A_i(\vec{k})$  at time  $t$  must already have been essential in  $P$ .

Therefore the value of  $A_i(\vec{k})$  at time  $t$  was already essential in  $P$  and so the result follows as before.

**T1:** Suppose that the value of  $A_i(\vec{k})$  at time  $t$  was not essential in  $P$ . Then in  $P'$  there is a path in  $G_{\text{dynamic}}$  from  $A_i(\vec{k})$  that makes  $A_i(\vec{k})$  essential, and this path did not exist in  $P$ . Since this path was introduced by a substitution of the form **T1**, it has the form illustrated on the left-hand side of Figure 1.7. The substitution (in this case of  $B_q(\vec{u})$  for  $B_p(\vec{u})$ ) is indicated by the dashed line.  $B_p(\vec{u})$  is essential, and  $A_i(\vec{k})$  occurs somewhere on the path to  $B_q(\vec{u})$ . The introduction of the dashed edge (which is a new edge in  $G_{\text{dynamic}}$  makes everything on the path leading up to  $B_q(\vec{u})$  essential.

First note that if in fact  $A_i(\vec{k})$  is  $B_q(\vec{u})$  then we are done, because the substitution requires that  $B_q(\vec{u})$  has the same value as  $B_p(\vec{u})$ . Since the value of  $B_p(\vec{u})$  at time  $t$  is essential, it is (by the inductive hypothesis) equal to the value of  $B(\vec{u})$  (which is in this case  $A(\vec{u})$  in the original program at time  $t$ , and therefore, the same holds for  $B_p(\vec{u}) = A_i(\vec{k})$ , and we are done. So let us assume that  $A_i(\vec{k})$  is not the same as  $B_q(\vec{u})$ . It therefore occurs somewhere on the path in  $G_{\text{dynamic}}$  leading to  $B_q(\vec{u})$ , as illustrated on the left-hand side of the figure.

Now in order for this substitution to be of the form **T1**, it must be the case that  $\tau_w B_p(\vec{u}) = \tau_w(B_q(\vec{u}))$  at time  $t$ . Therefore, the values in  $B_p(\vec{u})$  and  $B_q(\vec{u})$  entered  $webB_p$  at the same time by an assignment to (say)  $B_r(\vec{u})$ . This assignment must be on the path in  $G_{\text{dynamic}}$  leading to the expression involving  $B_p(\vec{u})$ . This assignment must also be on the path leading from the reference to  $A_i(\vec{k})$  to  $B_q(\vec{u})$ . Therefore we know that the graph really looks like that on the right-hand side of Figure 1.7. It is then clear that every dynamic reference on the path through  $A_i(\vec{k})$  is already essential in  $P$ , and therefore has the same value in  $P$  as in  $P'$ .

Further, every reference on the path from  $B_r(\vec{u})$  to  $B_q(\vec{u})$  has the same value, since this path is a path composed entirely of  $\phi$  assignments. And we know that the value of  $B_q(\vec{u})$  at time  $t$  is the same as that of  $B_p(\vec{u})$  at time  $t$ .

Therefore every dynamic reference that becomes essential by the substitution of  $B_q(\vec{u})$  for  $B_p(\vec{u})$  has the same value in  $P'$  as it did in  $P$ .

Exactly the same proof (somewhat simpler, in fact) works for  $\tau_w$  variables. □

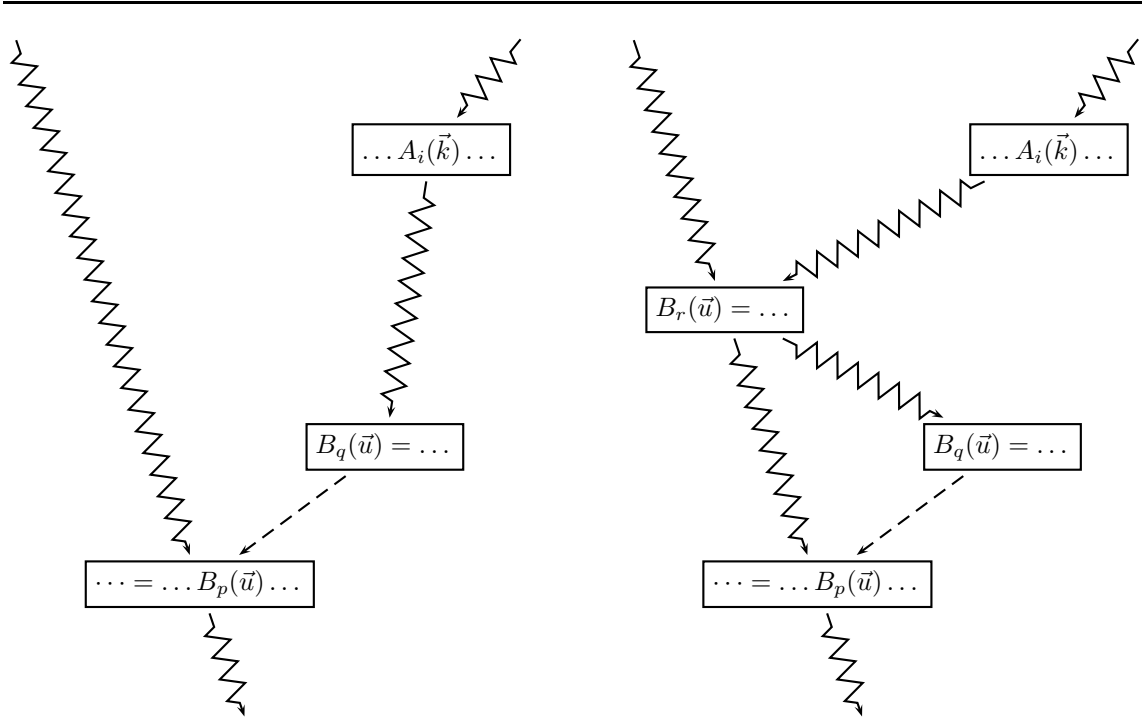


Figure 1.7:  $G_{\text{dynamic}}$  subgraphs referred to in the proof of Theorem 1.10

**1.11 Theorem** *A program which is the result of applying a sequence of admissible transformations to a program in initial Array SSA form reads the same input and produces the same output as in the original program.*

PROOF. We have already seen that admissible transformations preserve the sequence and number of **read** and **print** statements. The expressions printed are all essential by definition, and so by Theorem 1.10 these expressions have the same values as in the original program. □

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

- C1.** *The program must continue to satisfy the property described in Lemma 1.7 (page 23).*
- C2.** *If the right-hand side of an assignment statement contains a  $\tau_w$  variable (so in particular this is a statement that is a  $\phi$  assignment), then it contains a  $\tau_w$  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.)

We have seen in Theorem 1.11 that an admissible program has the same visible behavior as the original source program from which it was derived.

We continue to say that an admissible program is in Array SSA form.

## 1.4 General properties of $\tau_w$ and $\tau_v$ variables

The properties of  $\tau_w$  and  $\tau_v$  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  $\tau_v$  and  $\tau_w$  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  $\tau_v$  and  $\tau_w$  variables are not changed in this transformation. For ease of reference we restate those results here:

- 1.12 Lemma** *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  $\vec{k}$ , then (at that same point)*

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

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

### 1.4.2 Properties that are true for admissible programs

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

**1.14 Lemma** 1. For any ordinary array element  $A_i(\vec{k})$ , the values of  $\tau_v A_i(\vec{k})$  and  $\tau_w A_i(\vec{k})$  are both non-decreasing in time.

2. At any point in the course of execution of the program, the value in  $\tau_w A_i(\vec{k})$  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,

$$\tau_w A_i(\vec{k}) \leq \tau_v A_i(\vec{k})$$

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  $\phi$  assignment, then it is defined as a max of other  $\tau_w$  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.  $\square$

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  $\phi$  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 values being essential:

Lemma 1.12 remains true for admissible programs provided that the values of  $\tau_w A(\vec{k})$  and  $\tau_w B(\vec{k})$  at the time in question are both essential:

**1.15 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 the values  $\tau_w A(\vec{k})$  and  $\tau_w B(\vec{k})$  at that time are both essential, then (at that same time)

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

PROOF. We know that the values of  $\tau_w A(\vec{k})$  and  $\tau_w B(\vec{k})$  are the same as in the initial Array SSA form of the program. We also know that the inequalities are true in the initial Array SSA form.

Further, an admissible transformation cannot increase the value of  $\tau_v A(\vec{k})$ , and by Lemma 1.14(2), the two outer inequalities are always true. So all the inequalities remain true in the transformed program  $P'$ .  $\square$

Lemma 1.13 remains true for admissible programs provided that in the second part we restrict ourselves to values that are essential:

**1.16 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  $\text{web}(A_i)$ .*
2. *If on the other hand  $A_i$  is defined by a  $\phi$  assignment, then for each execution of that assignment statement, if  $\tau_w A_i(\vec{k})$  is an element of  $\tau_w A_i$  whose value is essential, 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  $\text{web}(A_i)$ .*

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 Theorem 1.10 (page 34) that since the value of  $\tau_w A_i(\vec{k})$  is essential, its value is the same as the value it started out with in the original initial Array SSA form of the program. And by Theorem 1.9, 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.  $\square$

To see why we need the stipulation that  $\tau_w A_i(\vec{k})$  be essential in part 2 of the theorem, consider the code in Figure 1.8.

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 1.16, except that (assuming there is no other code using  $A_5$ ) the value  $\tau_w A_5(1)$  is not essential. (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 6.) So  $P(\tau_w A)$  is the statement at which the value ultimately assigned to  $A$  at time  $\tau_v A$  enters  $\text{web}(A)$ .

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

---

```

A0(:) = ...
do i = 1, 2 * n - 1, 2
    A1(i) = ...
    ▷A2 = φ(A0, τwA0; A1, τwA1)
end do
▷A3 = φ(A2, τwA2; A0, τwA0)
A4(2) = 0
▷A5 = φ(A4, τwA4; A3, τwA3)
do i = 1, n
    ...
    ... = ... A5(2 * i) ...
    ...
end do

```

Figure 1.8: Code that illustrates why we need to restrict ourselves to values that are essential. The assignments to  $\tau_w$  variables have been suppressed for clarity; they occur in the actual code.

---

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 essential, 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  $\text{web}(A_i)$ .
2. Suppose  $S$  is a  $\phi$  assignment defining  $\tau_w A_i$ . If  $\tau_w A_i(\vec{k})$  is defined by  $S$  at time  $t$  and is essential, then one of the arguments to the  $\phi$  function in  $S$  has a value that is maximal among all the values  $\{\tau_w A_j(\vec{k}) : A_j \in \text{web}(A_i)\}$  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 1.13,  $\tau_w A_i(\vec{k})$  is the maximum value of  $\tau_w A_j(\vec{k})$  for all  $A_j$  in  $\text{web}(A_i)$ .

Theorem 1.9 (page 32) 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 1.16: 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  $\text{web}(A_i)$ , 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  $\phi$  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 afterward.

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, and the program continues to satisfy the constraints **C1** and **C2** on page 37.) Any computed value in the original program that is essential is computed to be the same in  $P$ . Values that were not essential 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<sup>1</sup>, and

---

<sup>1</sup>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  $\tau_w$  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  $\tau_w$  variable, however, the value  $\perp$  is meaningful, and so in a weak DSA program, a  $\tau_w$  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  $\tau_w$  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) suppose the original program looked like this:

```

do  $j = 1, n$ 
  do  $i = 1, n$ 
     $A(i, i + j - 2) = i$ 
    ...
    ... = ...  $A(i, j)$  ...
  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  $i = 1, n$ 
     $A(i, i + j - 2) = i$ 
    ...
    ... = ...  $A(i, i + j - 2)$  ...
  end do
end do

```

then again only 1 dimension is needed<sup>2</sup>.

- 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

---

<sup>2</sup>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  $\phi$  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  $\tau_v$  variables into the code. We can always eliminate explicit references to  $\tau_v$  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  $\mathcal{C} = \{d_{j_1}, \dots, d_{j_p}, i_{k_1}, \dots, i_{k_q}\}$ , 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  $A(\sigma(d_1), \dots, \sigma(d_m))$  to  $A$  is rewritten as

$$A^{\mathcal{C}}(\sigma(d_{j_1}), \dots, \sigma(d_{j_p})) [\tau_v A(\sigma(d_1), \dots, \sigma(d_m))_{k_1}, \dots, \tau_v A(\sigma(d_1), \dots, \sigma(d_m))_{k_q}]$$

As usual, the notation

$$\tau_v A(\sigma(d_1), \dots, \sigma(d_m))_{k_1}$$

refers to the  $k_1^{\text{th}}$  component of the  $q$ -tuple  $\tau_v A(\sigma(d_1), \dots, \sigma(d_m))^{\text{loc}}$ .

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

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

This is the reason we defined  $\sigma$  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.

Of course this rewriting only makes sense if the  $\tau_v$  values in the added iteration coordinates actually have meaningful values where they are needed. For this reason, we make the following restriction:

*The subspace  $\mathcal{C}$  used to rewrite a variable which is live on entry is simply that variable’s data space.*

This restriction is explicitly satisfied by the algorithm we will present later for computing sets of contributing dimensions (which is what the subspaces  $\mathcal{C}$  will be).

While this rewriting may at first look somewhat strange and intimidating, it will become evident that it is really quite natural. On page 53 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  $\tau_w$  variable (say  $A = \tau_w B$ ), then the rewriting would introduce variables of the form  $\tau_v \tau_w B$ . As we have seen in Lemma 1.14(3) (page 38), such a variable can be simply written as  $\tau_v B$ , 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  $\mathcal{C} = \emptyset$  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*

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

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

$$A^{\mathcal{C}}(\sigma(d_{j_1}), \dots, \sigma(d_{j_p}))[\tau_v A(\sigma(d_1), \dots, \sigma(d_m))_{k_1}, \dots, \tau_v A(\sigma(d_1), \dots, \sigma(d_m))_{k_q}]$$

*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  $\tau_w$  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  $R_{\text{def}}$  and  $R_{\text{use}}$  denote dynamic references to fixed elements of  $A$  in program  $P$ , and let  $R'_{\text{def}}$  and  $R'_{\text{use}}$  denote dynamic references to  $A^{\mathcal{C}}$  in program  $P'$ . We assume that  $R_{\text{use}}$  and  $R'_{\text{use}}$  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,  $R'_{\text{use}}$  has the same value as  $R'_{\text{def}}$ .*

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

Figure 2.3: Four dynamic references used in 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  $R_{\text{use}}$  has the same value as  $R_{\text{def}}$ .



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

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

$$\begin{aligned} R_{\text{def}} &= A(\sigma^{\text{def}}(d_1), \dots, \sigma^{\text{def}}(d_m)) \\ R_{\text{use}} &= A(\sigma^{\text{use}}(d_1), \dots, \sigma^{\text{use}}(d_m)) \end{aligned}$$

Since  $R_{\text{def}}$  and  $R_{\text{use}}$  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  $R_{\text{def}}$  occurs at the point  $[v(i_1), \dots, v(i_n)]$  in its local iteration space in  $P$ , and the value of  $\tau_v A$  at  $R_{\text{use}}$  is

$$\tau_v A(v(d_1), \dots, v(d_m))^{\text{loc}} = [\tau_v A(v(d_1), \dots, v(d_m))_1, \dots, \tau_v A(v(d_1), \dots, v(d_m))_n]$$

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

Now the value of  $R'_{\text{def}}$  is

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

Here the quantity in brackets is just the position of  $R_{\text{def}}$  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  $R'_{\text{def}}$ .

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

$$A^{\mathcal{C}}(v(d_{j_1}), \dots, v(d_{j_p})) [\tau_v A(v(d_1), \dots, v(d_m))_{k_1}, \dots, \tau_v A(v(d_1), \dots, v(d_m))_{k_q}]$$

We see that the quantity in brackets is just the position  $\tau_v A(v(d_1), \dots, v(d_m))^{\text{loc}}$  of  $R_{\text{def}}$  in its local iteration space  $\mathcal{I}$ , projected onto the local iteration space component  $\mathcal{C} \cap \mathcal{I}$  of the same  $\mathcal{C}$ .

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

2. It follows that the value of  $R'_{\text{use}}$  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  $R'_{\text{def}}$ . And one of these values is the value of  $R'_{\text{def}}$  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  $R'_{\text{def}}$  are the same. We therefore conclude that the value of  $R'_{\text{use}}$  is the same as the value of  $R'_{\text{def}}$ .  $\square$

**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  $A^C$  for all times before  $t$ , and
- either
  - $A$  is an ordinary variable, or
  - $A$  is a  $\tau_w$  variable and has the property that if for any time  $s \leq t$ ,  $P$  and  $P'$  are iteratively similar up through time  $s$  and if  $R'_{use}$  is a use of  $A^C$  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  $R$ .

(By our assumption, any variable live on entry is not changed by rewriting and so is the same in  $P'$  as in  $P$ , which is why this part of the inductive assumption is true at the start of the program.)

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  $R_{\text{def}}$  in the notation used in the lemma, we have just shown (simply by the inductive assumption) that the value of  $R_{\text{def}}$  is the same as the value of  $R'_{\text{def}}$ .

**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  $A^C$  in  $P'$ . If  $A$  is a  $\tau_w$  variable and the value of  $R$  is not  $\perp$ , then  $R$  is a use of an element of  $A$  defined at a previous definition  $R_{\text{def}}$ . By the lemma,  $R'$  has the same value as the corresponding definition  $R'_{\text{def}}$  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  $R'$ , which now may be an ordinary variable or a  $\tau_w$  variable, 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  $\tau_w$  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  $R_{\text{use}}$  and  $R'$  as  $R'_{\text{use}}$ . We then need to show that the value of  $R'_{\text{use}}$  is the same as the value of  $R_{\text{use}}$ . Now we know by definition that the value of  $R_{\text{use}}$  is the same as the value of  $R_{\text{def}}$ . Case I (second sub-case) showed that the value of  $R_{\text{def}}$  is the same as that of  $R'_{\text{def}}$ . (These definitions must have been considered previously in the induction.) And we showed in the lemma that the value of  $R'_{\text{def}}$  is the same as that of  $R'_{\text{use}}$ . This proves the result.

This shows that property C is satisfied. If the references  $R_{\text{use}}$  and  $R'_{\text{use}}$  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  $C$  for the array  $A$  in  $P$ , and if*

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

*then  $C$  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  $C$  puts the program in weak DSA form with respect to  $A$ .  $\square$

For example, consider the following source code:

```

do  $i = 1, n$ 
   $A(f(i)) = i$ 
  ...
  ... = ...  $A(g(i))$  ...
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  $i = 1, n$ 
   $A_4(:) = \dots$ 
   $\triangleright_{\tau_w} A_4(:) = \dots$ 
   $A_5(f(i)) = i$ 
   $\triangleright_{\tau_w} A_5(f(i)) = [i]^g$ 
  forall ( $k$ )
     $\triangleright A_6(k) = \begin{cases} A_5(k) & \text{if } k = f(i) \\ A_4(k) & \text{otherwise} \end{cases}$ 
     $\triangleright_{\tau_w} A_6(k) = \begin{cases} \tau_w A_5(k) & \text{if } k = f(i) \\ \tau_w A_4(k) & \text{otherwise} \end{cases}$ 
  end forall
  ...
  ... = ...  $A_6(g(i))$  ...
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  $i = 1, n$ 
   $A_4(\cdot) = \dots$ 
   $\triangleright \tau_w A_4(\cdot) = \dots$ 
   $A_5^C[i] = i$  ! Substitution 1
   $\triangleright \tau_w A_5(f(i)) = [i]^g$ 
  forall ( $k$ )
     $\triangleright A_6^C[i, k] = \begin{cases} A_5^C[i] & \text{if } k = f(i) \\ A_4(k) & \text{otherwise} \end{cases}$  ! Substitution 2
     $\triangleright \tau_w A_6(k) = \begin{cases} \tau_w A_5(k) & \text{if } k = f(i) \\ \tau_w A_4(k) & \text{otherwise} \end{cases}$ 
  end forall
   $\dots$ 
   $\dots = \dots A_6^C[i, g(i)] \dots$  ! Substitution 3
end do

```

where the indicated substitutions are made as follows:

**Substitution 1:**  $\tau_v A_5(f(i)) = [i]$ , so  $A_5(f(i))$  is replaced by  $A_5^C[\tau_v A_5(f(i))] = A_5^C[i]$ .

**Substitution 2:** On the right-hand side,  $\tau_v A_5(k)$  is only needed when  $k = f(i)$ , and we know that  $\tau_v A_5(f(i)) = [i]$ .

On the left-hand side,  $\tau_v A_6(k) = [i, k]$ .

**Substitution 3:**  $\tau_v A_6(g(i)) = [i, g(i)]$

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 $\tau_v$ variables

In the rewriting of the last example, all explicit references to  $\tau_v$  variables were eliminated. This will always be true. We give part of the explanation here.

Let us consider a reference to a variable  $A$ , which may be an ordinary variable or a  $\tau_w$  variable. A reference to  $\tau_v A$  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  $\tau_v A$  that appears. Let  $I$  denote the loop variable corresponding to the iteration dimension  $i_k$ . Since  $i_k$  is in  $\mathcal{C}$ , it must be in the local iteration space at the definition of  $A$ . That is, the definition of  $A$  must be inside the  $I$  loop.

We will show that if certain constraints are satisfied, the reference to  $\tau_v A(\vec{k})_{i_k}$  can be replaced by an explicit expression. The constraints are these:

- For each path from the definition of an array element whose value is essential 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  $\tau_v$  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 that the value of the array element be essential). 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  $\tau_v A(\vec{k})_{i_k}$  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  $\tau_v A(\vec{k})_{i_k}$ .

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

So now we may assume that the reference is a use of  $A$ . We will further assume that the value of the element of  $A$  is actually essential, since otherwise we really don't care about the reference at all—in particular, we don't care which element it refers to. (We do have to be careful not to introduce arithmetic exceptions. We assume we can do this.)

If  $A$  was defined by a  $\phi$  assignment, then the value of  $\tau_v A$  (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  $\phi$  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  $\phi$  assignment. The iteration variable of that  $\phi$  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  $\tau_v A$ . 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  $\tau_v A(\vec{k})_{i_k} = I$ .
2. If the reference occurs in the  $I$  loop lexically before the definition of  $A$ , then

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

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

Thus, while we use the notation  $\tau_v$ , 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  $R = A(\sigma(d_1), \dots, \sigma(d_m))$  is an essential dynamic reference, and if the conditions at the top of this section are satisfied, then each component of  $\tau_v A(\sigma(d_1), \dots, \sigma(d_m))$  is determined as follows:*

1. *If the component is in a  $\phi$  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  $\tau_v A(\sigma(d_1), \dots, \sigma(d_m))$  is  $\perp$ )*

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

The reader may be concerned that the three conditions needed for this theorem are unreasonably stringent. It is true that they are stringent, but there are two things to notice:

1. If we really need to apply this theorem, then the algorithm to be presented will ensure that these conditions apply, by reintroducing enough of the naive array SSA overhead to make the conditions true.
2. In many cases, however, we don't actually care, because there are no iteration dimensions in  $\mathcal{C}$  that need to be introduced. If iteration dimensions really need to be introduced, this indicates that the program really is quite complex in its data access patterns, and the overhead that we need to reintroduce is not at all spurious.

### 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  $\emptyset$  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  $\emptyset$  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  $e$ .  $\square$

**2.7 Lemma** If  $e$  is a function of the set of expressions  $\{e_1, e_2, \dots, e_n\}$  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  $\{e_1, \dots, e_n\}$ , which in turn determine the value of  $e$ .  $\square$

Of course in applications of this lemma one would choose the set  $\{e_1, \dots, e_n\}$  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

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

is a valid iteration subspace for  $e$ .<sup>3</sup>

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  $A$ —

$$(2.1) \quad \mathcal{S} \leftarrow \bigcup_{m \in \mathcal{C}(R)} \mathcal{S}(\sigma(m))$$

defines a valid iteration subspace for  $R$ .

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

$$A^{\mathcal{C}}(\sigma(d_{j_1}), \dots, \sigma(d_{j_p})) [\tau_v A(\sigma(d_1), \dots, \sigma(d_m))_{k_1}, \dots, \tau_v A(\sigma(d_1), \dots, \sigma(d_m))_{k_q}]$$

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  $\mathcal{S}$ .  $\square$

**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  $LHS$ .

<sup>3</sup>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  $R$ .



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

PROOF. Any loop index that cannot affect the value of RHS cannot affect the value of LHS either.  $\square$

**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  $A_k$ .  $\square$

## 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

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

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
- $A(3 + B(C(i + j)), f(j + x) - 11)$
- 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  $A$ ,  $\mathcal{C}(\tau_w A)$  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  $\emptyset$ .
- 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  $\emptyset$ .

**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  $\phi$  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

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

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  $\phi$  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 45.) Set  $\mathcal{C}(A) \leftarrow \mathcal{C}_1$ .

### Remarks

- The reason for the maximal initialization of  $\tau_w$  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  $\tau_w$  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  $\phi$  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  $\tau_v$  variables can be replaced by expressions not involving  $\tau_v$  variables, thereby eliminating all  $\tau_v$  variables from the program. If this were not possible, then the  $\tau_v$  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.  $\square$

In reading the next lemma, it is helpful to consider the program fragment

```

A0 = ...
do i = 1, n
  ▷A1 = φ(A0, A3)

  A2(i) = i
  ▷forall (k = 1 : n)
    ▷A3(k) =  $\begin{cases} A_2(k) & \text{if } k = i \\ A_1(k) & \text{otherwise} \end{cases}$ 
  ▷end forall

  ...
  ... = ... A3(f(i)) ...
  ...
end do

```

The lemma generalizes the fact that in this example our algorithm will produce  $\mathcal{C}(A_3) = \{d_1, i_1\}$ , where  $d_1$  is the (only) data dimension of  $A_3$  and  $i_1$  is the iteration dimension corresponding to the loop variable  $i$ .

**2.12 Lemma** *Suppose a variable  $A$  (which may be an ordinary variable or a  $\tau_w$  variable) is defined at a  $\phi$  assignment. (So each argument  $E$  of the  $\phi$  function is either an ordinary variable or a  $\tau_w$  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  $\phi$  function, together with*
- *each iteration dimension  $i_j$  of each  $\mathcal{C}(E)$  where  $E$  runs over those arguments of the  $\phi$  function such that both  $E$  and  $A$  are defined within the loop corresponding to  $i_j$ .*

PROOF. The  $\phi$  assignment looks like this:

```

forall (k1, k2, ..., kn)
  A(k1, k2, ..., kn) = φ(..., E(k1, k2, ..., kn), ...)
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  $\phi$  iteration coordinates that correspond to data dimensions in  $\mathcal{C}(E)$ .

Further, if  $E$  is an argument of the  $\phi$  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,  $\mathcal{S}(\sigma(i_j))$  is contained in  $\mathcal{S}(E)$ . It is therefore enough to show that  $\mathcal{S}(\sigma(i_j)) = \{i_j\}$ .

Now by action 5 the conclusion of Theorem 2.5 holds. Therefore,  $\sigma(i_j)$ , which is  $\tau_v E(k_1, k_2, \dots, k_n)_j$ , is either  $I(i_j)$  or  $I(i_j) - 1$ . (It can't be constant, because the loop corresponding to dimension  $i_j$  contains both the definition of  $E$  and the  $\phi$  assignment which uses it.) Therefore,  $\mathcal{S}(\sigma(i_j)) = \{i_j\}$ , 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  $i_j$ .  $\square$

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

- *The data dimensions in  $\mathcal{C}(\tau_w B_j)$  include all the dimensions of  $\tau_w B_j$  (which are also all the declared dimensions of  $B_i$ ).*
- *The iteration dimensions in  $\mathcal{C}(\tau_w B_j)$  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  $\mathcal{C}(\tau_w B_i)$  includes all the data and iteration dimensions at the definition 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  $\tau_w B_i$  to the definition of  $\tau_w B_j$ , 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  $\tau_w B_p$  to a use of  $\tau_w B_p$ . Nothing really happens on such an edge as far as  $\mathcal{C}(\tau_w B_p)$  is concerned, since  $\mathcal{C}(\tau_w B_p)$  is an attribute of the variable  $\tau_w B_p$ , not of a particular reference to it.
2. An edge from a use of  $\tau_w B_p$  as an argument to a  $\phi$  function defining  $\tau_w B_q$ . Lemma 2.12 shows that all the data dimensions together with the iteration dimension  $i_m$  are propagated to  $\mathcal{C}(\tau_w B_q)$ .  $\square$

**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  $\tau_w B$  in  $P$  at time  $t$ ,*
- *$R'$  is the corresponding dynamic use of  $\tau_w B^{\mathcal{C}(\tau_w B)}$  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  $\tau_w B^{C(\tau_w B)}$ , 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  $\perp$ .  $\square$

**2.15 Lemma** *If  $P'$  has the weak dynamic single assignment property for all times before  $t$  (that is, property  $A$  holds for all times before  $t$ ), then properties  $B$  and  $C$  hold up through time  $t$ .*

PROOF. We proceed by induction. To clarify the notation, let  $s_0 < s \leq t$  be elements of the executed global iteration space of the program  $P$ , and say  $s$  is the successor of  $s_0$ .

The inductive hypothesis  $H(\tau)$  (where  $\tau$  is any time in the executed global iteration space of  $P$ ) is the following statement:

1.  $P$  and  $P'$  are iteratively similar up through time  $\tau$ .
2. each data object in  $P$  has the same value as its corresponding object in  $P'$  at each time up through time  $\tau$ .
3. If  $R$  is a dynamic use of a variable  $\tau_w B$  in  $P$  at any time  $\tau' \leq \tau$  and  $R'$  is the corresponding dynamic use of  $\tau_w B^{C(\tau_w B)}$  in  $P'$  at the same time  $\tau'$ , and if the value of  $R'$  is not  $\perp$ , then the value of  $R$  is also not  $\perp$ .

$H(\tau)$  is automatically satisfied when  $\tau$  is the time of the start of execution of the program. So let us assume that  $H(s_0)$  is satisfied.

By condition 2, condition 1 continues to be true when  $\tau = s$  (i.e.,  $P$  and  $P'$  are iteratively similar up through time  $s$ ).

Lemma 2.14 then shows that condition 3 remains true when  $\tau = s$  (that is, for every  $\tau' \leq s$ ).

Theorem 2.2 (together with the fact that  $P'$  has the weak dynamic single assignment property up through time  $s_0$ ) then shows that properties  $B$  and  $C$  hold at time  $s$ —that is, that conditions 1 and 2 hold for  $\tau = s$ . (We already know that condition 1 holds at time  $s$  in any case.) Thus  $H(s)$  is satisfied; this is the inductive step.

Now since  $P'$  has the weak dynamic single assignment property for all times before  $t$ , we see therefore by induction that  $H(t)$  is true. Conditions 1 and 2 then state exactly that conditions  $B$  and  $C$  hold up through time  $t$ .  $\square$

**2.16 Lemma** *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.*

PROOF. This follows immediately from the previous Lemma 2.15.  $\square$

**2.17 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  $\tau_w$  variable)  $\mathcal{C}(A)$  is a valid set of contributing dimensions for  $A$ .

If this were not true, then by Lemma 2.16 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

$$A^{\mathcal{C}(A)}(\vec{a}) = \text{expr}_{\text{rhs}}$$

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

Note that none of the dimensions  $i_k$  at which  $s$  and  $t$  differ can be in  $\mathcal{S}(A(\vec{a}))$ . 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  $\text{expr}_{\text{rhs}}$  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  $\mathcal{S}(A(\vec{a}))$ .

On the other hand, by assumption,  $\text{expr}_{\text{rhs}}$  has different values at  $s$  and  $t$ .

Therefore there must be a primary expression  $X$  in  $\text{expr}_{\text{rhs}}$  which has different values at  $s$  and  $t$ . Order the primary expressions of  $\text{expr}_{\text{rhs}}$  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  $\mathcal{S}(A(\vec{a}))$ , 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, since  $P$  is in array SSA form, no such variable can be the target of an assignment statement, so for such a variable  $\mathcal{C}(X) = \mathcal{D}(X)$  and so the rewriting according to  $\mathcal{C}(X)$  leaves  $X$  alone.

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  $\tau_w$  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  $\tau_w$  variable; say the left-hand side of its definition is  $\tau_w B^{\mathcal{C}(\tau_w B)}(\vec{\beta})$ .

So if we have not yet arrived at a contradiction, we have found an array element  $\tau_w B^{\mathcal{C}(\tau_w B)}(\vec{b})$  (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  $\tau_w B^{\mathcal{C}(\tau_w B)}(\vec{b})$  might be  $\perp$ . But it does show that the assignment to  $A^{\mathcal{C}(A)}$  must be a  $\phi$  assignment, since it has a  $\tau_w$  variable on its right-hand side.

Let the definitions of  $\tau_w B^{\mathcal{C}(\tau_w B)}(\vec{b})$  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  $\tau_w B^{\mathcal{C}(\tau_w B)}(\vec{b})$  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  $\tau_w B$  which is an argument to the  $\phi$  function defining  $A$  (this may be *any* such argument, not necessarily the  $\tau_w B$  arrived at above), there is a source assignment to a member of  $web(\tau_w B)$  inside  $L$  that reaches the definition of  $\tau_w B$  by a path in the SSA graph that lies entirely within  $L$ . Say that source assignment is to  $\tau_w B_i$ . Then  $\mathcal{C}(\tau_w B_i)$  includes the dimension corresponding to the loop  $L$ . By Lemma 2.13, this dimension is then propagated by the algorithm to  $\mathcal{C}(\tau_w B)$ . 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 (constraint **C2** on page 37) that there be a  $\tau_w$  variable for each ordinary variable in the  $\phi$  assignment, it follows also that for each ordinary variable  $B$  which is an argument to the  $\phi$  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 constraint **C1** on page 37), 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 condition of Theorem 2.15 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  $A^c$  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, 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**, no. 4, 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).
- 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

- $\perp$ , 3, 14, 43
- admissible program, 36
- admissible transformation, 31
- Array SSA form, 37
  - initial, 19
  - naive, 11
- Boolean choice expressions, 14
- contributing dimensions, 44
  - valid set of, 48
- control  $\phi$ , 12
- control structure, 3
- $\mathcal{D}$ , 4
- data space, 4
- dead code, 30
- definition  $\phi$ , 12
- dynamic single assignment form, 41
  - strong, 41
  - weak, 42
- essential statement, 30
- essential value, 30
- executed global iteration space, 6
- $\mathcal{I}$ , 7
- $[i, \dots]$ , 5, 6
- $[i, \dots]^g$ , 7
- $I(i_j)$ , 7
- iteration space, 4
  - global, 5
  - local, 5
- iterative similarity, 7
- $\vec{k}$ , 2
- $\lambda$  (as subscript), 6
- $\mathcal{M}$ , 45
- maximal expansion, 42
- naive Array SSA form, 13
- ordinary variable, 14
- $\phi$  iteration coordinates, 18
- primary expression, 58
- reference
  - dynamic, 29, 41
  - static, 41
- rewritten program, 47
- $\sigma(d_j)$ , 4
- $\sigma(i_j)$ , 19
- SSA form, 9
- SSA graph, 14, 63
  - collapsed, 14
- SSA variant, 10
- structural similarity, 3
- $t^{\text{loc}}$ , 7
- $\tau_v$ , 7, 12
- $\tau_w$ , 7, 15
- ultimate use, 29
- valid iteration subspace, 45
- web, 15
- wrap-around  $\phi$ , 11