Chapter 8

Single-Assignment Forms

Imperative programming encourages programmers to use and reuse memory locations over and over again to store different values. Clearly, this "history" of values implements the data-flow graph, but not in a clear fashion. One way to expose the data flow is to transform the data structures of the program so that memory locations are less often reused, or perhaps even used only once. This clearly implies that data structures contain more elements.

Transforming a data structure to make it larger, whether conceptually (in an intermediate representation) or in the actual generated code, is called *expansion*. Formally, expansion (we should rather say: expansion of a data structure) means that there are two writes w_1 and w_2 such that, on the one hand,

$$\mathsf{write}(w_1) \cap \mathsf{write}(w_2) \neq \emptyset \tag{8.1}$$

meaning that both writes access overlapping memory locations in the original program, and, on the other hand, these same two writes access different locations in the transformed program:

$$\overline{\mathsf{write}}(w_1) \cap \overline{\mathsf{write}}(w_2) = \emptyset \tag{8.2}$$

Notice that both conditions are required: The first means there was a need for expansion in the first place, and the second means that expansion indeed took place.

8.1 Single-Assignment Form

If expansion is pushed to the extreme, all elements of all data structures are written at most once. The program is then said to have the *single-assignment* property, or to be in single-assignment form. Single-assignment form has been ubiquitous, in various flavors, not only in functional languages but in imperative programming as well. It appears in widespread languages, such as SISAL [12], and, in a restricted form, in a very popular intermediate representation called static single assignment, or SSA, which we study later in this chapter.

8.1.1 The Intuition: Straight-Line Codes

To see the purpose of single assignment, and how it works, it is best to start with an example without branches or calls, that is, a piece of straight-line code. Consider the simple example shown in Figure 8.1.

```
1  if (foo) then
2    x := 0
3  else
4    x := 1
5  end if
6  y := x+1;
7  x := 11
8  if ..  then
9    x := x + 1
10 else
11   x := 3
12 end if
13 z := x
```

(a) Original program

```
1  if (foo) then
2    x2 := 0
3  else
4    x4 := 1
5  end if
6  y6 := ?;
7  x7 := 11
8  if ..  then
9    x9 := ?  + 1
10 else
11   x11 := 3
12 end if
13 z13 := ?
```

(b) Program after renaming left-hand sides

Figure 8.1. Straight-line code being put in single-assignment form.

Converting to single-assignment form is conceptually done in two steps. The first step consists of giving a new and unique name to each variable appearing in all left-hand-side expressions of all statements, as shown in Figure 8.1.(b). That way, each variable is assigned to at most once.[1] Hence the name "single assignment."

However, since variable names in left-hand sides have changed, we have to modify all right-hand expressions accordingly. Each time an x appears in a right-hand expression, we have to decide to which "new" x it now corresponds. For the moment, we have no idea how to do it, and so we just fill in with question marks.

The second step is to replace the question marks. A question mark means we don't know which memory location to read from because it corresponds to a use that had several possible reaching definitions. However, when only one definition reaches a use (the reaching definitions set consists of a nonbottom singleton), then the ambiguity is lifted. In addition, thanks to the first step, each definition has its own memory location. Therefore, in the case of singleton reaching definitions, replacing the question mark by the correct reference is straightforward. As an example, the definitions reaching the use of x in statement **9** is the singleton $\{7\}$. As a consequence, we are sure that reading this memory location provides the right value: This memory location is not polluted by other definitions. The question mark can safely be replaced by the name of this memory location. In the case of statement **9**, x can be replaced by the "x" of statement **7**, that is, x7.

[1] A variable may not be written, depending on the outcome of the conditionals.

```
1  if (foo) then
2    x2 := 0
3  else
4    x4 := 1
5  end if
6  y6 := φ(x2,x4)+1;
7  x7 := 11
8  if ..  then
9    x9 := x7 + 1
10 else
11   x11 := 3
12 end if
13 z13 := φ(x9,x11)
```

Figure 8.2. Complete single-assignment form.

In other cases, solving the "question mark" problem is not so easy, because we have at least two definitions to choose from. If we pick the wrong one, the program transformation is incorrect. Indeed, x in **6** has two possible reaching definitions:

$$\mathrm{RD}(\mathbf{6}) = \{\mathbf{2}, \mathbf{4}\}$$

Similarly, x in statement **13** also has two reaching definitions:

$$\mathrm{RD}(\mathbf{13}) = \{\mathbf{9}, \mathbf{11}\}$$

In either case, we have no direct means to guess which version of x should be read.

We solve the problem by using an oracle function called a ϕ or *ϕ-function*. A ϕ selects which of its arguments is the correct x. Thanks to this oracle, we can say in statement **6** that y6 receives the sum of 1 and the value of whichever variable x2 or x4 is adequate. We also insert a ϕ-function where an x appeared in statement **13** (see Figure 8.2).

It is important to notice here that a ϕ-function is not a function at all in the mathematical sense. Indeed, expression ϕ(x2,x4) returns either the value of x2 *or* the value of x4—no decent function would return two possible results depending on its mood. Clearly, there are additional hidden arguments to ϕ, such as the memory state and guards for each explicit argument. Detailing all these hidden arguments would be tedious, so writing expressions like ϕ(x2,x4) is appealing shorthand.

For the moment, adding ϕ-functions may look like cheating. We just hand-waved their definition, and we did not say a word on how they work, let alone on how to implement them. For the moment, what matters is to realize two key properties of the program in Figure 8.2. First, each memory location defined in the program is never written twice. Second, the flow of data is clear and, in a sense, explicit: From the program in Figure 8.2, we could easily draw a graph like those on the first pages of this book. Input values in x2 and x4 would be passed along edges to an "operator" (not a multiply, like in Figure 1.1, but here a ϕ), creating a new value called y_6 (stored in y6). In addition, the program in Figure 8.2 is a very compact way of representing the flow of values in the program in Figure 8.1(a). In contrast, drawing a similar graph from the program in Figure 8.1(a) would be much more difficult and would require some program understanding that boils down to reaching definition analysis.

Notice also that x2 and x4 do not appear as arguments of the second ϕ in statement **13**. At this point in this book, the profound reason is probably clear: The definitions of x in statements **2** and **4** cannot reach statement **13** (and therefore, statement **13**). The reason is that the write of x in statement **7** kills all previous definitions of x that could otherwise be seen by statement **13**. In turn, the reason **7** kills all previous definitions of x possibly reaching statement **13** is that statement **7** *dominates* statement **13**. But the definition in statement **7** is itself killed by either statement **9** or statement **11**, so x7 is not a possible argument of the ϕ-function.

Left-Hand Sides Let us take one step back and look in retrospect at the example above. In the transformed program, we made sure that all writes access different memory locations. In that particular program, each write corresponds to one statement, but of course, in general, one write corresponds to one statement instance. A formal definition of a single-assignment form is

$$\forall u, v \in \mathcal{W} : (u \neq v) \Rightarrow (\overline{\mathsf{write}}(u) \cap \overline{\mathsf{write}}(v) = \emptyset) \qquad (8.3)$$

which says that two distinct writes do not access overlapping memory locations. Notice that this equation holds for *all* write pairs, whereas (8.2) was meant for a given pair of writes only. In other words, an expansion is not necessarily a conversion to single-assignment form.

Enforcing the single-assignment rule boils down to changing the left-hand expressions of assignments to make sure each assignment instance writes in its own private memory location. We saw that, when the source code is straight-line, instances and statements are equivalent. Therefore, a sufficient condition for this piece of code to be in single assignment is that each left-hand side writes to a separate structure name. This works for all data structures, scalars, arrays, and more complex ones.

When the code includes (structured) loops, the obvious and typical case of loop nests in single-assignment form appears when all statements in the nest body write to a separate array, and all arrays are subscripted by the iteration vector. For instance, the loop nest below is in single-assignment form:

```
1  for i := 1 to n do
2    for j := 1 to n do
3      a[i,j] := ..
4    end for
5  end for
```

Indeed, (8.3) holds because there are no pairs of instances of **3** that write to the same memory location:

$$\forall \mathbf{3}^{i,j}, \mathbf{3}^{i',j'} \in \mathcal{D}(\mathbf{3}) : \left(\mathbf{3}^{i,j} \neq \mathbf{3}^{i',j'}\right) \Rightarrow \left(\mathsf{write}(\mathbf{3}^{i,j}) \cap \mathsf{write}(\mathbf{3}^{i',j'}) = \emptyset\right)$$

which is even more obvious in the following equivalent expression:

$$\forall i, j, i', j' \in \mathbb{N} : ((i \neq i') \vee (j \neq j')) \Longrightarrow ((i, j) \neq (i', j'))$$

However, remember that the above gave only a *sufficient* condition. Data structures in a loop nest do not need to be indexed by the counters of surrounding loops for the nest to be in single-assignment form. Figure 8.3 shows an excerpt of the SPEC benchmark 176.gcc that illustrates this. Variables offset, i, bit, and sometimes_max serve as counters. The relevant modified data structures are arrays reg_basic_block and regs_sometimes_live and, indeed, we can check that two iterations of the body never write into the same array element. Indeed, variable sometimes_max is incremented each time regs_sometimes_live is accessed, making sure the next access will refer to another element of the array.

```
for (offset = 0, i = 0; offset < regset_size; offset++)
 for (bit = 1; bit; bit <<= 1, i++)
 {
  if (i == max_regno)
    break;
  if (old[offset] & bit)
  {
    reg_basic_block[i] = REG_BLOCK_GLOBAL;
    regs_sometimes_live[sometimes_max].offset = offset;
    regs_sometimes_live[sometimes_max].bit = i % REGSET;
    sometimes_max++;
  }
 }
```

Figure 8.3. Excerpt from 176.gcc.

The reason reg_basic_block is in single-assignment form is more subtle. Variable i is initialized in the initialization phase of the outer loop and is incremented at each iteration of the inner loop. Therefore, the index of reg_basic_block never has the same value twice while executing the loop nest.

Right-Hand Sides Situations where a ϕ-function is needed can be defined formally without the help (or should we say the additional complexity?) of the control-flow graph. A ϕ-function must be substituted to a reference m in the right-hand side of a statement **S** if there is an instance r of **S** that has more than one instancewise reaching definition:

$$\exists u, v \in \mathcal{W},\ u \neq v \wedge u \in \mathsf{RD}(\langle r, \mathtt{m}\rangle) \wedge v \in \mathsf{RD}(\langle r, \mathtt{m}\rangle) \qquad (8.4)$$

In such cases, the arguments of the ϕ-function are exactly the memory locations written by the us and vs satisfying (8.4). Formally, if (8.4) holds, then m should be replaced by

$$\phi(\overline{\mathsf{write}}(\mathsf{RD}(\langle r, \mathtt{x}\rangle))) \tag{8.5}$$

For instance, in Figure 8.1, the definitions reaching statement **6** are $\mathsf{RD}(\mathbf{6}) = \{2, 4\}$, so the reference to x in the right-hand expression of **6** should be replaced by

$$\phi(\overline{\mathsf{write}}(\overline{\mathsf{RD}}(\mathbf{6}))) = \phi(\overline{\mathsf{write}}(\{2,4\})) = \phi(\mathtt{x2}, \mathtt{x4}) \tag{8.6}$$

It is important to notice that conditionals commute with

$$\phi \circ \overline{\mathsf{write}}$$

In other words, **if-then-else** expressions that may appear in $\mathsf{RD}(\langle r, \mathtt{x}\rangle)$ in (8.5) can be pulled out of expression $\phi(\overline{\mathsf{write}}(\mathsf{RD}(\langle r, \mathtt{x}\rangle)))$. For instance, another valid expression for the definitions reaching x in Figure 8.1 could have been

$$\mathsf{RD}(\langle r, \mathtt{x}\rangle) = \textbf{if } foo \textbf{ then } \{2\} \textbf{ else } \{4\}$$

The occurrence of x in the right-hand expression of **6** could then be replaced by

$$\begin{aligned} \phi(\overline{\mathsf{write}}(\overline{\mathsf{RD}}(\mathbf{6}))) &= \phi(\overline{\mathsf{write}}(\textbf{if } foo \textbf{ then } \{2\} \textbf{ else } \{4\})) \\ &= \textbf{if } foo \textbf{ then } \phi(\overline{\mathsf{write}}(\{2\})) \textbf{ else } \phi(\overline{\mathsf{write}}(\{4\})) \\ &= \textbf{if } foo \textbf{ then } \mathtt{x2} \textbf{ else } \mathtt{x4} \end{aligned} \tag{8.7}$$

When no compile-time information of foo is available, (8.7) is just another way of saying (8.6). More precisely, it provides a possible *implementation* of (8.6) since it tells what ϕ should test to deliver the right value: foo in (8.7) gates the use of x2 and x4. This idea is the key to *gated SSA* [86]. Moreover, when the conditional depends on the iteration vector and its leaves are singleton sets, ϕ can be eliminated completely. We come back to this point in a few moments.

8.1.2 The Case of Loops

```
0  x := 0
1  for i := 1 to 10 do
2    x := x + i
3  end for
```

Figure 8.4. Simple loop with a recurrence.

Consider Figure 2.5, shown again here in Figure 8.4. The first step of conversion to single-assignment form consists, as before, of giving one private memory location


```
0  x0 := 0
1  for i := 1 to 10 do
2    x2[i] := ?  + i
3  end for
```

Figure 8.5. First step in converting Figure 8.4 to single-assignment form.

to each individual write—in this example, to each of the 10 individual instances of statement **2**.

The solution is probably obvious: Give statement **2** its own entire array, and give each instance of **2** an element of the newly defined array. In other words, construct a data structure that has the same shape as (which is isomorphic to) the domain of statement **2**. The result of step 1 is shown in Figure 8.5. Array x2 is statement **2**'s own private array, and we suppose it is declared as an array of 10 elements.

```
0  x0 := 0
1  for i := 1 to 10 do
2    x2[i] := φ(write(RD(2^i))) + i
3  end for
```

Figure 8.6. Second step in converting Figure 8.4 to single-assignment form.

As before, the second step consists of replacing all occurrences of x in right-hand sides. A memory reference like x is replaced by a ϕ-function if there are multiple reaching definitions. This replacement is done according to (8.5). The resulting program appears in Figure 8.6.

We can go one step further. We saw in (5.1) page 79 that the definitions reaching x in **2** are

$$\mathsf{RD}(\mathbf{2}^i) = \begin{array}{l} \textbf{if } i > 1 \\ \textbf{then } \{\mathbf{2}^{i-1}\} \\ \textbf{else } \{\mathbf{0}\} \end{array} \tag{8.8}$$

We now commute the conditional in (8.8) and $\phi \circ \overline{\mathsf{write}}$ in **2** in Figure 8.6, yielding the program shown in Figure 8.7. The whole process boils down to plugging in the conditional expression returned by the instancewise reaching definition analysis.

You may argue that the `if` construct in Figure 8.7 *is* a ϕ-function. This is wrong, however, because both parts of the `if` contain a single array element. ϕ-functions serve as syntactic sugar for a lack of compile-time knowledge on the flow of data. They are fancy forms of "don't-know" answers, any argument to a ϕ-function being a valid candidate. In contrast, the `if` construct in Figure 8.7 makes explicit the exact compile-time knowledge on the pattern of the data flow. This can be used in turn in

```
0  x0 := 0
1  for i := 1 to 10 do
2    x2[i] := (if i=1 then x0 else x2[i-1]) + i
3  end for
```

Figure 8.7. Simple loop with a recurrence, in single-assignment form.

additional optimizations, like loop peeling. As shown in Figure 8.8, the first iteration is removed from the iteration domain of statement **2**, so the initial value of `i` becomes 2. The corresponding instance $\mathbf{2}^1$ of statement **2** is inserted back in as a full statement line.

```
0   x0 := 0
2^1 x2[1] := x0 + 1
1   for i := 2 to 10 do
2     x2[i] := x2[i-1] + i
3   end for
```

Figure 8.8. Figure 8.7 after loop peeling.

The point we would like to make is that there is no way the loop-peeled version could be obtained from Figure 8.6. Contrasting Figures 8.6 and 8.7 shows that there is an intrinsic difference between a ϕ-function and a reaching definition expression with singleton leaves.

Let us also stress that the single-assignment form in Figure 8.7 is interesting not only as a sketch for the code actually generated, but also as an intermediate program representation. It provides, indeed, most of the information we know on the program's data flow, including the actual behavior of ϕ-functions.

This process is very general and extends to more complex programs. Consider the code for Choleski factorization studied in Exercise 5.5 and reproduced here in Figure 8.9. Applying the first step of single-assignment conversion yields left-hand sides `x2[i]`, `x4[i,k]`, `p6[i]`, `x8[i,j]`, `x10[i,j,k]`, and `a12[i,j]` for statements **2**, **4**, **6**, **8**, **10**, and **12**, respectively. Notice that the left-hand side of statement **6** is just renamed. The modification for statement **12** is both subtle and brute-force: Strictly obeying the rules for the first step, the array in the left-hand side must be indexed by the iteration vector, which is (i,j), not (j,i). This leads to subscript `[i,j]` in the left-hand side, in contrast with `[j,i]` in the original program.

We now apply step 2 of conversion to single-assignment form. In particular, statement **4** is transformed into

$$\texttt{x4[i,k]} := (\overline{\mathsf{write}}(\mathsf{RD}(\langle \mathbf{4}^{i,k}, \texttt{x}\rangle))) - (\overline{\mathsf{write}}(\mathsf{RD}(\langle \mathbf{4}^{i,k}, \texttt{a[i,k]}\rangle)))\texttt{\^{}2;}$$

```
1  for i :=1 to n do
2    x := a[i,i];
3    for k := 1 to i-1 do
4      x:= x - a[i,k]^2;
5    end for
6    p := 1.0/sqrt( x );
7    for j := i+1 to n do
8      x := a[i,j];
9      for k := 1 to i-1 do
10       x := x - a[j,k] * a[i,k] ;
11     end for
12     a[j,i] := x * p[i];
13   end for
14 end for
```

Figure 8.9. Choleski factorization.

```
1  for i := 1 to n do
2    x2[i] := a[i,i];
3    for k := 1 to i-1 do
4      x4[i,k]:=(if k>=2 then x4[i,k-1] else x2[i] endif)
              - a12[k,i]^2;
5    end for
6    p6[i]:=1.0/sqrt(if i>=2 then x4[i,i-1] else x2[i]);
7    for j := i+1 to n do
8      x8[i,j] := a[i,j];
9      for k := 1 to i-1 do
10       x10[i,j,k]:=
           (if k>=2 then x10[i,j,k-1] else x8[i,j] endif)
           - a12[k,j] * a12[k,i];
11     end for
12     a12[i,j] :=
          (if i>=2 then x10[i,j,i-1] else x8[i,j] endif)
          * p6[i];
13   end for
14 end for
```

Figure 8.10. Single-assignment form for Choleski.

We can leverage the result of reaching definition analysis provided in Eq. (5.10) through (5.16) page 97. In particular, Eq. (5.10) and (5.14) are:

$$\mathsf{RD}(\langle \mathbf{4}^{i,k}, \mathtt{x}\rangle) \;=\; \begin{array}{l}\textbf{if } k \geq 2\\ \textbf{then } \{\mathbf{4}^{i,k-1}\}\\ \textbf{else } \{\mathbf{2}^{i}\}\end{array} \tag{8.9}$$

$$\mathsf{RD}(\langle \mathbf{4}^{i,k}, \mathtt{a[i,k]}\rangle) \;=\; \{\mathbf{12}^{k,i}\} \tag{8.10}$$

Plugging (8.9) and (8.10) in statement **4**, and using the commutativity property, we get

$$\mathtt{x4[i,k]:=\ (if\ k>=2\ then\ } \overline{\mathsf{write}}(\mathbf{4}^{i,k-1}) \mathtt{\ else\ } \overline{\mathsf{write}}(\mathbf{2}^{i}) \mathtt{\ endif)}$$
$$\mathtt{-\ } \overline{\mathsf{write}}(\mathbf{12}^{k,i})\mathtt{\hat{}2;}$$

that is,

```
x4[i,k]:= (if k>=2 then x4[i,k-1] else x2[i] endif)
          - a12[k,i]^2;
```

Indeed, thanks to step 1, there is a one-to-one mapping between definition instances and memory locations. In addition, this mapping is just syntactical. Therefore, accesses to x and a in right-hand sides are just replaced by references to the array of the defining statement. Subscripts are given by the iteration vector of the defining instance.

The final program is shown in Figure 8.10. Observe that the right-hand side of statement **8** is not a typo. Page 98 shows that there are no definitions in this kernel reaching reference a[i,j]. We therefore consider that some previous statement outside the kernel provides the input values for a.

The code in Figure 8.10 was produced automatically (in a syntactically different but semantically equivalent form) by the PAF compiler developed by Prof. Feautrier and his team at the University of Versailles.

Other Data Structures Appropriate for Single Assignment Consider again the program in Figure 8.4. It is simple enough to tell us array x2 needs exactly 10 elements. When the lower or upper bound is not so clear, or when the loop is a while, single assignment stays conceptually identical.

There are cases, however, where single assignment cannot serve "as is" to generate the actual code. Consider the program in Figure 5.20 page 103, shown again in Figure 8.11. The while .. do construct is just a counted while loop and is introduced page 12.

Each instance $\mathbf{3}^{i}$ assigns variable x. In turn, statement **5** assigns x an undefined number of times (possibly zero). The value read in x by statement **7** is thus defined *either* by **3** *or* by some instance of **5** in *the same iteration* of the for loop (the same i). Therefore, if the expansion assigns distinct memory locations to $\mathbf{3}^{i}$ and to instances of $\mathbf{5}^{i,w}$, how can instance $\mathbf{7}^{i}$ "know" which memory location to read from?

As before, to solve this problem, we use the result of an instancewise reaching definition analysis. This result is given in (5.22) and (5.23) page 104:

$$(5.22) \Leftrightarrow \mathsf{RD}(\mathbf{5}^{i,w}) = \begin{array}{l}\textbf{if } w > 1\\ \textbf{then } \{\mathbf{5}^{i,w-1}\}\\ \textbf{else } \{\mathbf{3}^{i}\}\end{array}$$

?

```
1  real x, A[N]
2  for i := 1 to N do
3    x := foo(i)
4    while (..)  do w := 1 begin
5      x := bar(x)
6    end while
7    A[i] := x
8  end for
```

Figure 8.11. Program with a while *loop introduced in Exercise 5.7.*

and

$$(5.23) \Leftrightarrow \mathsf{RD}(\mathbf{7}^{i}) = \{\mathbf{3}^{i}\} \cup \{\mathbf{5}^{i,w} : w \geq 1\}$$

Now let us convert the program to single-assignment form, making **3** write into x3[i] and **5** into x5[i,w]. Then each memory location is assigned to at most once, complying with the definition of single assignment. To transform right-hand sides, we use (5.22) and (5.23). This yields the program in Figure 8.12. The right-hand side of **5** depends only on w. The right-hand side of **7** depends on the control flow, thus needing a ϕ-function.

```
2  for i := 1 to N do
3    x3[i] := foo(i)
4    while (..)  do w := 1 begin
5      x5[i,w] := if w>1 then bar(x5[i,w-1])
                         else bar(x3[i])
                  end if
6    end while
7    A7[i] := φ({x3[i]} ∪ {x5[i,w] : w ≥ 1})
8  end for
```

Figure 8.12. Program in Figure 8.11 put in single-assignment form.

The program in Figure 8.12 has the property that any reference to, say, x5[3,2] refers to the same value. However, that program cannot be compiled as is, because we have no upper bound on the second dimension of x5. (We face the same issue when addressing recursive programs in a few moments.)

Another solution, therefore, is to use data structures that can be extended dynamically. Indeed, to implement single-assignment form, these structures have to manage as many elements as the instance count. In addition, these data structures should have the same shape as the iteration domains to simplify the mapping of instances to data elements. (We say the domain of a statement and its new associated data structure are isomorphic.) In this example, each instance 5^w of **5** pushes the produced value on a stack. (A list would do the trick as well.) The assignment in **5** then has the following form:

```
mystack := push(mystack, newvalue);
```

In Section 1.8, we agree on the convention that the counters of "counted" `while` loops are set to 0 if the loop does not iterate at all. The ϕ-function therefore has a natural implementation: If w equals 0 at statement **7**, then the value defined by **3** can be used. Otherwise, statement **7** simply has to pop the value from the appropriate stack. The transformed program can be summarized as shown in Figure 8.13.

```
1  Declare N stacks x5[1]..x5[N];
2  for i := 1 to N do
3    x3[i] := foo(i)
4    while (..)  do w := 1 begin
5      x5[i] := push(x5[i],
                  (if w>1 then bar(pop(x5[i]))
                          else bar(x3[i])
                   end if));
6    end while
7    A7[i] := if w=0 then x3[i] else pop(x5[i]);
8  end for
```

Figure 8.13. Single assignment for Figure 8.11, using stacks.

Notice that the use of dynamic data structures can be generalized: For example, in theory two nested `while` loops can be converted to single assignment using a list of lists. We see later that single assignment for a binary (or n-ary, with $n > 1$) recursive procedure can be supported by a tree of appropriate degree.

8.1.3 Recursive Procedures

Conversion of recursive programs to single-assignment form has been much less studied in the literature. However, the conversion guidelines given above still apply.

For one, the new data structure a recursive procedure writes into has the same shape as the procedure's domain. For instance, if the procedure is doubly recursive, then the a tree of degree 2 is enough to allow for single assignment — whatever the original data structure was. Left-hand sides are then simply a mapping from the instruction's control word to the tree. As an example, let us consider the program of Figure 3.12, shown here again in Figure 8.14.

Because it is doubly recursive, its call graph is a tree, as discussed in Chapter 3, whose nodes correspond to instances of the procedure and can be labeled by words in


```
1  procedure P
2  do
3    v := foo(v);
4    if c1 then P() endif;
5    if c2 then P() endif
6  end
```

Figure 8.14. A simple recursive procedure.

the regular language $(\mathbf{4}+\mathbf{5})^*$. In the procedure shown in Figure 8.14, each procedure invocation writes into `v`, so converting into single-assignment form requires replacing scalar `v` by a tree-shaped data structure—a data structure isomorphic to the call tree. This tree will be called `v3`, to stress the fact it corresponds to statement **3**. (There might be more than one tree corresponding to scalar `v`, as several arrays `x1`, `x2`, etc. correspond to scalar `x` in Figure 8.4.) As discussed earlier, changing left-hand expressions is the first step in converting to single-assignment form. The result of this first step is shown in Figure 8.15, where the notation `[..]` is used to index tree-like structures *à la* array. Because statement **4** goes along left children and statement **5** toward right children, a node in `v3` is denoted by `v3[`w`]`, where $w \in (\mathbf{4}+\mathbf{5})^*$. In other words, the left child of a node w of `v3` is referenced by `v3[`w`.4]` and the right child by `v3[`w`.5]`.

```
0  Type "word" captures the regular language (4+5)*
1  procedure P(word w)
2  do
3    v3[w] := foo(...);
4    if c1 then P(w.4) endif;
5    if c2 then P(w.5) endif
6  end
```

Figure 8.15. First step toward a conversion to single-assignment form.

In the second step, we replace references to `v` by ϕ-functions each time there might be more than one reaching definition, as illustrated in Figure 8.16. The program in Figure 8.16 can be compared with the one in Figure 8.6.

Furthermore, applying the result of the reaching definition analysis done in Chapter 5 gives more details. We saw that one way to express the result of this analysis is given by the following three equations:

$$\mathsf{RD}(3) = \{\bot\} \tag{8.11}$$

```
1  procedure P(w)
2  do
3    v3[w] := foo(φ(write(RD(3^w))));
4    if c1 then P(w.4) endif;
5    if c2 then P(w.5) endif
6  end
```

Figure 8.16. Single-assignment form for the procedure of Figure 8.14.

$$\forall w \in \mathcal{D}(\mathbf{3}), w' \in (\mathbf{4}+\mathbf{5})^* : (w = w'\mathbf{43}) \Rightarrow (\mathsf{RD}(w) = \{w'\mathbf{3}\}) \tag{8.12}$$

$$\forall w \in \mathcal{D}(\mathbf{3}), w' \in (\mathbf{4}+\mathbf{5})^* : (w = w'\mathbf{53}) \Rightarrow (\mathsf{RD}(w) = w'(\mathbf{4}+\varepsilon)(\mathbf{4}+\mathbf{5})^*\mathbf{3}) \tag{8.13}$$

Equation (8.11) gives the definition reaching the root instance in the call tree, (8.12) provides the exact definition reaching a given left child, whereas (8.13) is not as definite since it maps right children to sets (regular languages) of writes.

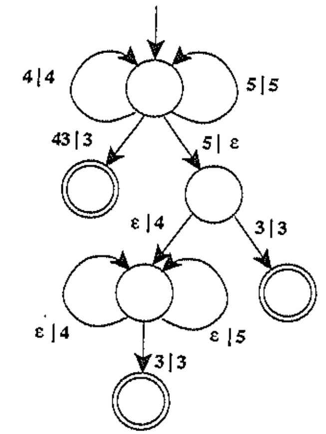

Figure 8.17. Reaching definition transducer for the program in Figure 8.14.

The equivalent transducer appears in Figure 5.28, reproduced here in Figure 8.17. Taking the image by this relation of the current instance of statement **3** gives the set of all possible definitions reaching that instance. Applying mapping write to that set gives the labels of all elements of tree v3 that possibly contain the value we need. The abstract transformed program is shown in Figure 8.18. (In Figure 8.18, we drop the final "**3**" in words w.) The `matches` operator is a pattern matching operator, as in the PERL language. That is, the following expression

`w matches w'4`

if true, means that w ends with **4** and that w' is the corresponding prefix.

```
1  procedure P(w)
2  do
3    v3[w] :=
        foo( (if w=ε then v3[ε]
                else (if w matches w'4 then v3[w']
                      else φ(write(RD(3^w))) )));
4    if c1 then P(w.4) endif;
5    if c2 then P(w.5) endif
6  end
```

Figure 8.18. Refinement of Figure 8.16.

You can notice that the case-by-case expression in Figure 8.18 represents a refinement of the program shown in Figure 8.16. In the first two cases, there is a direct mapping to the appropriate element of tree v3. However, the expression for single-assignment form cannot be derived beyond this point. Indeed, there is no simple expression for the ϕ in Figure 8.16 because there is no simple closed-form expression for the relation captured by the transducer in Figure 5.28. A dedicated run-time support is required to restore the data flow in the final case of statement **3**.

8.1.4 Array Single Assignment

So far, all data structures in input programs have been scalars. However, single assignment applies as well to more general structures, and to arrays in particular. We then call it *array single assignment*.

Array single assignment has exactly the same purpose as single assignment: It makes the data flow explicit in imperative programs, extending this idea to programs manipulating arrays. Thanks to this property, array single assignment has been used for a long time in many fields, including parallelizing compilers and VLSI array design [59].

To see how array single assignment helps transform programs, consider the example in Figure 8.19. Its single-assignment counterpart is shown in Figure 8.20. As you can notice, by definition of single assignment, all anti- and output dependencies are removed. The benefit of this property is that any reference to, say, `a4[3,2]` anywhere in the program refers not only to the same memory location but also to the same value. The second benefit for parallelizing compilers is of course that fewer dependencies means more parallelism. A parallelizing compiler might then want to use the array SA form as a draft for the code actually generated.

```
a[..]  := ...
for i := 1 to n do
  for j := 1 to n do
    a[i+j] := 1 + a[i+j-1]
  end for
end for
```

Figure 8.19. Single assignment extended to arrays.

```
a1[..]  := ...
for i := 1 to n do
  for j := 1 to n do
    a4[i,j] := 1 +
      (if j>=2 then a4[i,j-1]
       else (if i>=2 then
        a4[i-1,j] else a1[i+j-1] ))
  end for
end for
```

Figure 8.20. Program in Figure 8.19 in single-assignment form.

Another point is that it really is difficult to visualize how data flow in the course of an execution. In contrast, Figure 8.20 shows how regular the flow of data actually is. The regularity of the data flow of the program in Figure 8.19 appears even more clearly in Figure 8.21. For instance, the graph makes clear that, for j greater than 1, the source of the data flow is the immediate neighbor down the vertical axis. Notice that the data flow in the program in Figure 8.19 is, in fact, identical. However, its data flow is lost among many memory-based dependencies that are not relevant to the actual algorithm. In contrast, the simplicity of the algorithm of the program in Figure 8.19 shows up very well in Figure 8.20.

What Is New Here? Notice that we stress "array" in the expression "array single assignment" for pedagogical reasons only. By emphasizing "array," we underline the difference between the material of this section and what can be found elsewhere in the literature. But there is in fact nothing new, and we should not even need to mention the word "array": Single assignment is a general concept not limited to scalars or arrays. Indeed, it is important to realize that, whatever the control-flow structures or the data structures, conversion to single-assignment form can be automated as long as an adequate instancewise reaching definition analysis is available. Feautrier was the first researcher to realize this and to present an algorithm for single-assignment conversion in the presence of nonscalar data structures [29]. The instancewise reaching definition analysis presented in this book gives a means to transform a program, even if it includes

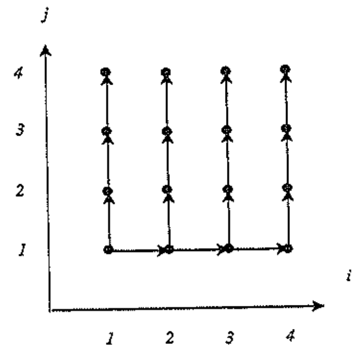

Figure 8.21. Flow of data in the programs in Figures 8.19 and 8.20.

arrays, to single-assignment form.

In fact, the alert reader may have noticed that, in all the material of this chapter, the original data structure is *not* relevant in converting to single-assignment form. What matters is the *domain* of the statement. More precisely, whatever the data structure in the original programs, single-assignment form in the transformed program *only* requires the new data structure to be isomorphic to (i.e., in one-to-one mapping with) the set of write instances, that is, the domain of the assignment.

```
a[..]  := ...
for i := 1 to n do
  if P(i) then
    for j := 1 to n do
      a[i+j] := 1 + a[i+j-1]
    end for
  end if
end for
```

Figure 8.22. Program in Figure 8.19 with additional test.

How General Is Conversion to Array Single-Assignment Form? As stated earlier, conversion to single-assignment form (or the "array" version, for that matter) only requires us (1) to be able to create data structures that are isomorphic to the control flow and (2) to have an instancewise reaching definition. If some control structures have no bound, like `while` loops in general, then requisite (1) is an issue and some tricks need to be applied, when possible. If control structures include conditionals, then the difficulty lies more in (2), that is, the reaching definition analysis. However, we have seen in previous chapters that this is not a difficult issue.

Consider the example shown in Figure 8.22, which adds a conditional to the program in Figure 8.19. For illustration purposes, we assume the first statement initializes all elements of array a in that program. A reaching definition analysis is done in Exercise 5.9 page 108. This analysis reports that the definitions reaching $5^{i,j}$ are

$$\mathrm{RD}(5^{i,j}) = \begin{array}{l} \textbf{if } j \geq 2 \\ \textbf{then } \{5^{i,j-1}\} \\ \textbf{else } \ \textbf{if } i \geq 2 \\ \qquad \textbf{then } \{1\} \cup \{5^{i',j'} : 1 \leq i' < i, 1 \leq j' \leq n, i'+j' = i+j-1\} \\ \qquad \textbf{else } \{1\} \end{array}$$

Again, instancewise reaching definitions give nearly all the information we need: In two cases, there is one and only one definition reaching the read of `a[i+j-1]`. These two cases are, on the one hand, all iterations of the inner loop except the first one (j greater than or equal to 2) and, on the other hand, the very first execution of statement 5. Therefore, no ϕ-function is needed in either case. However, when $i > 1$ and $j = 1$, many definitions may reach `a[i+j-1]`, and the right one cannot be selected at compile time. A ϕ-function is then (and only then) needed to abstract (or actually implement!) this selection. The single-assignment form of Figure 8.22 is shown in Figure 8.23.

```
1  a1[..]  := ...
2  for i := 1 to n do
3    if P(i) then
4      for j := 1 to n do
5        a5[i,j] := 1 +
           if j>=2 then a5[i,j-1]
           else
             if i>=2 then
               φ({a1[i+j-1]}∪
                 {a5[i',j'] : 1 ≤ i' < i, 1 ≤ j' ≤ n, i'+j' = i+j-1})
             else
               a1[i+j-1]
             end if
           end if
6      end for
7    end if
8  end for
```

Figure 8.23. Single-assignment form of the program in Figure 8.22.

Exercise 8.1 Convert the program shown in Figure 8.24 to single assignment form. (We assume $1 \leq foo \leq n+1$.) Hint: The reaching definitions in this program are studied in Exercise 5.8.

```
1  for i = 1 to n do
2    a[i] := ...;
3    if(..)  a[i+1]:=...  end if
4  end for
5  a[n+1] := ..
6  ..  := a[foo]
```

Figure 8.24. Program used in Exercise 8.1.

Solution The reaching definitions given by (5.24) page 108 are:

$$\mathrm{RD}(\mathbf{6}) = \begin{array}{l} \textbf{if } foo = n+1 \\ \textbf{then } \{5\} \\ \textbf{else } \{2^{foo}\} \end{array}$$

This expression directly yields the transformed program shown in Figure 8.25. Notice that `a3` does not appear in the right-hand expression of **6** thanks to the result of reaching definition analysis.

```
1  for i = 1 to n do
2    a2[i] := ...;
3    if(..)  a3[i+1]:=...  end if
4  end for
5  a5[n+1] = ..;
6  ..  := if (foo=n+1) then a5[n+1] else a2[foo] end if
```

Figure 8.25. Single-assignment form for the program in Figure 8.24.

Notice that, for the purpose of conversion to single-assignment form, reaching definition analysis should not be too aggressive in eliminating execution-time constants. For instance, since the analysis knows little about the value of `foo` (just that $1 \leq foo \leq n+1$), it might be tempted to collapse the conditional, yielding

$$\mathrm{RD}(\mathbf{6}) = \{2^{foo}\} \cup \{5\} \quad (8.14)$$

The corresponding transformed program would then be the one in Figure 8.26. ■

```
1  for i := 1 to n do
2    a2[i] := ...
3    if(..) a3[i+1] := ...
4  end for
5  a5[n+1] := ..
6  .. := φ(a2[foo],a5[n+1])
```

Figure 8.26. Single-assignment form for the program in Figure 8.24.

Exercise 8.2 Convert the program skeleton shown in Figure 5.15, shown here again in Figure 8.27, to single-assignment form.

```
1  a[1] := 0
2  for i := 1 to n do
3    for j := 1 to n
4      a[i+j] := ...
5      a[i] := ...  a[i+j-1]
6    end for
7  end for
```

Figure 8.27. Program used in Exercise 8.2.

```
1  a1[1] := 0
2  for i := 1 to n
3    for j := 1 to n
4      a4[i,j] := ...
5      a5[i,j]:=.. if j>=2 then a4[i,j-1]
                   else  if i>=2 then a4[i-1,j]
                         else a1[1]
                         end if
                   end if
6    end for
7  end for
```

Figure 8.28. Single-assignment form for the program shown in Figure 8.27.

Solution We see in (5.9) that the definitions reaching `a[i+j-1]` are

$$\mathsf{RD}(5^{i,j}) = \begin{array}{l}\textbf{if } j \geq 2 \\ \textbf{then } \{4^{i,j-1}\} \\ \textbf{else } \textbf{if } i \geq 2 \\ \quad\quad \textbf{then } \{4^{i-1,j}\} \\ \quad\quad \textbf{else } \{1\}\end{array}$$

This tells us that definitions in the left-hand side of statement **5** never reach the right-hand side. Therefore, private array `a5` of **5** does not appear as an argument to the ϕ-function. An expression for the ϕ-function that depends only on loop counters can also be derived from (5.9). The end result appears in Figure 8.28. ■

Exercise 8.3 Consider the program below:

```
0  x[0] := 0
1  for i:= 1 to 2*n do
2    t := x[2*n-i+1];
3    x[i] := x[i-1] + t
4  end for
```

Convert it to single-assignment form. Hint: We compute the definition reaching `x` in statement **2**, page 92.

Solution On page 92, we see that the definitions of `x` reaching 2^i are

$$\mathsf{RD}(\langle 2^i, \texttt{x[2*n-i+1]}\rangle) = \begin{array}{l}\textbf{if } 1 \leq i \leq n \\ \textbf{then } \{\perp\} \\ \textbf{else } \{3^{2n-i+1}\}\end{array}$$

The definitions reaching `x[i-1]` and `t` in statement **3** are easier to find:

$$\mathsf{RD}(\langle 3^i, \texttt{x[i-1]}\rangle) = \begin{array}{l}\textbf{if } i = 1 \\ \textbf{then } \{0\} \\ \textbf{else } \{3^{i-1}\}\end{array}$$

and

$$\mathsf{RD}(\langle 3^i, \texttt{t}\rangle) = \{2^i\}$$

A single-assignment equivalent of this program is, therefore,

```
0  x0[0] := 0
1  for i:= 1 to 2*n do
2    t2[i]:= if 1<=i<=n then x[2*n-i+1]
             else x3[2*n-i+1] end if;
3    x3[i]:= (if i=1 then x0[0] else x3[i-1] end if)
             + t2[i]
4  end for
```

Notice that the "bottom" $\perp$ in $\mathsf{RD}(\langle 2^i, x\rangle)$ means that the corresponding reference is not defined in the given program fragment. Therefore, the transformation should plug in the original reference, verbatim, for this particular case. This is why the right-hand expression in statement **2** begins with

```
if 1<=i<=n then x[2*n-i+1]
```

even though array x is not defined anywhere in the transformed program. Indeed, we should not forget that more statements, including definitions to array x, may precede statement **0**. This single-assignment program is used in Chapter 11. ∎

A Few More Tricks So far, conversion to single-assignment form can be done in an automatic or semiautomatic way. However, there are cases where automatic conversion fails. Alternatively, when the reaching definition analysis returns a very approximate result, the transformed program may be too complex to be useful. In these cases, conversion by hand is a very useful option. Indeed, some applications do not absolutely require an automatic tool to transform a program. A case in point is systolic array design [59] for which designers are willing to spend hours on the specification (the program to be mapped to a systolic array) if this can improve the area, the throughput, or the power consumption of the final circuit. (The next section is dedicated to systolic array design.)

Transforming a program by hand also allows us to play more tricks. Consider the simple loop below:

```
1  for i := 1 to n do
2    if p(i) then
3      x := x+1
4    end if
5  end for
```

Converting the loop to single-assignment form using the method described earlier in this chapter would consist of two steps. First, find the instancewise reaching definitions of x in **3**:

$$\mathsf{RD}(3^i) = \begin{array}{l}\textbf{if } i = 1 \\ \textbf{then } \{\perp\} \\ \textbf{else } \{3^{i'} : 1 \le i' < i\}\end{array}$$

Second, expand left-hand expressions and plug the reaching definitions in right-hand sides:

```
for i := 1 to n do
  if p(i) then
    x[i] := (if i=1 then x else φ({x[i'] : 1 ≤ i' < i}))
            + 1
  end if
end for
```

which is correct but quite intricate. In particular, we lost the fact that the value of x in the right-hand side comes was produced by the last instance of **3**.

On the other hand, the initial program can equivalently be seen as

```
1  for i := 1 to n do
2    if p(i) then
3      x := x+1
4    else
5      x := x
6    end if
7  end for
```

Of course, the new assignment in the "else" branch of the conditional is just a copy that serves to propagate the value of x from one iteration to the next. Data such as x, which are propagated without being modified, are called *transmittent data* [59].

The big asset of the modified loop is that the definition reaching x in statement **3** at a given iteration i, for $i > 1$, comes from one of *exactly* two definitions: 3^{i-1} or 5^{i-1}. Then, converting the loop to single-assignment form is pretty simple:

```
for i := 1 to n do
  if p(i) then
    x[i] := x[i-1]+1
  else
    x[i] := x[i-1]
  end if
end for
```

Notice that, in the latter loop, we did not duplicate array x into two different arrays x3 and x5. In other words, we did not change the left-hand expressions of **3** and **5** into x3[i] and x5[i], respectively. This allows us to keep simple right-hand sides. We revisit this idea in Chapter 9.

8.1.5 Single Assignment and Systolic Array Design

For instance, single assignment is of the utmost importance in *systolic array* design. In that context, transforming the input program or algorithm in any way—by a tool or by the designer—can make sense because the time available to the design of a circuit, including compilation, is much longer than the time considered to be acceptable in the classical compilation context (where the user is staring at the screen waiting for his or her compilation to finish). In circuit design, the user has the freedom to change the input program to make the systolic implementation faster.

In particular, it would make no sense for a systolic array designer to keep the output dependencies of the original algorithm or program. These dependencies would only reduce the amount of parallelism in the circuit. Second, systolic arrays need local, regular data flow.

The best way to exhibit this locality and this regularity is to put the program into single assignment [59]. Pieces of the program can then be considered as functional building blocks that can be composed at will as very well illustrated in [1]. These pieces become circuit parts, more often the cells of the systolic array, and can be laid out more easily on the circuit.

```
1  Lmax := 0;
2  Iopt := 0;
3  for i := 0 to n-Ls-1 do
4    L[i] := 0;
5    E[i] := 1;
6    for j := 0 to Ls-1 do
7      if(M[j]=B[i+j] and E[i]=1) then
8        L[i] := L[i]+1
9      else
10       E[i] := 0
11     end if
12   end for
13   if Lmax <= L[i] then Iopt:=i; Lmax:=L[i] endif
14 end for
```

Figure 8.29. Code for LZ compression.

An example in point given in recent literature [50] is the code for LZ data compression shown in Figure 8.29. The LZ algorithm, designed by Lempel and Ziv in 1977, is one of the most popular algorithms for data compression. A lot of work has been devoted to optimized implementations, in software, of numerous versions of the LZ algorithm, but their speeds are often too slow for some applications. Therefore, researchers are trying to synthesize specialized circuits in a semiautomatic way from a high-level description (in a language like C); for instance, Hwang and Wu recently presented a VLSI systolic array implementation of LZ compression [50].

The SA form of the program in Figure 8.29 is shown in Figure 8.30. The trick described earlier is applied twice, once for `L` and once for `E`.

This SA form has several benefits. First, output dependencies have disappeared. Therefore, more parallelism is available is the transformed program. This becomes apparent when deriving a systolic circuit out of this program. An elementary cell of our circuit consists of statements **7** through **11**. This "mini-program" has four inputs (`M[j]`, `B[i+j]`, `E[i,j]`, and `L[i,j]`) and two outputs (`L[i,j+1]` and `E[i,j+1]`). We really can consider `L[i,j+1]` and `E[i,j+1]` as the two output signals, as opposed to storage location, thanks to the single-assignment property.

8.2 Static Single Assignment

Static single assignment (SSA) is a limited form of single-assignment form. This limitation is on purpose and is an advantage in many cases, especially for its original goal as an intermediate representation of programs using scalars. The usefulness and applicability of the SSA framework [23] are undisputed, and SSA is still the subject of active research (e.g., [82]).


```
1   Lmax := 0;
2   Iopt := 0;
3   for i := 0 to n-Ls-1 do
4     L[i,0] := 0;
5     E[i,0] := 1;
6     for j := 0 to Ls-1 do
7       if (M[j]=B[i+j] and E[i,j]=1) then
8         L[i,j+1] := L[i,j]+1;
8b        E[i,j+1] := E[i,j]     // E[i,j] equals one
9       else
10        E[i,j+1] := 0;
10b       L[i,j+1] := L[i,j];
11      end if
12    end for
13    if Lmax <= L[i,Ls-1] then
        Iopt:=i; Lmax:=L[i,Ls-1]
      endif
14  end for
```

Figure 8.30. Handwritten SA form for the code in Figure 8.29.

The goal of SSA is to give each *statement* its own private data structure. Formally, SSA can be defined as

$$\begin{array}{l}\forall \mathbf{S} \in \mathcal{S}, \mathbf{T} \in \mathcal{S}, u \in \mathcal{D}(\mathbf{S}), v \in \mathcal{D}(\mathbf{T}) : \\ (\mathbf{S} \neq \mathbf{T}) \Rightarrow (\overline{\mathsf{write}}(u) \cap \overline{\mathsf{write}}(v) = \emptyset)\end{array} \tag{8.15}$$

That is, in the transformed program (indicated by the line over write), two instances spawned by two distinct statements are guaranteed to write to different memory locations. If they are spawned by one statement, then SSA does not make any requirement. In other words, what matters is that the two statements **S** and **7** are different, not that the instances (u and v) are.

In contrast, plain SA gives each statement *instance* its data structure, so for straight-line codes without loops, SSA and SA are no different. Therefore, the program in Figure 8.2 also serves as a valid SSA form for Figure 8.1. (Notice, however, that the "official" SSA form, as defined by its inventors, does not use ϕ in expressions but inserts additional explicit assignments to new temporary variables. An example of the "official" form, as applied to the running example, appears in Figure 8.31. Conceptually, this makes no difference, so we stick to the version with ϕ-expressions. These additional assignments in SSA form, however, are helpful to simplify the construction of SSA.)

To see how SSA really differs from plain SA, let's consider a simple example with loops, like the program in Figure 8.4 on page 170. In contrast to SA, which expands scalar `x` into an array shown in Figure 8.4, SSA just duplicates `x` into two scalars.

```
1   if (..)  then
2     x2 := 0
3   else
4     x4 := 1
5   end if
6   x6 := φ(x2,x4);
6'  y6 := x6+1;
7   x7 := 11
8   if (..)  then
9     x9 := 2
10  else
11    x11 := 3
12  end if
13  x13 := φ(x7,x9,x11);
13' z13 := x13
```

Figure 8.31. An SSA version that complies with the original definition.

The resulting program appears in Figure 8.32. Does the difference matter? It depends on the program properties you want to capture. In a basic block, SSA has the useful property that two occurrences of the same name represent the same value. Outside a block, you lose that property. For instance, there are two occurrences of x2 in the SSA form in Figure 8.32, and each represents a different value. In contrast, in the SA form in Figure 8.7, two references to the same element of array x2 are guaranteed to represent the same value.

```
0   x0 := 0
1   for i := 1 to 10
2     x2 := φ(x0,x2) + i
3   endfor
```

Figure 8.32. Figure 8.4 in static single-assignment form.

Notice that the "official" SSA definition does not try to give a closed-form expression to the ϕ-function in Figure 8.32. Indeed, we could mimic Figure 8.7 and consider the program in Figure 8.33 as the SSA form for Figure 8.4. It all depends on the assumptions made on the reaching definition analysis. Figure 8.32 assumes a representation of def-use chains that is not based on instances, whereas Figure 8.33 assumes an instancewise representation.


```
0   x0 := 0
1   for i := 1 to 10
2     x2 := (if i=1 then x0 else x2) + i
3   endfor
```

Figure 8.33. SSA form for Figure 8.4 with a closed-form expression for ϕ.

Array SSA Array SSA, an extension of SSA to arrays, is introduced in [56]. As in SSA, array SSA enforces that each statement writes into its own data structure. Therefore, array SSA is also defined by Eq. (8.15)—the implicit difference from classical SSA is that original structures (the mapping write) are of array "type." Indeed, (8.15) does not specify whether the original data structure is a scalar, an array, or a graph. Therefore, constructing the left-hand sides of an Array SSA form is easy: We just have to give a new, unique array name to each statement. Usually, the name of the new array is the old name plus the statement number.

As an example, consider the program in Figure 8.27 on page 184. In the first step, converting to array SSA form consists of providing new array names to assignments, so the left-hand expressions in statements **1**, **4**, and **5** simply become a1[1], a4[i+j], and a5[i], respectively. The outcome of this first step appears in Figure 8.34.

```
1   a[1] := 0
2   for i := 1 to n do
3     for j := 1 to n
4       a[i+j] := ...
5       a[i] := ...  ?
6     end for
7   end for
```

Figure 8.34. First step of array SSA conversion for Figure 8.27.

Concerning right-hand sides, all the necessary information is once again provided by instancewise reaching definitions. Equation (5.9) on page 96 tells us that the definitions reaching a[i+j-1] are

$$\mathsf{RD}(2^{i,j}) = \begin{array}{l} \textbf{if } j \geq 2 \\ \textbf{then } \{4^{i,j-1}\} \\ \textbf{else } \ \textbf{if } i \geq 2 \\ \qquad \textbf{then } \{4^{i-1,j}\} \\ \qquad \textbf{else } \{1\} \end{array} \tag{8.16}$$

```
1  a1[1] := 0
2  for i := 1 to n do
3    for j := 1 to n do
4      a4[i+j] := ...
5      a5[i]:=.. if j>=2
                 then
                   a4[i+(j-1)]
                 else  if i>=2
                       then a4[(i-1)+j]
                       else a1[1]
                       end if
                 end if
6    end for
7  end for
```

Figure 8.35. Array SSA for the program in Figure 8.27.

This expression of reaching definitions can be plugged into the transformed code nearly verbatim, as you can see in Figure 8.35. All that remains to be done is to map definitions to the array elements they define. This is not hard, since we know the values of the loop counters (like $(i, j-1)$ when $j \geq 2$) and since subscripts in left-hand sides are the same as in the original program. As an example, when j is greater than 1, we know the loop counters of the reaching definition are $(i, j-1)$, and since the syntactic expression of that definition is `a4[i+j]`, we know we have to read from `a4[i+(j-1)]`.

You can also notice that no ϕ-function is needed: For any given instance of statement 5, (8.16) tells at compile time which definition to read from. There is no run-time ambiguity, as can be shown by the very fact we can peel the loops, as shown in Figure 8.36. Notice also that a second pass of renaming must be applied on the program in Figure 8.36 to restore the array SSA property.

8.3 Further Reading

Conversion to single assignment removes all output and antidependencies. This approach might be an overkill. How to selectively remove such dependencies for the purpose of scheduling is detailed in [11].

Many techniques allow us to reduce the size of arrays in single-assignment programs [17, 61] or, in other words, to go out of an array single-assignment intermediate representation. The simplest idea is to first compute a schedule function for statement instances and to compute the last use of a definition according to (7.14) in Section 7.3. The scheduled time step for this last reference is then known as well, and the associated memory location can be deallocated just after this time step.

```
a1[1] := 0;
a4[1] := ...;
a5[1] := ...  a1[1];
for j := 2 to n do
  a4[1+j] := ...;
  a5[1] := ...a4[1+(j-1)]
end for
for i := 2 to n do
  a4[i+1] := ...;
  a5[i] := ...a4[(i-1)+1]
  for j := 2 to n do
    a4[i+j] := ...;
    a5[i] := ...a4[i+(j-1)]
  end for
end for
```

Figure 8.36. Figure 8.35 after loop peeling.

Regarding static single assignment, [13] extends SSA to *predicated code.* Predicated codes are a type of assembly-level programs where instructions are guarded by Boolean predicates indicating whether or not the instruction's side effects should be committed. Predicates appear in the instruction set of recent microprocessors, including the Itanium(R) processor [51].

Classical SSA can be extended to include more sophisticated ϕ-functions that preserve more information. Indeed, the selection a ϕ must perform can be gated by the conditions of `if` statements or by predicates on loop iteraSuctions. This *gated SSA* is studied in [86].

8.4 Conclusion

Single assignment and static single assignment are two program forms that capture the data flow, or at least a part of it in the case of SSA. This property makes imperative programming closer to functional programming, as illustrated by the SISAL language, among others.

Transforming a program to either form boils down to modifying all left-hand expressions and updating right-hand sides accordingly. Rules to change left-hand sides are pretty simple. However, how to change the right-hand sides is hard because we have to make sure the original data flow is preserved. Changing right-hand sides often requires us to use oracle ϕ-functions, whose implementation can be involved. Fortunately, we saw that right-hand sides, with both forms, can be obtained easily once an instancewise reaching definition analysis is done. Such an analysis also detects cases where ϕ-functions are, in fact, not needed.

Don't be misled, however: Using instancewise reaching definition analysis is no guarantee that *all* ϕ-functions disappear from the array SSA form. When a leaf in the expression of reaching definitions is a set with more than one element, there is an ambiguity — perhaps because the analysis wasn't precise enough, or perhaps because the ambiguity is intrinsic to the program. To see why, consider again the program in Figure 8.24 on page 183. More than one definition possibly reaches statement **6**, as given by (8.14). Without information on the condition of the `if` in statement **3**, no compile-time analysis will ever be able to lift this ambiguity, and therefore a ϕ *has* to appear in both the array SA and the array SSA equivalents of Figure 8.24. Actually, for that particular program, the array SA and array SSA forms coincide, and the SSA form is the code appearing in Figure 8.25.

Chapter 9

Maximal Static Expansion

We have seen that data dependencies hamper automatic parallelization of imperative programs. We have also seen that a general method to reduce the number of dependencies is to change writes so that they access distinct memory locations, that is, to *expand* data structures. However, expanding data structures has a cost. The increase in memory is an obvious cost. However, other costs discussed below may also be incurred. Therefore, a general problem arises: Given a cost criterion, what expansion provides maximum parallelism at the lowest cost?

As we said, the mere size of required memory is a cost, and this might be the chief optimization criterion on systems with little available memory, like embedded systems. Another optimization criterion can be the removal of dependencies that hamper parallel execution [11]. However, the criterion we address in this chapter is related to performance when available memory is not an issue.

To see where this cost comes from, consider again single assignment or static single assignment, and imagine we keep this representation in the generated code: ϕ-functions must be materialized to "merge" multiple reaching definitions. These ϕ-functions represent a run-time overhead, especially for nonscalar data structures or when replicated data are distributed across processors. As detailed in Chapter 8, a symbolic instancewise reaching definition analysis can help significantly reduce the number of ϕ-functions. However, as discussed in Section 8.4, the data flow may be intrinsically dynamic—that is, it is sometimes impossible to predict data flow at compile time. Therefore, even excellent analyses will not be able to avoid ϕ-functions in single-assignment and static single-assignment forms.

There is thus an apparent catch-22. On the one hand, both single-assignment forms expose some of the program's data flow and therefore allow some optimizations like parallelization. On the other hand, they introduce these "oracle" ϕ-functions whose overhead may defeat the purpose of parallelization. Maximal static expansion [8], or MSE for short, solves this contradiction by offering a tradeoff.

However, MSE should not be considered as the *only* or the *best* tradeoff between parallelism and memory usage, but just as *one* possible solution. It does provide, however, a theoretical upper bound on the parallelism in a program. This upper bound is derived under the following two assumptions: The transformed program must be free

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs (2nd edition).* MIT Press, 1996.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *ACM Sym. on Prin. of Prog. Lang. (PoPL)*, pages 1–11, January 1988.

[3] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. of the ACM SIGPLAN Symp. on Prin. and Practice of Parallel Prog. (PPoPP)*, pages 39–50, June 1991.

[4] Th. Ball and J. R. Larus. Using paths to measure, explain and enhance program behavior. *IEEE Computer*, 33(7):57–65, July 2000.

[5] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[6] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, 1992.

[7] D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints.* Ph.D. thesis, Univ. Versailles, February 1998.

[8] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *ACM Sym. on Prin. of Prog. Lang. (PoPL)*, pages 98–106, San Diego, CA, January 1998.

[9] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on El. Computers*, EC-15, 1966.

[10] R. Bodík and S. Anik. Path-sensitive value-flow analysis. In *ACM Sym. on Prin. of Prog. Lang. (PoPL)*, pages 237–251, San Diego, CA, January 1998.

[11] P.-Y. Calland, A. Darte, Y. Robert, and F. Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithms. *Parallel Computing*, 23(1-2):251–266, 1997.

[12] D. C. Cann. *SISAL 1.2: A Bief Introduction and Tutorial.* Lawrence Livermore Nat. Lab., Livermore, CA. `ftp://sisal.llnl.gov/pub/sisal/`.

[13] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques*, Newport Beach, CA, October 1999.

[14] S. Chaterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Array distribution in data-parallel programs. Technical report.

[15] Ph. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proc. ACM Intl. Conf. on Supercomputing*, Philadelphia, May 1996.

[16] A. Cohen. *Program analysis and transformation: From the polytope model to formal languages*. Ph.D. thesis, Univ. Versailles, December 1999.

[17] J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, ed., *Proc. Intl. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G 10.3*, pages 185–194, Caracas, Venezuela, April 1994. North Holland.

[18] J.-F. Collard and J. Knoop. A comparative study of reaching definitions analyses. Technical Report 1998/22, PRiSM, U. of Versailles, 1998.

[19] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proc. ACM Conf. on Princ. of Prog. Lang. (PoPL)*, pages 12–25, Boston, MA, 2000.

[20] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM Sym. on Prin. of Prog. Lang. (PoPL)*, pages 84–96, January 1978.

[21] B. Creusillet. *Array Region Analyses and Applications*. Ph.D. thesis, Ecole des Mines de Paris, December 1996.

[22] B. Creusillet and F. Irigoin. Interprocedural array region analysis. *Intl. J. of Parallel Prog.*, 24(6):513–545, 1996.

[23] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Prog. Lang. Sys.*, 13(4):451–490, October 1991.

[24] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser, 2000.

[25] A. Dolzmann and Th. Sturm. *REDLOG User Manual: A REDUCE Logic Package*. Univ. Passau, Germany, `www.fmi.uni-passau.de/~redlog`.

[26] E. Duesterwald, R. Gupta, and M.-L. Soffa. A practical data flow framework for array reference analysis and its use in optimization. In *ACM SIGPLAN'93 Conf. on Prog. Lang. Design and Implementation*, pages 68–77, June 1993.

[27] B. Chapman et al. Extending Vienna fortran with task parallelism. In *Proc. Intl. Conf. on Parallel and Distributed Systems*, pages 258–263, Hsinchu, Taiwan, December 1994.

[28] P. Feautrier. *Solving Systems of Affine (In)Equalities: PIP's User's Guide*. Univ. Versailles. `http://www.prism.uvsq.fr/~paf`.

[29] P. Feautrier. Array expansion. In *ACM Intl. Conf. on Supercomputing, St. Malo*, pages 429–441, 1988.

[30] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[31] P. Feautrier. Dataflow analysis of scalar and array references. *Intl. J. of Parallel Prog.*, 20(1):23–53, February 1991.

[32] P. Feautrier. Some efficient solution to the affine scheduling problem, Part I, One-dimensional time. *Intl. J. of Parallel Prog.*, 21(5):313–348, October 1992.

[33] P. Feautrier. Some efficient solution to the affine scheduling problem, Part II, Multidimensional time. *Intl. J. of Parallel Prog.*, 21(6), December 1992.

[34] P. Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, eds., *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 79–103. Springer-Verlag, June 1996.

[35] P. Feautrier. A parallelization framework for recursive tree programs. In *Europar'98*, number 1470 in LNCS, pages 470–479. Springer-Verlag, September 1998.

[36] J. Ferrante, D. Grunwald, and H. Srinivasan. Computing communication sets for control parallel programs. In *W. on Lang. and Comp. for Par. Comp. (LCPC)*, volume 892 of *LNCS*, pages 316–330, Ithaca, NY, August 1994. Springer-Verlag.

[37] M. Gerndt and R. Berrendorf. *SVM-Fortran, Reference Manual, Version 1.4*. Research Center Jülich, May 1995.

[38] S. Gorlatch and H. Bischof. A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel Proc. Lett.*, 8(4):447–458, December 1998.

[39] Ph. Granger. *Analyses semantiques de congruence*. Ph.D. thesis, Ecole Polytechnique, 1991.

[40] Ph. Granger. Static analysis of linear congreunce equalities among variables of a program. 1991.

[41] M. Griebl and C. Lengauer. On scanning space-time mapped while loops. In B. Buchberger, ed., *Parallel Processing: CONPAR 94 – VAPP VI*, Lecture Notes in Computer Science 854, pages 677–688, Linz, Austria, 1994. Springer-Verlag.

[42] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[43] D. Grunwald and H. Srinivasan. Data flow equations for explicitely parallel programs. In *Proc. ACM Conf. on Prin. of Prog. Lang. (PoPL)*, pages 159–168, May 1993.

[44] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *ACM Supercomputing'95*, San Diego, CA, December 1995.

[45] J. Gu, Z. Li, and G. Lee. Experience with efficient array data flow analysis for array privatization. In *ACM SIGPLAN Sym. on Princ. and Prac. of Par. Prog. (PPoPP)*, Las Vegas, June 1997.

[46] E. M. Gurari. *An Introduction to the Theory of Computation*. Computer Science Press, 1989.

[47] N. Heintze, J. Jaffar, and R. Voicu. A framework for combining analysis and verification. In *Proc. ACM Conf. on Princ. of Prog. Lang. (PoPL)*, pages 26–39, Boston, MA, 2000.

[48] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. In *Intl. Conf. on Parallel Proc.*, pages II–49 – II–56, 1989.

[49] C. A. Herrmann and C. Lengauer. On the space-time mapping of a class of divide-and-conquer recursions. *Parallel Proc. Lett.*, 6(4):525–537, 1996.

[50] S.-A. Hwang and C.-W. Wu. Unified VLSI systolic array design for LZ data compression. *IEEE Trans. on VLSI Systems*, 9(4):489–499, August 2001.

[51] Intel Corp. *Intel IA-64 Application Developer's Architecture Guide*, 2000.

[52] J. L. Jensen, M. E. Jorgensen, M. I. Schwartzbach, and N. Klarlund. Automatic verification of pointer programs using monadic second-order logic. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implem. (PLDI)*, 1997.

[53] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.

[54] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Calculator and Library, version 1.1.0*. Univ. Maryland, November 1996. `http://www.cs.umd.edu/projects/omega/index.html`.

[55] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. *Intl. J. of Parallel Prog.*, 24(6):579–598, 1996.

[56] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *ACM Sym. on Prin. of Prog. Lang. (PoPL)*, pages 107–120, San Diego, CA, January 1998.

[57] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Proc. of the 4th Intl. Conf. on Compiler Construction (CC'92)*, number 641 in LNCS, Paderborn, Germany, 1992.

[58] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Stelle Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[59] S. Y. Kung. *VLSI Array Processors*. Prentice Hall, 1998.

[60] J. Lee, S. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *W. on Lang. and Comp. for Par. Comp. (LCPC)*, August 1997.

[61] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *J. Parallel Comp.*, 24:649–671, 1998.

[62] C. Lengauer. Loop parallelization in the polytope model. In E. Best, ed., *CONCUR '93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

[63] B. Lisper. Data parallelism and functional programming. In G.-R. Perrin and A. Darte, eds., *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 220–251. Springer-Verlag, June 1996.

[64] V. Maslov. Global value propagation through value flow graph and its use in dependence analysis. Technical Report CS-TR-3310, CS Dept, Univ. of Maryland, 1994.

[65] V. Maslov. Enhancing array dataflow dependence analysis with on-demand global value propagation. In *Proc. Intl. Conf. on Supercomputing*, pages 265–269, Barcelona, 1995.

[66] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *Proc. ACM Conf. on Prin. of Prog. Lang. (PoPL)*, pages 2–15, January 1993.

[67] S. Moon and M. W. Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. In *Proc. ACM SIGPLAN Symp. on Princ. and Prac. of Parallel Prog. (PPoPP)*, Atlanta, GA, May 1999.

[68] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[69] OpenMP Architecture Review Board. *OpenMP Fortran application program interface, 2.0*, November 2000.

[70] G. D. Plotkin. *Structural operational semantics*. Lecture Notes, DAIMI FN-19. Aarhus Univ., Denmark, 1981.

[71] W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. *Comm. of the ACM*, 8:102–114, August 1992.

[72] W. Pugh. A practical algorithm for exact array dependence analysis. *Comm. of the ACM*, 35(8):27–47, August 1992.

[73] W. Pugh. Counting solutions to Presburger formulas: How and why. Technical Report CS-TR-3234, Univ. Maryland, March 1994.

[74] W. Pugh and E. Rosser. Iteration space slicing and its application to communication optimization. In *International Conference on Supercomputing*, Vienna, Austria, July 1997. Springer-Verlag.

[75] W. Pugh and E. Rosser. Iteration space slicing for locality. In L. Carter and J. Ferrante, eds., *Workshop on Lang. and Compilers for Parallel Comp. (LCPC)*, volume 1863 of *LNCS*, pages 164–184, La Jolla, CA, 1999. Springer-Verlag.

[76] W. Pugh and D. Wonnacott. Eliminating false data dependences using the omega test. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Implem. (PLDI)*, 1992.

[77] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Trans. on Prog. Lang. and Systems*, 1891, 1998.

[78] J. Ramanujam. Non-unimodular transformations of nested loops. In *Supercomputing'92*, pages 214–223, November 1992.

[79] X. Redon. *Détection et exploitation des récurrences dans les programmes scientifiques en vue de leur parallélisation*. Ph.D. thesis, Univ. Versailles, 1995.

[80] X. Redon and P. Feautrier. Detection of reductions in sequentials programs with loops. In A. Bode, M. Reeve, and G. Wolf, eds., *Proc. 5th Intl. Parallel Architectures and Lang. Europe*, pages 132–145, June 1993.

[81] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *ACM Sym. on Prin. of Prog. Lang. (PoPL)*, pages 12–27, 1988.

[82] A. V. S. Sastry and R. D. C. Ju. A new algorithm for scalar register promotion based on SSA form. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implem. (PLDI)*, pages 15–25, Montreal, June 1998.

[83] A. Schrijver. *Theory of linear and integer programming*. Wiley, 1986.

[84] R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, July 1981.

[85] R. E. Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, July 1981.

[86] P. Tu and D. Padua. Gated SSA-Based demand-driven symbolic analysis for parallelizing compilers. In *ACM Intl. Conf. on Supercomputing*, pages 414–423, Barcelona, July 1995.

[87] M. Weiser. Program slicing. *IEEE Trans. on Software Eng.*, pages 352–357, 1984.

[88] D. Wonnacott. *Constraint-based Array Dependence Analysis*. Ph.D. thesis, Univ. Maryland, 1995.

[89] D. Wonnacott. Extending scalar optimizations for arrays. In *Workshop on Lang. and Compilers for Parallel Comp. (LCPC)*, 2000.

[90] D. Wonnacott and W. Pugh. Nonlinear array dependence analysis. In *Proc. 3rd Workshop on Lang., Compilers and Run-Time Systems for Scalable Comp.*, 1995. Troy, NY.