CHAPTER 1

Array SSA Form

Vivek Sarkar
Kathleen Knobe
Stephen Fink

Progress: 45%

Material gathering in progress

1.1 Introduction

In this chapter, we introduce an Array SSA form that captures precise element-level data flow information for array variables, and coincides with standard SSA form when applied to scalar variables. It can also be applied to structures, objects and other variable types that can be modeled as arrays. ~~As we will see, Array SSA form has a more powerful ϕ function than scalar SSA form, since it can merge values from distinct definitions on an element by element basis. There are several potential uses for Array SSA form in compiler analysis and optimization for uniprocessor and multiprocessor systems~~. In this chapter, we will use *constant propagation* and *conditional constant propagation* as exemplars of program analyses that can be extended to array variables using Array SSA form, and *redundant load elimination* and *dead store elimination* as exemplars of program optimizations that can be extended to array variables and heap objects using Array SSA form. As ~~with many~~ for the corresponding algorithms based on scalar SSA form, the algorithms presented in this chapter are linear in the size of the Array SSA form representation. Though Array SSA form can be made manifest at run-time (*e.g.,* when enabling parallelization via storage duplication [10]), all the algorithms described in this chapter use Array SSA form as a basis for program analysis, which means that the Array SSA form structures can be removed after the program properties of interest have been discovered.

Needs more details

what about transformations under partial array SSA?

You should clearly state that (if I understood well):
- full array SSA is intended only for what you call run-time evaluation
- partial array SSA is intended only for program analysis ie you do not do any transformation under partial SSA

Please no citations in the main part:
Explain what you mean instead.

The rest of the chapter is organized as follows. Section 1.2 reviews full Array SSA form for run-time evaluation as in ~~[10]~~, and also introduces partial Array SSA form for static analysis. Section 1.3 describes how we extend the constant propagation lattice so that it can efficiently record information about array elements. Section 1.4 presents an extension to the Sparse Constant propagation (SC) algorithm from [15] that enables constant propagation through array elements. For simplicity, the algorithm in section 1.4 is restricted to cases in which both the subscript and the value of an array definition are constant. Section 1.5 generalizes the algorithm from section 1.4 so that it can operate on non-constant (symbolic) array subscripts as well *e.g.,* to propagate a def such as $A[m] := 99$ into a use of $A[m]$ even if m is not a constant. Section 1.6 shows how Array SSA form can be extended to support analysis and optimization of object field and array element accesses in strongly typed languages, and section 1.7 contains our conclusions.

1.2 Array SSA Form

The goal of Array SSA form is to provide the same benefits for arrays that traditional SSA provides for scalars. Section 1.2.1 summarizes the *full Array SSA form* introduced in [10]. Full Array SSA form provides exact use-def information at run-time for each dynamic read access of an array element. Section 1.2.2 introduces *partial Array SSA form* as a representation for static analysis; partial Array SSA form provides conservative use-def information at compile-time for each static read access of an array element. Throughout this chapter, we assume that all array operations in the input program are expressed as reads and writes of individual array elements. The extension to more complex array operations (*e.g.,* on array sections or whole arrays) has been omitted to simplify the presentation of this chapter.

1.2.1 *Full Array SSA Form*

The reader should know what is SSA at this point :-)

~~As a reminder, the salient properties of traditional scalar SSA form are as follows [4]:~~

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

Yes but you can illustrate it on fig 1.3 also.

It is important to note that the ϕ function in traditional SSA form statement, $S_3 := \phi(S_1, S_2)$, is not a pure function of S_1 and S_2 because its value depends on the control flow path taken to reach the statement.

```
if (C) then
    S := ...
else
    S := ...
end if
```

Fig. 1.1 Control Flow with Scalar Definitions

You can drop this and explain everything with fig 1.3 that is quite simple.

```
@S_1 := ( ) ; @S_2 := ( )

if (C) then
    S_1 := ...
    @S_1 := (1)
else
    S_2 := ...
    @S_2 := (1)
end if
S_3 := Φ(S_1, @S_1, S_2, @S_2)
@S_3 := max(@S_1, @S_2)
```

Fig. 1.2 After conversion of figure 1.1 to Array SSA form

This discussion comes too early in the text. You should start giving the semantic of your PHI before.

The full Array SSA form uses Φ operators instead of ϕ functions used by traditional SSA form. The semantics of the Φ operator can be defined as a pure function. This is one respect in which Array SSA form has advantages over traditional SSA form even for scalar variables. @ *variables* (pronounced "at variables") are used to obtain a pure function semantics for the Φ operator. Each ϕ function in traditional SSA form such as $\phi(S_1, S_2)$ is rewritten as $\Phi(S_1, @S_1, S_2, @S_2)$. For each static definition S_k, its @ variable $@S_k$ identifies the most recent *iteration vector* ("timestamp") *at* which S_k was modified by this definition.

The *iteration vector* ~~[16, 14]~~ of a static definition S_k identifies a single iteration in the iteration space of the set of loops that enclose the definition. We do not require that the surrounding loops be structured counted loops or that the surrounding loops be tightly nested. Our only assumption in full Array SSA form is that all loops are single-entry, or equivalently, that the control flow graph is *reducible* [8, 1]. ~~(As we will see in section 1.2.2, this assumption is not necessary for partial Array SSA form.~~) For single-entry loops, we know that each def executes at most once in a

Not here please! Say you describe it further.

This is dûe to your restrictive notion of iteration vector, no? See the discussion on review-1.txt

if you remove the timestamp yes of course.∴ :-)
It would be more helpfull explain why you have this restriction instead.

given iteration of its surrounding loops, hence the iteration vector serves the purpose of a "timestamp".

There is a pb with your iteration vector. See txt file.

Let n be the number of loops that enclose a given definition. For convenience, we treat the outermost region of acyclic control flow in a procedure as a dummy outermost loop with a single iteration. Therefore $n \geq 1$ for each definition. A single point in the iteration space is specified by the iteration vector $\mathbf{i} = (i_1, \ldots, i_n)$, which is an n-tuple of iteration numbers one for each enclosing loop. We assume that all @ variables, $@S_k$, are initialized to the empty vector, $@S_k := (\ )$, at the start of program execution. For each real (non-Φ) definition of a renamed scalar, S_k, we assume that a statement of the form $@S_k := \mathbf{i}$ is inserted immediately after definition S_k, where $\mathbf{i}$ is the current iteration vector for all loops that surround S_k. All @ variables are initialized to the empty vector because the empty vector is the identity element for a lexicographic max operation *i.e.,* $\max((\ ), \mathbf{i}) = \mathbf{i}$, for any @ variable value $\mathbf{i}$.

As a simple example, figure 1.2 shows the Array SSA form for the program in figure 1.1. Note that @ variables $@S_1$ and $@S_2$ are explicit arguments of the Φ operator. In this example of acyclic code, there are only two possible values for each @ variable — the empty vector, ( ), and the unit vector[1], (1).

Figure 1.3 shows an example for-loop and its conversion to Array SSA form. Given a Φ operator, $S_3 := \Phi(S_2, @S_2, S_1, @S_1)$, the value of S_3 is given by the following conditional expression (where $\geq$ denotes a lexicographic greater-than-or-equal comparison of iteration vectors):

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

Each Φ definition in Array SSA form also has an associated @ variable. Specifically, the statement, $@S_3 := \max(@S_2, @S_1)$, is inserted after the Φ definition, $S_3 := \Phi(S_2, @S_2, S_1, @S_1)$, where max represents a *lexicographic maximum* operation of iteration vector values $@S_2$ and $@S_1$. No initialization is required for an @ variable for a Φ definition because its value is completely determined by other @ variables.

The prior discussion was on full Array SSA form for scalar variables; we now describe full Array SSA form for array variables. Figures 1.4 and 1.5 show an example program with an array variable, and the conversion of the program to Array SSA form ~~as defined in [10]~~. The key differences between Array SSA form for array variables and Array SSA form for scalar variables are as follows:

1. **Renamed array variables:**
   All array variables are renamed so as to satisfy the static single assignment property *i.e.,* each definition of an array element is assigned a unique name.

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

Example for-loop:

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

After conversion to Array SSA form:

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

Fig. 1.3 A for-loop and its conversion to Array SSA form

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

Fig. 1.4 Example program with array variables

you do not need a loop to
illustrate array case. Straightline code
is sufficient. The case of a join
is straightforward.

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

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

Fig. 1.5 Conversion of program in figure 1.4 to Full Array SSA Form

Analogous to traditional scalar SSA form, control Φ operators are introduced to generate new names for merging two or more prior definitions, and to ensure that each use refers to exactly one definition.

2. **Definition Φ's:**
A *definition* Φ operator is introduced in Array SSA form to deal with preserving ("non-killing") definitions of arrays. Consider A_0 and A_1, two renamed arrays that originated from the same array variable in the source program such that $A_1[k] := \ldots$ is an update of a single array element and A_0 is the prevailing definition at the program point just prior to the definition of A_1. A definition Φ of the form $A_2 := d\Phi(A_1, @A_1, A_0, @A_0)$ is inserted immediately after the definition for A_1 and $@A_1$. (We use the notation $d\Phi$ when we want to distinguish a definition Φ operator from a control Φ operator.) Since definition A_1 only updates one element of A_0, A_2 represents an element-level merge of arrays A_1 and A_0. Definition Φ's did not need to be inserted for definitions of scalar variables because a scalar definition completely kills the old value of the variable.

This is very verbose.
Just providing the semantic
of the PHI (as given at end of
1.2.1) and mentioning that for
a phi at a definition point
the evaluation can
be optimized (as @A1 is made
only of one defined element)
is sufficient I think.

I do not like the notation.
I would prefer a simple Phi as the
optimisation is not significant.

3. **Array-valued @ variables:**
One consequence of renaming arrays is that each (renamed) array variable, A_j, in Array SSA form has an associated @ variable, $@A_j$, such that $@A_j$ has the same shape (rank and dimension sizes) as array variable A_j. Each update of a single array element of the form $A_j[k] := \ldots$, is followed by the statement of the form $@A_j[k] := \mathbf{i}$ where $\mathbf{i}$ is the iteration vector for the loops surrounding the

definition of A_j. Thus, an array-valued @ variable, $@A_j$, can record a separate iteration vector for each element that is assigned by definition A_j.

4. **Array-valued Φ operators:**
Another consequence of renaming arrays is that a Φ operator for array variables must also return an array value. Consider a (control or definition) Φ operator in a statement of the form, $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$. Its semantics can be specified exactly by the following conditional expression for each element, $A_2[j]$, in the result array A_2:

yes

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

The key extension over the scalar case is that the conditional expression specifies an element-level merge of arrays A_1 and A_0.

1.2.2 *Partial Array SSA Form*

The previous section described full Array SSA form with @ variables and Φ operators that can be evaluated at run-time. This section introduces *partial Array SSA form* as a representation for static analysis. Partial Array SSA form is a compile-time approximation of full Array SSA form that provides conservative use-def information for each static read access of an array element.

Consider a statement of the form, $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$, that contains a Φ operator. A static analysis will need to approximate the computation of this Φ operator by some data flow transfer function, $\mathcal{L}_\Phi$. The inputs and output of $\mathcal{L}_\Phi$ will be *lattice elements* for scalar/array variables that are compile-time approximations of their run-time values. We use the notation $\mathcal{L}(V)$ to denote the lattice element for a scalar or array variable V. Therefore, the statement, $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$, will (in general) be modeled by the data flow equation $\mathcal{L}(A_2) = \underline{\mathcal{L}_\Phi}(\mathcal{L}(A_1), \mathcal{L}(@A_1), \mathcal{L}(A_0), \mathcal{L}(@A_0))$.

bof. This is too much notations. You can easily avoid the notations L_phi, L_dphi, etc in what follows using paraphrases.

Our first observation is that there is no extra information provided at compile-time by the @ variables for any static analysis that does not distinguish between reachable code and unreachable code. In such cases, it is sufficient to model $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$ by a data flow equation of the form $\mathcal{L}(A_2) = \mathcal{L}_\phi(\mathcal{L}(A_1), \mathcal{L}(A_0))$ that does not use lattice variables $\mathcal{L}(@A_1)$ and $\mathcal{L}(@A_0)$. For array variables, the only useful information provided by an @ variable, $@A_1$ (say), at compile-time is an indication of which elements were updated by the assignment to array A_1. However, as we will see in section 1.3, this information is also included in the lattice value $\mathcal{L}(A_1)$ for array A_1.

I do not agree: my understanding is that this simplification is justified because _your_ analysis does not provide any information at compile time.
Actually some analysis such as abstract interpretation can provide symbolic information.
See review.txt

Our second observation is that a static analysis that needs to distinguish between unreachable code and reachable code can do so efficiently by introducing *executable flags* for nodes and edges in the CFG (Control Flow Graph) ~~as in [15]~~. If executable

See note on (un)reachable code on review.txt

I personally find this discussion not very useful as:
- this is quite trivial
- this is restricted to one analysis
Still (if I understand well but some statements are contradictory) this is important to say that for partial (as opposed to full) you do not drop of the control semantic of the phi.

flags are computed in the data flow analysis, then the @ variables again do not provide any useful extra information. In fact, if we consider a control Φ statement $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$, with array values A_1 and A_0 carried by incoming CFG edges $e1$ and $e0$ respectively, then the corresponding data flow equation (in the presence of unreachable code elimination) will be $\mathcal{L}(A_2) = \mathcal{L}_\Phi(\mathcal{L}(A_1), X_{e1}, \mathcal{L}(A_0), X_{e0})$ where X_{e1} and X_{e0} are the executable flags for edges $e1$ and $e0$. ~~(Full details can be found in [11].)~~

You can remove as you already said all that.

Since @ variables need not be modeled for the compile-time analyses discussed in this chapter, we drop them and use the ϕ operator, $A_2 := \phi(A_1, A_0)$ instead of $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$ in *partial* Array SSA form. A consequence of dropping @ variables is that partial Array SSA form does not need to deal with iteration vectors, and therefore does not require the control flow graph to be *reducible* as in full Array SSA form. The use of ϕ operators without @ variables brings partial Array SSA form closer to traditional SSA form. The key difference is that partial Array SSA form still contains renamed arrays and definition ϕ operators for updates to array elements.

1.3 Array Lattice

In this section, we describe the lattice representation used to model array values for the analyses in this chapter. Constant propagation for scalar variables has historically been performed by efficient dataflow analysis ~~or abstract interpretation~~ techniques in which values of variables are modeled as *lattice elements* [9, 15]. Given a scalar variable v, the usual approach is to allow the value of its lattice element $\mathcal{L}(v)$ to be $\top$, *Constant* or $\bot$. When $\mathcal{L}(v)$ is *Constant*, the lattice element also contains the value of the constant[2].

I don't think so

Formally, a lattice used for scalar constant propagation consists of:

Lattice formalism is considered really crippling by some people. You shoud:
- refer to the chapter "propagation engine".
- Recall the semantic (use the same than the one of the prop engine chapter) of the bottom and top.
Note that you sometimes use bottom for top and the reverse. Please check.

1. A set of lattice elements: a lattice element for a program variable v is written as $\mathcal{L}(v)$, and denotes $\text{SET}(\mathcal{L}(v))$ = a set of possible values for variable v.
2. $\top$ ("top") and $\bot$ ("bottom"), two distinguished elements of $\mathcal{L}$. The sets denoted by these lattice elements are $\text{SET}(\top) = \{\ \}$ (the empty set), and $\text{SET}(\bot) = \mathcal{U}^v$, where $\mathcal{U}^v$ is the universal set of values for variable v.
3. If $\mathcal{L}(v)$ is a *Constant* lattice element $\text{SET}(\mathcal{L}(v)) = \{Constant\}$, the singleton set containing a constant.
4. A *join* operator, $\sqcap$, such that for any lattice element e, $e \sqcap \top = e$ and $e \sqcap \bot = \bot$. The $\sqcap$ operator on lattice elements corresponds to the set *union* operation on the sets denoted by lattice elements *i.e.*, if e and f are two lattice elements, then $\text{SET}(e \sqcap f) = \text{SET}(e) \cup \text{SET}(f)$.

extensions to be put in last section

[2] An extension that is sometimes employed is to also include lattice elements that can represent a small set of constants or a range of constants[7]. This functionality is not addressed in our chapter, but would be a straightforward extension to our framework.

~~It follows that the $\sqcap$ operator is idempotent, commutative, and associative, and that the lattice is *complete i.e.,* $\sqcap$ is closed on $\mathcal{L}$.~~

5. ~~A $\sqsupseteq$ operator such that $e \sqsupseteq f$ if and only if $e \sqcap f = f$, and a $\sqsupset$ operator such that $e \sqsupset f$ if and only if $e \sqsupseteq f$ and $e \neq f$.~~
~~The $\sqsupseteq$ and $\sqsupset$ operators on lattice elements correspond to the *inclusive subset* and *proper subset* operations on the sets denoted by lattice elements *i.e.,* $e \sqsupseteq f$ if and only if $\text{SET}(e) \subseteq \text{SET}(f)$, and $e \sqsupset f$ if and only if $\text{SET}(e) \subset \text{SET}(f)$.~~
~~The $\sqsupset$ operator defines a partial order on lattice elements. If $e \sqsupset f$, we say that e is "above" f and that f is "below" e in the lattice. Hence, $\top$ is above all other lattice elements and $\bot$ is below all other lattice elements.~~

~~The *height* H of lattice $\mathcal{L}$ is the length of the largest sequence of lattice elements $e_1, e_2, \ldots, e_H$ such that $e_i \sqsupset e_{i+1}$ for all $1 \le i < H$. The height of the lattice for scalar constant propagation is 3.~~

This can be simplified and shortened I think: See note on review.txt (item about lattice representation)

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

1. $\mathcal{L}(A) = \top \quad \Rightarrow \quad \text{SET}(\mathcal{L}(A)) = \{\ \}$
This "top" case indicates that the set of possible index-element pairs that have been identified thus far for A is the empty set, $\{\ \}$.
2. $\mathcal{L}(A) = \langle (i_1, e_1), (i_2, e_2), \ldots \rangle$
$\Rightarrow \quad \text{SET}(\mathcal{L}(A)) = \{(i_1, e_1), (i_2, e_2), \ldots\} \cup (\mathcal{U}^A_{ind} - \{i_1, i_2, \ldots\}) \times \mathcal{U}^A_{elem}$
In general, the lattice element for this "constant" case is represented by a finite list of index-element pairs, $\langle (i_1, e_1), (i_2, e_2), \ldots \rangle$ where $i_1, i_2, \ldots$ are constant index values, and $e_1, e_2, \ldots$ are constant element values. The constant indices, $i_1, i_2, \ldots$, must represent distinct (non-equal) index values. As in the scalar case, the lattice ordering ($\sqsupset$) for these elements is determined by the subset relationship among the sets that they denote.
The meaning of this "constant" lattice element is that the current stage of analysis has identified some finite number of constant index-element pairs for array variable A, such that $A[i_1] = e_1$, $A[i_2] = e_2$, etc. All other elements of A are assumed to be non-constant. (Extensions to handle non-constant indices are described in section 1.5.)
3. $\mathcal{L}(A) = \bot \quad \Rightarrow \quad \text{SET}(\mathcal{L}(A)) = \mathcal{U}^A_{ind} \times \mathcal{U}^A_{elem}$
This "bottom" case indicates that, according to the approximation in the current stage of analysis, array A may take on any value from the universal set of index-element pairs. Note that $\mathcal{L}(A) = \bot$ is equivalent to an empty list, $\mathcal{L}(A) = \langle\ \rangle$, in case (2) above; they both denote the universal set of index-element pairs.

Regardless of the size of an array, A, the number of index-element pairs in $\mathcal{L}(A)$ is bounded by the number of static assignments to A in the source. For the sake of efficiency, we will further restrict the constant array lattice elements, $\mathcal{L}(A) = \langle (i_1, e_1), (i_2, e_2), \ldots \rangle$, to lists that are bounded in size by some constant, why do you do so?

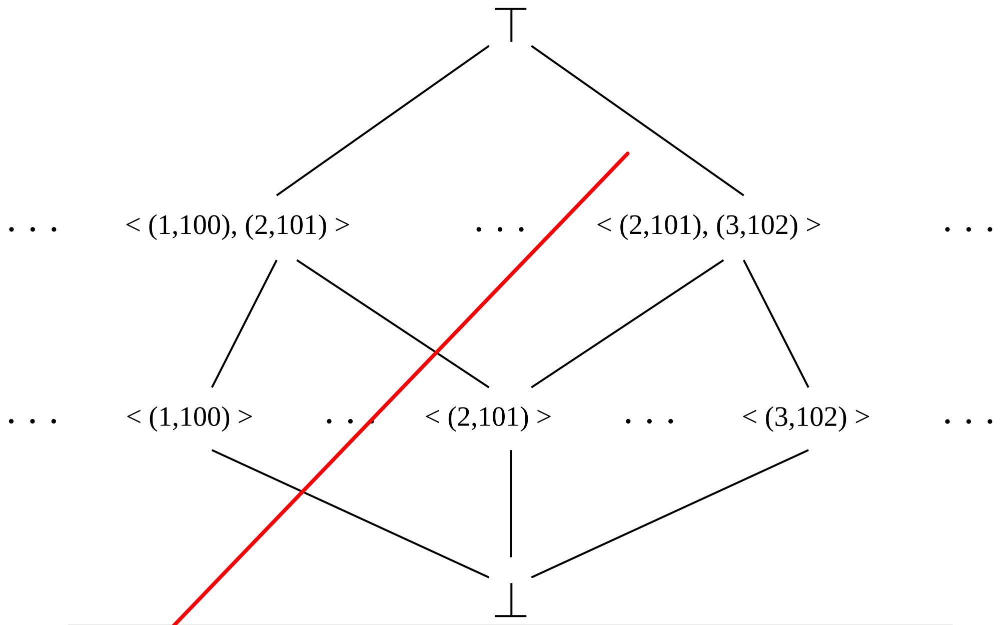

Fig. 1.6 Lattice elements of array values with maximum list size $Z = 2$

$Z \geq 1$. The lattice structure for the $Z = 2$ case is shown in figure 1.6. This lattice has four levels. The second level (just below $\top$) contains all possible lists that contain exactly two constant index-element pairs. The third level (just above $\perp$) contains all possible lists that contain a single constant index-element pair. As mentioned earlier, the lattice ordering is determined by the subset relationship among the sets denoted by the lattice elements. For example, consider two lattice elements $\mathcal{L}_1 = \langle(1,100),(2,101)\rangle$ and $\mathcal{L}_2 = \langle(2,101)\rangle$. The sets denoted by these lattice elements are:

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

Therefore, $\textsc{set}(\mathcal{L}_1)$ is a proper subset of $\textsc{set}(\mathcal{L}_2)$ and we have $\mathcal{L}_1 \sqsupset \mathcal{L}_2$ *i.e.,* $\mathcal{L}_1$ is above $\mathcal{L}_2$ in the lattice in figure 1.6.

The notation make it more complicated than it is actually (till end of page 12): this is actually nothing else than the cross product of all elements of the array.
An "array" notation of your lattice prior to your notation with "sets" (that is useful for symbolic index & objects) would help. Then you could avoid going into the details of figs 1.7, 1,8, etc

We now describe how array lattice elements are computed for various operations that appear in Array SSA form. We start with the simplest operation *viz.,* a reference (read access) to an array element. Figure 1.7 shows how $\mathcal{L}(A_1[k])$, the lattice element for array reference $A_1[k]$, is computed as a function of $\mathcal{L}(A_1)$ and $\mathcal{L}(k)$, the lattice elements for A_1 and k. We denote this function by $\mathcal{L}_{[\,]}$ *i.e.,* $\mathcal{L}(A_1[k]) = \mathcal{L}_{[\,]}(\mathcal{L}(A_1),\mathcal{L}(k))$. The interesting case in figure 1.7 occurs in the middle cell when neither $\mathcal{L}(A_1)$ nor $\mathcal{L}(k)$ is $\top$ or $\perp$. In this case, $\mathcal{L}(k) = Constant$ and $\mathcal{L}(A_1)$ is a nonempty list of the form, $\langle(i_1,e_1),\ldots\rangle$. For this case, if there exists an index-element pair (i_j,e_j) in $\mathcal{L}(A_1)$ such that i_j is the same as the constant $\mathcal{L}(k)$,

$\mathcal{L}(A_1[k])$	$\mathcal{L}(k) = \top$	$\mathcal{L}(k) = Constant$	$\mathcal{L}(k) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\top$	$\bot$
$\mathcal{L}(A_1) = \langle (i_1, e_1), \ldots \rangle$	$\top$	e_j, if $\exists\ (i_j, e_j) \in \mathcal{L}(A_1)$ with $\mathcal{DS}(i_j, \mathcal{L}(k)) = true$ $\bot$, otherwise	$\bot$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 1.7 Lattice computation for $\mathcal{L}(A_1[k]) = \mathcal{L}_{[\,]}(\mathcal{L}(A_1), \mathcal{L}(k))$, where $A_1[k]$ is an array element read operator

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

Fig. 1.8 Lattice computation for $\mathcal{L}(A_1) = \mathcal{L}_{d[\,]}(\mathcal{L}(k), \mathcal{L}(i))$, where $A_1[k] := i$ is an array element write operator

then the value returned for $\mathcal{L}(A_1[k])$ is the constant e_j. Otherwise, $\bot$ is returned as the value for $\mathcal{L}(A_1[k])$.

The notation $\mathcal{DS}$ in the middle cell in figure 1.7 represents a “definitely-same” binary relation *i.e.,* $\mathcal{DS}(a, b) = true$ if and only if a and b are known to have exactly the same value. If, as in the current discussion, a and b are constants then $\mathcal{DS}(a, b) = true$ if and only if $a = b$. However, we use the $\mathcal{DS}$ notation for generality because later in section 1.5 we show how the lattice modeling of arrays introduced in this section can be extended to symbolic index values.

Next, consider a definition (write access) of an array element, which in general has the form $A_1[k] := i$. Figure 1.8 shows how $\mathcal{L}(A_1)$, the lattice element for the array being written into, is computed as a function of $\mathcal{L}(k)$ and $\mathcal{L}(i)$, the lattice elements for k and i. We denote this function by $\mathcal{L}_{d[\,]}$ *i.e.,* $\mathcal{L}(A_1) = \mathcal{L}_{d[\,]}(\mathcal{L}(k), \mathcal{L}(i))$. As before, the interesting case in figure 1.8 occurs in the middle cell when both $\mathcal{L}(k)$ and $\mathcal{L}(i)$ are constant. For this case, the value returned for $\mathcal{L}(A_1)$ is simply the singleton list, $\langle\ (\mathcal{L}(k), \mathcal{L}(i))\ \rangle$, which contains exactly one constant index-element pair.

Now, we turn our attention to the ϕ functions. Consider a definition ϕ operation of the form, $A_2 := d\phi(A_1, A_0)$. The lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{d\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0))$ is shown in figure 1.9. Since A_1 corresponds to a definition of a single array element, the list for $\mathcal{L}(A_1)$ can contain at most one pair (see figure 1.8). Therefore, the three cases considered for $\mathcal{L}(A_1)$ in figure 1.9 are $\mathcal{L}(A_1) = \top$, $\mathcal{L}(A_1) = \langle (i', e') \rangle$, and $\mathcal{L}(A_1) = \bot$.

The notation $\textsc{update}((i', e'), \langle (i_1, e_1), \ldots \rangle)$ used in the middle cell in figure 1.9 denotes a special update of the list $\mathcal{L}(A_0) = \langle (i_1, e_1), \ldots \rangle$ with respect to the constant index-element pair (i', e'). UPDATE involves four steps:

1. Compute the list $T = \{ (i_j, e_j) \mid (i_j, e_j) \in \mathcal{L}(A_0)$ and $\mathcal{DD}(i', i_j) = true \}$. List T contains only those pairs from $\mathcal{L}(A_0)$ that have an index value i_j that is *definitely different* from i'.
   Analogous to $\mathcal{DS}$, $\mathcal{DD}$ denotes a "definitely-different" binary relation *i.e.,*$\mathcal{DS}(a, b) = true$ if and only if a and b are known to have distinct (non-equal) values.
2. Insert the pair (i', e') into T to obtain a new list, I.
3. If the size of list I exceeds the threshold size Z, then one of the pairs in I is dropped from the output list so as to satisfy the size constraint. (Since the size of $\mathcal{L}(A_0)$ must have been $\leq Z$, it is sufficient to drop only one pair to satisfy the size constraint.)
4. Return I as the value of UPDATE$((i', e'), \langle (i_1, e_1), \ldots \rangle)$.

I do not see any interest of doing this as what you do is not abstract interpretation. The size of the list is bounded by the size of the static number of size of assignments.
In all cases this is more an optimization that you can keep for the last section of the chapter

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

Fig. 1.9 Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{d\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0))$ where $A_2 := d\phi(A_1, A_0)$ is a definition ϕ operation

$\mathcal{L}(A_2) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle (i_1, e_1), \ldots \rangle$	$\mathcal{L}(A_0) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\mathcal{L}(A_0)$	$\bot$
$\mathcal{L}(A_1) = \langle (i'_1, e'_1), \ldots \rangle$	$\mathcal{L}(A_1)$	$\mathcal{L}(A_1) \cap \mathcal{L}(A_0)$	$\bot$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 1.10 Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0)) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$, where $A_2 := \phi(A_1, A_0)$ is a control ϕ operation

Finally, consider a control ϕ operation that merges two array values, $A_2 := \phi(A_1, A_0)$. The join operator ($\sqcap$) is used to compute $\mathcal{L}(A_2)$, the lattice element for A_2, as a function of $\mathcal{L}(A_1)$ and $\mathcal{L}(A_0)$, the lattice elements for A_1 and A_0 *i.e.,* $\mathcal{L}(A_2) = \mathcal{L}_{\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0)) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$. The rules for computing this join operator are shown in figure 1.10, depending on different cases for $\mathcal{L}(A_1)$ and $\mathcal{L}(A_0)$. The notation $\mathcal{L}(A_1) \cap \mathcal{L}(A_0)$ used in the middle cell in figure 1.10 denotes a simple intersection of lists $\mathcal{L}(A_1)$ and $\mathcal{L}(A_0)$ — the result is a list of pairs that appear in both $\mathcal{L}(A_1)$ and $\mathcal{L}(A_0)$.

We conclude this section by discussing the example program in figure 1.11. The partial Array SSA form for this example is shown in figure 1.12, and the data flow equations for this example are shown in figure 1.13. Each assignment statement in the partial Array SSA form (in figure 1.12) results in one data flow equation (in figure 1.13); the numbering S1 through S8 indicates the correspondence. In general, each data flow equation in our framework has a single lattice variable on its LHS

(Left Hand Side). Also, a lattice variable will only appear on the LHS of at most one equation.

The lattice operations ($\mathcal{L}_\phi$, $\mathcal{L}_{d\phi}$, $\mathcal{L}_{[\,]}$, $\mathcal{L}_{d[\,]}$, $\mathcal{L}_*$) in figure 1.13 depend on the operations within the corresponding statements. For example, there are reads of array elements in the RHS of statements S3 and S5, writes of array elements in the LHS of statements S3 and S5, definition ϕ operators in statements S4 and S6, and a control ϕ operator in statement S7. We also incorporate lattice computations for specific arithmetic operators such as lattice function L_* for the multiply operator in statements S3 and S5. Tables for lattice computations such as L_* are straightforward and are not shown.

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

Fig. 1.11 Sparse Constant Propagation Example

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

Fig. 1.12 Array SSA form for the Sparse Constant Propagation Example

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

Fig. 1.13 Data Flow Equations for the Sparse Constant Propagation Example

1.4 Sparse Constant Propagation for Scalars and Array Elements

This section shows how we extend the Sparse Constant propagation (SC) algorithm from ~~[15]~~ so as to enable constant propagation through array elements. For simplicity, this algorithm is restricted to cases in which both the subscript and the value of an array definition are constant. A generalization to non-constant subscripts is presented next in section 1.5.

Figure 1.14 contains an outline of the sparse constant propagation algorithm for scalar and array variables. It is similar in structure to the sparse constant propagation algorithm for scalars presented in ~~[15]~~. A major difference is that the data flow equations now include the lattice values for array variables, as described in section 1.3. In addition, the algorithm in figure 1.14 uses a worklist of data flow equations (rather than a worklist of SSA edges as in [15]), thus enabling future integration with other analysis algorithms that are based on data flow equations.

Why would it be important to present this particular SCP algorithm? The one presented on chapter "propagation engine" could work also?
Actually we do not see why worklist on dataflow equations is so important.

- If I am correct you should remove all this section and simply apply the algo of chapter prop engine on your example.
- If this is not possible, then you need to clarify this point.

Let us consider how the algorithm in figure 1.14 will work on the data flow equations in figure 1.13. Recall that the Array SSA form for this example program (see figure 1.12) includes the propagation of array Y_2 into two control flow successors, and the propagation into array D_5 from two control flow predecessors.

The initialization step first initializes $\mathcal{L}(Y_0)$ and $\mathcal{L}(D_0)$ to $\perp$, since they are both considered to be unknown variables in this example program. Next, *worklist* is initialized to contain equations S1, S2, S3, S4, S5, S6, S8, since they all contain some terms that are different from $\top$ *i.e.,* terms that contain a constant or equal $\perp$.

For the fixpoint iteration step, the heuristic in figure 1.14 causes the equations in *worklist* to be processed in their textual order *i.e.,* S1, S2, S3, ... (This heuristic is similar to the heuristic of processing basic blocks in a topological order of their positions in the control flow graph.) The iteration terminates when no LHS lattice variable changes its value, thus causing *worklist* to become empty. The final values of the lattice variables obtained after the fixpoint iteration step has completed are shown in figure 1.15.

The above lattice values were obtained assuming $\mathcal{L}(I) = \perp$. If, instead, variable I is known to equal 3 *i.e.,* $\mathcal{L}(I) = 3$, then the lattice variables that would be obtained after the fixpoint iteration step has completed are shown in figure 1.16.

Even if a worklist on equations is mandatory (see previous note) details of the algorithm can be dropped off.

```
/* INITIALIZATION */

   for each lattice variable $\mathcal{L}(v)$ do
      if it is not possible that $\mathcal{L}(v)$ can be recognized as a constant
            at compile-time (e.g., $v$ is a return value from an unknown call)
      then
         $\mathcal{L}(v) \leftarrow \perp$
      else
         $\mathcal{L}(v) \leftarrow \top$
      end if
   end for

   Intialize worklist ← an empty list
   for each equation $E$ do
      if the RHS of equation $E$ has at least one term that is $\neq \top$
            (i.e., at least one term that is $\perp$ or contains a constant)
      then
         Insert equation $E$ into the worklist
      end if
   end for

/* FIXPOINT ITERATION */

   while worklist is not empty do
      $E$ ← remove any equation from worklist
      /* Heuristic: remove an equation that depends on the smallest number
       of other equations in worklist */
      Recompute $\mathcal{L}(v)$, the LHS of equation $E$, based on the values of $E$'s RHS terms
      if the LHS of equation $E$ has changed then
         for each equation $E'$ that uses $\mathcal{L}(v)$ in its RHS do
            Insert equation $E'$ into worklist
         end for
      end if
   end while

/* TERMINATION */

   For each Array SSA variable $v$ such that $\mathcal{L}(v)$ contains a constant, transform the
   program to replace each use of $v$ by a constant, if profitable to do so
```

Fig. 1.14 Algorithm for Sparse Constant propagation (SC) for scalar and array variables

$$\begin{aligned}
&\text{S1:}\ \mathcal{L}(Y_1) = \langle\, (3,99)\, \rangle\\
&\text{S2:}\ \mathcal{L}(Y_2) = \langle\, (3,99)\, \rangle\\
&\text{S3:}\ \mathcal{L}(D_1) = \langle\, (1,198)\, \rangle\\
&\text{S4:}\ \mathcal{L}(D_2) = \langle\, (1,198)\, \rangle\\
&\text{S5:}\ \mathcal{L}(D_3) = \perp\\
&\text{S6:}\ \mathcal{L}(D_4) = \perp\\
&\text{S7:}\ \mathcal{L}(D_5) = \perp\\
&\text{S8:}\ \ \mathcal{L}(Z) = \perp
\end{aligned}$$

Fig. 1.15 Solution to data flow equations from figure 1.13, assuming I is unknown

$$
\begin{array}{ll}
\text{S1:} & \mathcal{L}(Y_1) = \langle\, (3,99)\, \rangle \\
\text{S2:} & \mathcal{L}(Y_2) = \langle\, (3,99)\, \rangle \\
\text{S3:} & \mathcal{L}(D_1) = \langle\, (1,198)\, \rangle \\
\text{S4:} & \mathcal{L}(D_2) = \langle\, (1,198)\, \rangle \\
\text{S5:} & \mathcal{L}(D_3) = \langle\, (1,198)\, \rangle \\
\text{S6:} & \mathcal{L}(D_4) = \langle\, (1,198)\, \rangle \\
\text{S7:} & \mathcal{L}(D_5) = \langle\, (1,198)\, \rangle \\
\text{S8:} & \mathcal{L}(Z) = 198
\end{array}
$$

Fig. 1.16 Solution to data flow equations from figure 1.13, assuming I is known to be = 3

In either case ($\mathcal{L}(I) = \perp$ or $\mathcal{L}(I) = 3$), the resulting constants revealed by the algorithm can be used in whatever analyses or transformations the compiler considers to be profitable to perform. This is the job of the termination step in figure 1.14.

1.5 Beyond Constant Indices

```
k := 2
do i := ...
     ...
     a[i] := k * 5
     ... := a[i]
enddo
```

Fig. 1.17 Example of Constant Propagation through Non-constant Index

In this section we address constant propagation through *non-constant array subscripts*, as a generalization of the algorithm for constant subscripts described in section 1.4. As an example, consider the program fragment in figure 1.17. In the loop in figure 1.17, we see that the read access of $a[i]$ will have a constant value ($k * 5 = 10$), even though the index/subscript value i is not a constant. We would like to extend the framework from sections 1.3 and 1.4 to be able to recognize the read of $a[i]$ as constant in such programs. There are two key extensions that need to be considered for non-constant (symbolic) subscript values:

- For constants, C_1 and C_2, $\mathcal{DS}(C_1, C_2) \neq \mathcal{DD}(C_1, C_2)$. However, for two symbols, S_1 and S_2, it is possible that both $\mathcal{DS}(S_1, S_2)$ and $\mathcal{DD}(S_1, S_2)$ are FALSE, that is, we don't know if they are the same or different.

- For constants, C_1 and C_2, the values for $\mathcal{DS}(C_1, C_2)$ and $\mathcal{DD}(C_1, C_2)$ can be computed by inspection. For symbolic indices, however, some program analysis is necessary to compute the $\mathcal{DS}$ and $\mathcal{DD}$ relations.

We now discuss the compile-time computation of $\mathcal{DS}$ and $\mathcal{DD}$ for symbolic indices. Observe that, given index values I_1 and I_2, only one of the following three cases is possible:

```
Case 1: 𝒟𝒮(I₁, I₂) = FALSE; 𝒟𝒟(I₁, I₂) = FALSE
Case 2: 𝒟𝒮(I₁, I₂) = TRUE; 𝒟𝒟(I₁, I₂) = FALSE
Case 3: 𝒟𝒮(I₁, I₂) = FALSE; 𝒟𝒟(I₁, I₂) = TRUE
```

The first case is the most conservative solution. In the absence of any other knowledge, it is always correct to state that $\mathcal{DS}(I_1, I_2) = \textit{false}$ and $\mathcal{DD}(I_1, I_2) = \textit{false}$.

yes

The problem of determining if two symbolic index values are the same is equivalent to the classical problem of *global value numbering* [2, 12]. If two indices i and j have the same value number, then $\mathcal{DS}(i, j)$ must $= \textit{true}$. The problem of computing $\mathcal{DD}$ is more complex. Note that $\mathcal{DD}$, unlike $\mathcal{DS}$, is not an equivalence relation because $\mathcal{DD}$ is not transitive. If $\mathcal{DD}(A, B) = \textit{true}$ and $\mathcal{DD}(B, C) = \textit{true}$, it does not imply that $\mathcal{DD}(A, C) = \textit{true}$. However, we can leverage past work on array dependence analysis [16] to identify cases for which $\mathcal{DD}$ can be evaluated to *true*. For example, it is clear that $\mathcal{DD}(i, i + 1) = \textit{true}$, and that $\mathcal{DD}(i, 0) = \textit{true}$ if i is a loop index variable that is known to be ≥ 1.

For DD I am not convinced I would like more details. I Still beleive that FADA is more powerful than full array SSA (as it is in DSA) and that data-flow equations are more powerful than partial array SSA.
One advantage of partial array SSA would be its simplicity. So you need to describe here a degenerated version of array dependence analysis to look credible.

Let us consider how the $\mathcal{DS}$ and $\mathcal{DD}$ relations for symbolic index values are used by our constant propagation algorithms. Note that the specification of how $\mathcal{DS}$ and $\mathcal{DD}$ are used is a separate issue from teh precision of the $\mathcal{DS}$ and $\mathcal{DD}$ values. We now describe how the lattice and the lattice operations presented in section 1.3 can be extended to deal with non-constant subscripts.

First, consider the lattice itself. The $\top$ and $\perp$ lattice elements retain the same meaning as in section 1.3 *viz.*, $\text{SET}(\top) = \{\ \}$ and $\text{SET}(\perp) = \mathcal{U}^A_{ind} \times \mathcal{U}^A_{elem}$. Each element in the lattice is a list of index-value pairs where the value is still required to be constant but the index may be symbolic — the index is represented by its value number.

We now revisit the processing of an array element read of $A_1[k]$ and the processing of an array element write of $A_1[k]$. These operations were presented in section 1.3 (figures 1.7 and 1.8) for constant indices. The versions for non-constant indices appear in figure 1.18 and figure 1.19. For the read operation in figure 1.18, if there exists a pair (i_j, e_j) such that $\mathcal{DS}(i_j, \text{VALNUM}(k)) = \textit{true}$ (*i.e.*, i_j and k have the same value number), then the result is e_j. Otherwise, the result is $\top$ or $\perp$ as specified in figure 1.18. For the write operation in figure 1.19, if the value of the right-hand-side, i, is a constant, the result is the singleton list $\langle(\text{VALNUM}(k), \mathcal{L}(i))\rangle$. Otherwise, the result is $\top$ or $\perp$ as specified in figure 1.19.

Let us now consider the propagation of lattice values through $d\phi$ operators. The only extension required relative to figure 1.9 is that the $\mathcal{DD}$ relation used in performing the UPDATE operation should be able to determine when $\mathcal{DD}(i', i_j) = \textit{true}$ if

$\mathcal{L}(A_1[k])$	$\mathcal{L}(k) = \top$	$\mathcal{L}(k) = \text{VALNUM}(k)$	$\mathcal{L}(k) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\top$	$\bot$
$\mathcal{L}(A_1) = \langle (i_1, e_1), \ldots \rangle$	$\top$	e_j, if $\exists\, (i_j, e_j) \in \mathcal{L}(A_1)$ with $\mathcal{DS}(i_j, \text{VALNUM}(k)) = true$; $\bot$, otherwise	$\bot$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 1.18 Lattice computation for $\mathcal{L}(A_1[k]) = \mathcal{L}_{[\,]}(\mathcal{L}(A_1), \mathcal{L}(k))$, where $A_1[k]$ is an array element read operator. If $\mathcal{L}(k) = \text{VALNUM}(k)$, the lattice value of index k is a value number that represents a constant or a symbolic value.

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

Fig. 1.19 Lattice computation for $\mathcal{L}(A_1) = \mathcal{L}_{d[\,]}(\mathcal{L}(k), \mathcal{L}(i))$, where $A_1[k] := i$ is an array element write operator. If $\mathcal{L}(k) = \text{VALNUM}(k)$, the lattice value of index k is a value number that represents a constant or a symbolic value.

i' and i_j are symbolic value numbers rather than constants. (If no symbolic information is available for i' and i_j, then it is always safe to return $\mathcal{DD}(i', i_j) = false$.)

I still do not see why you drop elements. I suggest you talk about this in the last section (and motivate it).

Note that the UPDATE operation (section 1.3, page 11) always returns a non-empty list, even if it uses the most conservative $\mathcal{DD} = false$ approach. Further, the list will always contain the pair (i', e'), so long as the heuristic for dropping a pair to obey the $\leq Z$ size limit chooses a pair other than (i', e') to drop. This is sufficient to handle the propagation of the constant through the symbolic index in the example in figure 1.17. More precise computations of the $\mathcal{DD}$ relation will lead to more precise (longer) lists, and hence lead to discovery of more constants for more complicated programs with symbolic subscripts.

yes but in your case the size of the list is bounded by the number of static assignments.

1.6 Extension to Object References: Redundant Load and Dead Store Elimination in Strongly Typed Languages

In this section, we present a simple, unified approach for the analysis and optimization of object field and array element accesses in strongly typed languages, that works in the presence of object references/pointers. We show how SSA-based program analyses developed for scalars and arrays can be extended to operate on object references in a strongly typed language like Java. This extension models object references as indices into hypothetical *heap arrays*. We then present two new sparse analysis algorithms using the heap array representation; one identifies redundant loads, and the other identifies dead stores. Using strong typing to help disambiguation, these algorithms are more efficient than equivalent analyses for weakly typed languages. Using the results of these algorithms, we can perform scalar replacement transformations to change operations on object fields and array elements into operations on scalar variables.

1.6.1 Analysis Framework

In this section, we describe a unified representation called *Extended Array SSA* form, which can be used to perform sparse dataflow analysis of values through scalars, array elements, and object references. First, we introduce a formalism called *Heap Arrays* which allows us to represent object references with the same representation used to represent named arrays [6]. Then, we show how to use the Extended Array SSA representation and global value numbering to disambiguate pointers with the same framework used to analyze array indices.

1.6.1.1 Heap Arrays

The partitioning of memory locations into heap arrays is analogous to the partitioning of memory locations using type-based alias analysis [5]. The main difference is that our approach also performs a flow-sensitive analysis of element-level accesses to the heap arrays. We model accesses to object fields as follows. For each field x, we introduce a hypothetical one-dimensional heap array, $\mathcal{H}^x$. Heap array $\mathcal{H}^x$ consolidates all instances of field x present in the heap. Heap arrays are indexed by object references. Thus, a GETFIELD of $p.x$ is modeled as a read of element $\mathcal{H}^x[p]$, and a PUTFIELD of $q.x$ is modeled as a write of element $\mathcal{H}^x[q]$. The use of distinct heap arrays for distinct fields leverages the fact that accesses to distinct fields must be directed to distinct memory locations in a strongly typed language. Note that field x is considered to be the same field for objects of types C_1 and C_2, if C_2 is a subtype of C_1.

Recall that arrays in an object-oriented language like Java are also allocated, so both an object reference and an integer subscript are necessary for accessing an array element. Such arrays can be modeled as *two-dimensional* heap arrays, with one dimension indexed by the object reference as in heap arrays for fields, and the second dimension indexed by the integer subscript. Further details on how array objects are handled can be found in [6].

Having modeled object and array references as accesses to named arrays, we can rename heap arrays and scalar variables to build an extended version of Array SSA form [10]. First, we rename heap arrays so that each renamed heap array has a unique static definition. This includes renaming of the dummy definition inserted at the start block to capture the unknown initial value of the heap array.

We insert three kinds of ϕ functions to obtain an *extended* Array SSA form that we use for data flow analyses[3]:

1. A *control* ϕ from scalar SSA form [4].
2. A *definition* ϕ ($d\phi$) from Array SSA form [10, 11].
3. A *use* ϕ ($u\phi$) function creates a new name whenever a statement reads a heap array element. $u\phi$ functions represent the extension in "extended" Array SSA form.

I would prefer not to have another notation.

It would be useful to outline the semantic of phi functions just after an instruction (actually in parallel with the instruction).

refer to "SSI" chapter for this extension that is used whenever we have forward propagation with some information created both on defs and uses.

The main purpose of the $u\phi$ function is to link together load instructions for the same heap array in control flow order. Intuitively, the $u\phi$ function creates a new SSA variable name, with which a sparse dataflow analysis can associate a lattice variable. We present one dataflow algorithm that uses this information for redundant load identification later in the chapter. Other algorithms (eg. constant propagation) will not require a new name at each use, in which case the $u\phi$ function can be ignored.

1.6.1.2 Definitely-Same and Definitely-Different Analyses for Heap Array Indices

?

In this section, we show how the heap arrays of Extended Array SSA form reduce questions of pointer analysis questions regarding array indices. In particular, we show how global value numbering and allocation site information can be used to efficiently compute *definitely-same* ($\mathcal{DS}$) and *definitely-different* ($\mathcal{DD}$) information for heap array indices.

As an example, consider the following Java source code fragment annotated with heap array accesses:

```
r = p ;
q = new Type1 ;
p.y = ... ;      // H^y[p] := ...
q.y = ... ;      // H^y[q] := ...
... = r.y ;      // ... := H^y[r]
```

One analysis goal is to identify the redundant load of `r.y`, enabling the compiler to replace it with a use of scalar temporary that captures the value stored into `p.y`. We need to establish two facts to perform this transformation: 1) object references p and r are identical (definitely same) in all program executions, and 2) object references q and r are distinct (definitely different) in all program executions.

As before, we use the notation $\mathcal{V}(i)$ to denote the value number of SSA variable i. Therefore, if $\mathcal{V}(i) = \mathcal{V}(j)$, then $\mathcal{DS}(i, j) = true$. For the code fragment above, the statement, `p = r`, ensures that p and r are given the same value number (*i.e.,* $\mathcal{V}(p) = \mathcal{V}(r)$), so that $\mathcal{DS}(p, r) = true$.

~~[3] The extended Array SSA form can also be viewed as a sparse data flow evaluation graph [3] for a heap array.~~

Reference to chapter SSI + a few sentences rephrasing why we need to do additional live-range splitting will be sufficient and more clear.

The problem of computing $\mathcal{DD}$ for object references is more complex than value numbering, and relates to the classical work on pointer alias analysis. We outline a simple approach below, which can be replaced by more sophisticated techniques that may be available. We rely on two observations related to allocation-site information:

All this is very vague for me as I do not know the terminology. Please make sure this is wildly used terms. Otherwise be more explicit.

1. Object references that contain the results of distinct allocation-sites must be different.
2. An object reference containing the result of an allocation-site must be different from any object reference that occurs at a program point that dominates the allocation site in the control flow graph. (As a special case, this implies that the result of an allocation site must be distinct from all object references that are method parameters.)

For example, in the above code fragement, the presence of the allocation site in `q = new Type1` ensures that $\mathcal{DD}(p, q) = true$.

This has to be put somewhere else

In the remainder of the chapter, we will use the notation $\mathcal{V}(\mathbf{k})$ to represent a vector of value numbers, $(\mathcal{V}(k_1), \ldots)$, given a vector index $\mathbf{k} = (k_1, \ldots)$. Thus, $\mathcal{DS}(\mathbf{j}, \mathbf{k})$ is *true* if and only if vectors $\mathbf{j}$ and $\mathbf{k}$ have the same size, and their corresponding elements are definitely-same *i.e.,* $\mathcal{DS}(j_i, k_i) = true$ for all i. Analogously, $\mathcal{DD}(\mathbf{j}, \mathbf{k})$ is *true* if and only if vectors $\mathbf{j}$ and $\mathbf{k}$ have the same size, and at least one pair of elements is definitely-different *i.e.,* $\mathcal{DD}(j_i, k_i) = true$ for some i.

1.6.2 Scalar Replacement Algorithms

In this section, we introduce two new analyses based on Extended Array SSA form. These two analyses form the backbone of *scalar replacement* transformations, which replace accesses to memory by uses of scalar temporaries. First, we present an analysis to identify fully redundant loads. Then, we present an analysis to identify dead stores.

Figure 1.20 illustrates three different cases of scalar replacement for object fields. All three cases can be identified by the algorithms presented in this chapter. (Similar examples can be constructed for scalar replacement of array elements.) For the original program in figure 1.20(a), introducing a scalar temporary `T1` for the store (def) of `p.x` can enable the load (use) of `p.x` to be eliminated *i.e.,* to be replaced by a use of `T1`. Figure 1.20(b) contains an example in which a scalar temporary (`T2`) is introduced for the first load of `p.x`, thus enabling the second load of `p.x` to be eliminated *i.e.,* replaced by `T2`. Finally, figure 1.20(c) contains an example in which the first store of `p.x` can be eliminated because it is known to be dead (redundant); no scalar temporary needs to be introduced in this case.

Original program:

```
  p   := new Type1
  q   := new Type1
  . . .
 p.x := ...
 q.x := ...
...   := p.x
```

After redundant load elimination:

```
  p   := new Type1
  q   := new Type1
  . . .
  T1  := ...
 p.x := T1
 q.x := ...
...   := T1
```

(a)

Original program:

```
  p   := new Type1
  q   := new Type1
  . . .
...   := p.x
 q.x := ...
...   := p.x
```

After redundant load elimination:

```
  p   := new Type1
  q   := new Type1
  . . .
  T2  := p.x
...   := T2
 q.x := ...
...   := T2
```

(b)

Original program:

```
  p   := new Type1
  q   := new Type1
  r   := p
  . . .
 p.x := ...
 q.x := ...
 r.x := ...
```

After dead store elimination:

```
  p   := new Type1
  q   := new Type1
  r   := p
  . . .
 q.x := ...
 r.x := ...
```

(c)

Fig. 1.20 Examples of scalar replacement

1.6.2.1 *Redundant Load Elimination*

Figure 1.21 outlines our algorithm for identifying uses (loads) of heap array elements that are redundant with respect to prior defs and uses of the same heap array. The algorithm's main analysis is *index propagation*, which identifies the set of indices that are *available* at a specific def/use A_i of heap array A.

Index propagation is a dataflow problem, which computes a lattice value $\mathcal{L}(\mathcal{H})$ for each heap variable $\mathcal{H}$ in the Array SSA form. This lattice value $\mathcal{L}(\mathcal{H})$ is a set of value number vectors $\{\mathbf{i_1}, \ldots\}$, such that a load of $\mathcal{H}[\mathbf{i}]$ is *available* if $\mathcal{V}(\mathbf{i}) \in \mathcal{L}(\mathcal{H})$. Note that the lattice element does not include the value of $\mathcal{H}[\mathbf{i}]$ (as in constant propagation), just teh fact that it is available. Figures 1.22, 1.23 and 1.24 give the lattice computations which define the index propagation solution. The notation $\textsc{update}(\mathbf{i'}, \langle \mathbf{i_1}, \ldots \rangle)$ used in the middle cell in figure 1.22 denotes a special update of the list $\mathcal{L}(A_0) = \langle \mathbf{i_1}, \ldots \rangle$ with respect to index $\mathbf{i'}$. UPDATE involves four steps:

1. Compute the list $T = \{\ \mathbf{i_j} \mid \mathbf{i_j} \in \mathcal{L}(A_0)$ and $\mathcal{DD}(\mathbf{i'}, \mathbf{i_j}) = \mathit{true}\ \}$. List T contains only those indices from $\mathcal{L}(A_0)$ that are *definitely different* from $\mathbf{i'}$.
2. Insert $\mathbf{i'}$ into T to obtain a new list, I.
3. If the size of list I exceeds the threshold size Z, then one of the indices in I is dropped from the output list so as to satisfy the size constraint. (Since the size of $\mathcal{L}(A_0)$ must have been $\leq Z$, it is sufficient to drop only one index to satisfy the size constraint.)
4. Return I as the value of $\textsc{update}(\mathbf{i'}, \langle \mathbf{i_1}, \ldots \rangle)$.

Input: Intermediate code for method being optimized, augmented with the $\mathcal{DS}$ and $\mathcal{DD}$ relations defined in Section 1.6.1.2.

Output: Transformed intermediate code after performing scalar replacement.

Algorithm:

1. **Build extended Array SSA form for each heap array.**
   Build Array SSA form, inserting control ϕ, $d\phi$ and $u\phi$ functions as outlined in Section 1.6.1.1, and renaming of all heap array definitions and uses.
   As part of this step, we annotate each call instruction with dummy defs and uses of each heap array for which a def or a use can reach the call instruction. If interprocedural analysis is possible, the call instruction's heap array defs and uses can be derived from a simple flow-insensitive summary of the called method.
2. **Perform index propagation.**
   a. Walk through the extended Array SSA intermediate representation, and for each ϕ, $d\phi$, or $u\phi$ statement, create a dataflow equation with the appropriate operator as listed in Figures 1.22, 1.23 or 1.24.
   b. Solve the system of dataflow equations by iterating to a fixed point.

   After index propagation, the lattice value of each heap array, A_i, is $\mathcal{L}(A_i) = \{ \mathcal{V}(\mathbf{k}) \mid \text{location } A[\mathbf{k}] \text{ is "available" at def } A_i \text{ (and all uses of } A_i) \}$.
3. **Scalar replacement analysis.**
   a. Compute *UseRepSet* = { use $A_j[\mathbf{x}]$ | $\exists\ \mathcal{V}(\mathbf{x}) \in \mathcal{L}(A_j)$ } *i.e.,* use $A_j[\mathbf{x}]$ is placed in *UseRepSet* if and only if location $A[\mathbf{x}]$ is available at the def of A_j and hence at the use of $A_j[\mathbf{x}]$. (Note that A_j uniquely identifies a use, since all uses are renamed in extended Array SSA form.)
   b. Compute *DefRepSet* = { def $A_i[\mathbf{k}]$ | $\exists$ use $A_j[\mathbf{x}] \in$ *UseRepSet* with $\mathcal{V}(\mathbf{x}) = \mathcal{V}(\mathbf{k})$ } *i.e.,* def $A_i[\mathbf{k}]$ is placed in *DefRepSet* if and only if a use $A_j[\mathbf{x}]$ was placed in *UseRepSet* with $\mathcal{V}(\mathbf{x}) = \mathcal{V}(\mathbf{k})$.
4. **Scalar replacement transformation.**
   Apply scalar replacement actions selected in step 3 above to the *original* program and obtain the transformed program.

Fig. 1.21 Overview of Redundant Load Elimination algorithm.

$\mathcal{L}(A_2)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle(\mathbf{i_1}), \ldots\rangle$	$\mathcal{L}(A_0) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\top$	$\top$
$\mathcal{L}(A_1) = \langle(\mathbf{i'})\rangle$	$\top$	$\text{UPDATE}((\mathbf{i'}), \langle(\mathbf{i_1}), \ldots\rangle)$	$\langle(\mathbf{i'})\rangle$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 1.22 Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{d\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0))$ where $A_2 := d\phi(A_1, A_0)$ is a definition ϕ operation

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

Fig. 1.23 Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{u\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0))$ where $A_2 := u\phi(A_1, A_0)$ is a use ϕ operation

$\mathcal{L}(A_2) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle(\mathbf{i}_1), \ldots\rangle$	$\mathcal{L}(A_0) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\mathcal{L}(A_0)$	$\bot$
$\mathcal{L}(A_1) = \langle(\mathbf{i}'_1), \ldots\rangle$	$\mathcal{L}(A_1)$	$\mathcal{L}(A_1) \cap \mathcal{L}(A_0)$	$\bot$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 1.24 Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_\phi(\mathcal{L}(A_1), \mathcal{L}(A_0)) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$, where $A_2 := \phi(A_1, A_0)$ is a control ϕ operation

(a) Extended Partial Array SSA form:	(b) After index propagation:	(c) Scalar replacement actions selected:	(d) After transforming original program:
`p := new Type1`	$\mathcal{L}(\mathcal{H}^x_0) = \{\ \}$	$\mathit{UseRepSet} = \{\mathcal{H}^x_4[p]\}$	`p := new Type1`
`q := new Type1`	$\mathcal{L}(\mathcal{H}^x_1) = \{\mathcal{V}(p)\}$	$\mathit{DefRepSet} = \{\mathcal{H}^x_1[p]\}$	`q := new Type1`
`. . .`	$\mathcal{L}(\mathcal{H}^x_2) = \{\mathcal{V}(p)\}$		`. . .`
$\mathcal{H}^x_1[p]$ `:= ...`	$\mathcal{L}(\mathcal{H}^x_3) = \{\mathcal{V}(q)\}$		$A\_temp_{\mathcal{V}(p)}$ `:= ...`
$\mathcal{H}^x_2$ `:=` $d\phi(\mathcal{H}^x_1, \mathcal{H}^x_0)$	$\mathcal{L}(\mathcal{H}^x_4) = \{\mathcal{V}(p), \mathcal{V}(q)\}$		`p.x :=` $A\_temp_{\mathcal{V}(p)}$
$\mathcal{H}^x_3[q]$ `:= ...`	$\mathcal{L}(\mathcal{H}^x_5) = \{\mathcal{V}(p), \mathcal{V}(q)\}$		`q.x := ...`
$\mathcal{H}^x_4$ `:=` $d\phi(\mathcal{H}^x_3, \mathcal{H}^x_2)$			`... :=` $A\_temp_{\mathcal{V}(p)}$
`... :=` $\mathcal{H}^x_4[p]$			
$\mathcal{H}^x_5$ `:=` $u\phi(\mathcal{H}^x_4, \mathcal{H}^x_3)$			

Fig. 1.25 Trace of load elimination algorithm from figure 1.21 for program in figure 1.20(a)

After index propagation, the algorithm selects an array use (load), $A_j[\mathbf{x}]$, for scalar replacement if and only if index propagation determines that an index with value number $\mathcal{V}(\mathbf{x})$ is available at the def of A_j. If so, the use is included in *UseRepSet* the set of uses selected for scalar replacement. Finally, an array def, $A_i[\mathbf{k}]$, is selected for scalar replacement if and only if some use $A_j[\mathbf{x}]$ was placed in *UseRepSet* such that $\mathcal{V}(\mathbf{x}) = \mathcal{V}(\mathbf{k})$. All such defs are included in *DefRepSet* the set of defs selected for scalar replacement.

Figure 1.25 illustrates a trace of this load elimination algorithm for the example program in figure 1.20(a). Figure 1.25(a) shows the partial Array SSA form computed for this example program. The results of index propagation are shown in figure 1.25(b). These results depend on definitely-different analysis establishing that $\mathcal{V}(p) \neq \mathcal{V}(q)$ by using allocation site information as described in Section 1.6.1.2. Figure 1.25(c) shows the scalar replacement actions derived from the results of index propagation, and Figure 1.25(d) shows the transformed code after performing these scalar replacement actions. The load of `p.x` has thus been eliminated in the transformed code, and replaced by a use of the scalar temporary, $A\_temp_{\mathcal{V}(p)}$.

We conclude this section with a brief discussion of the impact of the Java Memory Model (JMM). It has been observed that redundant load elimination can be an illegal transformation for multithreaded programs with data races written for a memory model, such as the JMM, that includes the memory coherence assumption [13]. If necessary, our algorithms can be modified to obey memory coherence by simply treating each $u\phi$ function as a $d\phi$ function *i.e.,* by treating each array use also as an array def.

1.6.2.2 *Dead Store Elimination*

In this section, we show how our Array SSA framework can be used to identify redundant (dead) stores of array elements. Dead store elimination is related to load elimination, because scalar replacement can convert non-redundant stores into redundant stores. For example, consider the program in Figure 1.20(a). If it contained an additional store of `p.x` at the bottom, the first store of `p.x` will become redundant after scalar replacement. The program after scalar replacement will then be similar to the program shown in Figure 1.20(c) as an example of dead store elimination.

Our algorithm for dead store elimination is based on a backward propagation of *DEAD* sets. As in load elimination, the propagation is *sparse i.e.,* it goes through ϕ nodes in the Array SSA form rather than basic blocks in a control flow graph. However, $u\phi$ functions are not used in dead store elimination, since the ordering of uses is not relevant to identifying a dead store. Without $u\phi$ functions, it is possible for multiple uses to access the same heap array name. Hence, we use the notation $\langle A, s\rangle$ to refer to a specific use of heap array A in statement (instruction) s. We also use the term "non-ϕ" to refer to any def or use that does not appear in a (control or definition) ϕ statement *i.e.,* a def or use from the original program.

Consider a ϕ *def* A_i, a ϕ or non-ϕ *use* $\langle A_j, s\rangle$, and a real (non-ϕ) def A_k in Array SSA form. We define the following four sets:

$$\begin{aligned}
DEAD_{def}(A_i) &= \{\ \mathcal{V}(x) \mid \text{element } x \text{ of array } A \text{ is dead at } \phi \text{ def } A_i\ \}\\
DEAD_{use}(\langle A_j, s\rangle) &= \{\ \mathcal{V}(x) \mid \text{element } x \text{ of array } A \text{ is dead at non-}\phi \text{ use of } A_j \text{ in statement } s\ \}\\
KILL(A_k) &= \{\ \mathcal{V}(x) \mid \text{element } x \text{ of array } A \text{ is killed by non-}\phi \text{ def of } A_k\ \}\\
LIVE(A_i) &= \{\ \mathcal{V}(x) \mid \exists \text{ a non-}\phi \text{ use } A_i[x] \text{ of } \phi \text{ def } A_i\ \}
\end{aligned}$$

The *KILL* and *LIVE* sets are local sets *i.e.,* they can be computed immediately without propagation of data flow information. If A_i "escapes" from the procedure (*i.e.,* definition A_i is exposed on procedure exit), then we must conservatively set $LIVE(A_i) = \mathcal{U}^A_{ind}$, the universal set of index value numbers for array A. Note that in Java, every instruction that can potentially throw an uncaught exception must be treated as a procedure exit, although this property can be relaxed with some interprocedural analysis.

The data flow equations used to compute the $DEAD_{def}$ and $DEAD_{use}$ sets are given in Figure 1.26. The goal of our analysis is to find the maximal $DEAD_{def}$ and $DEAD_{use}$ sets that satisfy these equations. Hence our algorithm will initialize each $DEAD_{def}$ and $DEAD_{use}$ set to $= \mathcal{U}^A_{ind}$ (for renamed arrays derived from original array A), and then iterate on the equations till a fixpoint is obtained. After *DEAD* sets have been computed, we can determine if a real (non-ϕ) definition is redundant quite simply as follows. Consider a real definition, $A_1[j] := \ldots$, followed by a definition ϕ statement, $A_2 := d\phi(A_1, A_0)$. Then, if $\mathcal{V}(j) \in DEAD(A_2)$, then def (store) A_1 is redundant and can be eliminated.

1. **Propagation from the LHS to the RHS of a control ϕ:**
Consider a control ϕ statement s of the form, $A_2 := \phi(A_1, A_0)$. In this case, the uses, $\langle A_1, s\rangle$ and $\langle A_0, s\rangle$, must both come from ϕ defs, and the propagation of $DEAD_{def}(A_2)$ to the RHS is a simple copy *i.e.,* $DEAD_{use}(\langle A_1, s\rangle) = DEAD_{def}(A_2)$ and $DEAD_{use}(\langle A_0, s\rangle) = DEAD_{def}(A_2)$.
2. **Propagation from the LHS to the RHS of a definition ϕ:**
Consider a $d\phi$ statement s of the form $A_2 := d\phi(A_1, A_0)$. In this case use $\langle A_1, s\rangle$ must come from a real (non-ϕ) definition, and use $\langle A_0, s\rangle$ must come from a ϕ definition. The propagation of $DEAD_{def}(A_2)$ and $KILL(A_1)$ to $DEAD_{use}(\langle A_0, s\rangle)$ is given by the equation, $DEAD_{use}(\langle A_0, s\rangle) = KILL(A_1) \cup DEAD_{def}(A_2)$.
3. **Propagation to the LHS of a ϕ statement from uses in other statements:**
Consider a definition or control ϕ statement of the form $A_i := \phi(\ldots)$. The value of $DEAD_{def}(A_i)$ is obtained by intersecting the $DEAD_{use}$ sets of all uses of A_i, and subtracting out all value numbers that are not definitely different from every element of $LIVE(A_i)$. This set is specified by the following equation:

$$DEAD_{def}(A_i) = \left(\bigcap_{s \text{ is a } \phi \text{ use of } A_i} DEAD_{use}(\langle A_i, s\rangle) \right) - \{v \mid \exists w \in LIVE(A_i) s.t. \neg DD(v, w)\}$$

Fig. 1.26 Data flow equations for $DEAD_{def}$ and $DEAD_{use}$ sets

1.7 Summary

I do not want any summary. I want references, additional readings, open questions...

Static single assignment (SSA) form for scalars has been a significant advance. It has simplified the way we think about scalar variables. It has simplified the design of some optimizations and has made other optimizations more effective. A direct application of classical SSA form to arrays, would view an array as a single object. But the kinds of analyses that sophisticated compilers need to perform on arrays, for example those that drive loop parallelization, are at the element level. In this chapter, we introduced an Array SSA form that captures precise element-level data flow information for array variables. It is general and simple, and coincides with standard SSA form when applied to scalar variables. It can also be used for structures and other variable types that can be modeled as arrays. We presented efficient algorithms based on Array SSA form that perform constant propagation and conditional constant propagation through both scalar and array references. These algorithms use an extension of the classical constant propagation lattice so that it can efficiently record information about array elements. The original sparse conditional constant propagation algorithm in [15] dealt with control flow and data flow separately by maintaining two distinct work lists. The algorithm presented in this chapter is conceptually simpler because it uses a single set of data flow equations instead.

be careful when you mention loop parallelization as array SSA is still _single_ assignment form. So it is far less powerful than DSA (such as equational based IR -- eg alpha) for this purpose (sorry I have been indoctrinated by some people in my team... :-))

No: you dropped of the control information.

But this is more powerful isn't it????

We also presented a unified framework to analyze object-field and array-element references for programs written in strongly-typed languages such as Java and Modula-3. Our solution models object references as heap arrays, and uses global value numbering and allocation site information to determine if two object references are known to be same or different. We presented algorithms to identify fully redundant loads and dead stores, based on sparse propagation in an extended Array SSA form.

There are many possible directions for future research based on this work. One direction is to extend the value numbering and definitely-different analyses mentioned in section 1.5 so that they can be combined with conditional constant propagation rather than performed as a pre-pass. Further, it would be interesting to gain an understanding of when adding lattice elements for @ variables could lead to extra precision in program analysis compared to using executable flags. An ultimate goal is to combine conditional constant and type propagation, value numbering, partial redundancy elimination, and scalar replacement analyses with a single framework that can analyze heap accesses as effectively as scalar operations.

This citation is to make chapters without citations build without error. Please ignore it: [**?**].

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting Equality of Variables in Programs. *Fifteenth ACM Principles of Programming Languages Symposium*, pages 1–11, January 1988. San Diego, CA.
3. Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic Construction of Sparse Data Flow Evaluation Graphs. *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, January 1991.
4. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
5. Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. pages 106–117, May 1998.
6. Stephen J. Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 155–174, London, UK, 2000. Springer-Verlag.
7. William H. Harrison. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, SE-13(3), May 1977.
8. Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, Inc., 1977.
9. G. Kildall. A Unified Approach to Global Program Optimization. *Conference Record of First ACM Symposium on Principles of Programming Languages*, pages 194–206, January 1973.
10. Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in Parallelization. *Conf. Rec. Twenty-fifth ACM Symposium on Principles of Programming Languages, San Diego, California*, January 1998.
11. Kathleen Knobe and Vivek Sarkar. Conditional constant propagation of scalar and array references using array SSA form. In Giorgio Levi, editor, *Lecture Notes in Computer Science, 1503*, pages 33–56. Springer-Verlag, 1998. Proceedings from the *5th International Static Analysis Symposium*.
12. Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1997.
13. William Pugh. Fixing the Java Memory Model. In *ACM Java Grande Conference*, June 1999.
14. Vivek Sarkar. The PTRAN Parallel Programming System. In B. Szymanski, editor, *Parallel Functional Programming Languages and Compilers*, ACM Press Frontier Series, pages 309–391. ACM Press, New York, 1991.
15. Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.
16. Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.