CHAPTER 16

Definite Assignment

All the evolution we know of proceeds from the vague to the definite.
—Charles Peirce

EACH local variable (§14.4) and every blank `final` (§4.5.4) field (§8.3.1.2) must have a *definitely assigned* value when any access of its value occurs. An access to its value consists of the simple name of the variable occurring anywhere in an expression except as the left-hand operand of the simple assignment operator `=`. A Java compiler must carry out a specific conservative flow analysis to make sure that, for every access of a local variable or blank `final` field *f*, *f* is definitely assigned before the access; otherwise a compile-time error must occur.

Similarly, every blank `final` variable must be assigned at most once; it must be *definitely unassigned* when an assignment to it occurs. Such an assignment is defined to occur if and only if either the simple name of the variable, or its simple name qualified by `this`, occurs on the left hand side of an assignment operator. A Java compiler must carry out a specific conservative flow analysis to make sure that, for every assignment to a blank `final` variable, the variable is definitely unassigned before the assignment; otherwise a compile-time error must occur.

The remainder of this chapter is devoted to a precise explanation of the words "definitely assigned before" and "definitely unassigned before".

The idea behind definite assignment is that an assignment to the local variable or blank `final` field must occur on every possible execution path to the access. Similarly, the idea behind definite unassignment is that no other assignment to the blank `final` variable is permitted to occur on any possible execution path to an assignment. The analysis takes into account the structure of statements and expressions; it also provides a special treatment of the expression operators `!`, `&&`, `||`, and `? :`, and of boolean-valued constant expressions.

For example, a Java compiler recognizes that `k` is definitely assigned before its access (as an argument of a method invocation) in the code:

```
{
    int k;
    if (v > 0 && (k = System.in.read()) >= 0)
```

```
        System.out.println(k);
}
```

because the access occurs only if the value of the expression:

```
v > 0 && (k = System.in.read()) >= 0
```

is true, and the value can be `true` only if the assignment to `k` is executed (more properly, evaluated).

Similarly, a Java compiler will recognize that in the code:

```
{
    int k;
    while (true) {
        k = n;
        if (k >= 5) break;
        n = 6;
    }
    System.out.println(k);
}
```

the variable `k` is definitely assigned by the `while` statement because the condition expression `true` never has the value `false`, so only the `break` statement can cause the `while` statement to complete normally, and `k` is definitely assigned before the `break` statement.

On the other hand, the code

```
{
    int k;
    while (n < 4) {
        k = n;
        if (k >= 5) break;
        n = 6;
    }
    System.out.println(k);// k is not "definitely assigned" before this
}
```

`must be rejected by a Java compiler, because in this case the while statement is not guaranteed to execute its body as far as the rules of definite assignment are concerned.`

Except for the special treatment of the conditional boolean operators &&, `||`, and `? :` and of boolean-valued constant expressions, the values of expressions are not taken into account in the flow analysis.

For example, a Java compiler must produce a compile-time error for the code:

```
{
    int k;
```

```
        int n = 5;
        if (n > 2)
            k = 3;
        System.out.println(k);// k is not “definitely assigned” before this
    }
```

even though the value of n is known at compile time, and in principle it can be known at compile time that the assignment to k will always be executed (more properly, evaluated). A Java compiler must operate according to the rules laid out in this section. The rules recognize only constant expressions; in this example, the expression n > 2 is not a constant expression as defined in §15.28.

As another example, a Java compiler will accept the code:

```
    void flow(boolean flag) {
        int k;
        if (flag)
            k = 3;
        else
            k = 4;
        System.out.println(k);
    }
```

as far as definite assignment of k is concerned, because the rules outlined in this section allow it to tell that k is assigned no matter whether the flag is true or false. But the rules do not accept the variation:

```
    void flow(boolean flag) {
        int k;
        if (flag)
            k = 3;
        if (!flag)
            k = 4;
        System.out.println(k); // k is not “definitely assigned” before here
    }
```

and so compiling this program must cause a compile-time error to occur.

A related example illustrates rules of definite unassignment. A Java compiler will accept the code:

```
    void unflow(boolean flag) {
        final int k;
        if (flag) {
            k = 3;
            System.out.println(k);
        }
        else {
            k = 4;
            System.out.println(k);
        }
    }
```

as far as definite unassignment of `k` is concerned, because the rules outlined in this section allow it to tell that `k` is assigned at most once (indeed, exactly once) no matter whether the flag is `true` or `false`. But the rules do not accept the variation:

```
void unflow(boolean flag) {
    final int k;
    if (flag) {
        k = 3;
        System.out.println(k);
    }
    if (!flag) {
        k = 4;        // k is not “definitely unassigned” before here
        System.out.println(k);
    }
}
```

and so compiling this program must cause a compile-time error to occur.

In order to precisely specify all the cases of definite assignment, the rules in this section define several technical terms:

- whether a variable is *definitely assigned before* a statement or expression;
- whether a variable is *definitely unassigned before* a statement or expression;
- whether a variable is *definitely assigned after* a statement or expression; and
- whether a variable is *definitely unassigned after* a statement or expression.

For boolean-valued expressions, the last two are refined into four cases:

- whether a variable is *definitely assigned after* the expression *when true*;
- whether a variable is *definitely unassigned after* the expression *when true*;
- whether a variable is *definitely assigned after* the expression *when false*; and
- whether a variable is *definitely unassigned after* the expression *when false*.

Here *when true* and *when false* refer to the value of the expression.

For example, the local variable k is definitely assigned a value after evaluation of the expression

```
a && ((k=m) > 5)
```

when the expression is `true` but not when the expression is `false` (because if `a` is `false`, then the assignment to `k` is not necessarily executed (more properly, evaluated)).

The phrase “*V* is definitely assigned after *X*” (where *V* is a local variable and *X* is a statement or expression) means “*V* is definitely assigned after *X* if *X* completes normally”. If *X* completes abruptly, the assignment need not have occurred, and the rules stated here take this into account. A peculiar consequence of this definition is that “*V* is definitely assigned after `break;`” is always true! Because a `break` statement never completes normally, it is vacuously true that *V* has been assigned a value if the `break` statement completes normally.

Similarly, the statement “*V* is definitely unassigned after *X*” (where *V* is a variable and *X* is a statement or expression) means “*V* is definitely unassigned after *X* if *X* completes normally”. An even more peculiar consequence of this definition is that “*V* is definitely unassigned after `break;`” is always true! Because a `break` statement never completes normally, it is vacuously true that *V* has not been assigned a value if the `break` statement completes normally. (For that matter, it is also vacuously true that the moon is made of green cheese if the `break` statement completes normally.)

In all, there are four possibilities for a variable *V* after a statement or expression has been executed:

- *V* is definitely assigned and is not definitely unassigned.
  (The flow analysis rules prove that an assignment to *V* has occurred.)
- *V* is definitely unassigned and is not definitely assigned.
  (The flow analysis rules prove that an assignment to *V* has not occurred.)
- *V* is not definitely assigned and is not definitely unassigned.
  (The rules cannot prove whether or not an assignment to *V* has occurred.)
- *V* is definitely assigned and is definitely unassigned.
  (It is impossible for the statement or expression to complete normally.)

To shorten the rules, the customary abbreviation “iff” is used to mean “if and only if”. We also use an abbreviation convention: if a rule contains one or more occurrences of “[un]assigned” then it stands for two rules, one with every occurrence of “[un]assigned” replaced by “definitely assigned” and one with every occurrence of “[un]assigned” replaced by “definitely unassigned”.

For example:

- *V* is [un]assigned after an empty statement iff it is [un]assigned before the empty statement.

should be understood to stand for two rules:

- *V* is definitely assigned after an empty statement iff it is definitely assigned before the empty statement.

V is definitely unassigned after an empty statement iff it is definitely unassigned before the empty statement.

The definite unassignment analysis of loop statements raises a special problem. Consider the statement `while (`*e*`)` *S*. In order to determine whether *V* is definitely unassigned within some subexpression of *e*, we need to determine whether *V* is definitely unassigned before *e*. One might argue, by analogy with the rule for definite assignment (§16.2.10), that *V* is definitely unassigned before *e* iff it is definitely unassigned before the `while` statement. However, such a rule is inadequate for our purposes. If *e* evaluates to true, the statement *S* will be executed. Later, if *V* is assigned by *S*, then in the following iteration(s) *V* will have already been assigned when *e* is evaluated. Under the rule suggested above, it would be possible to assign *V* multiple times, which is exactly what we have sought to avoid by introducing these rules.

A revised rule would be: "*V* is definitely unassigned before *e* iff it is definitely unassigned before the while statement and definitely unassigned after *S*". However, when we formulate the rule for *S*, we find: "*V* is definitely unassigned before *S* iff it is definitely unassigned after *e* when true". This leads to a circularity. In effect, *V* is definitely unassigned *before* the loop condition *e* only if it is unassigned *after* the loop as a whole!

We break this vicious circle using a *hypothetical* analysis of the loop condition and body. For example, if we assume that *V* is definitely unassigned before *e* (regardless of whether *V* really is definitely unassigned before *e*), and can then prove that *V* was definitely unassigned after *e* then we know that *e* does not assign *V*. This is stated more formally as:

Assuming *V* is definitely unassigned before *e*, *V* is definitely unassigned after *e*.

Variations on the above analysis are used to define well founded definite unassignment rules for all loop statements in the language.

Throughout the rest of this chapter, we will, unless explicitly stated otherwise, write *V* to represent a local variable or a blank `final` field (for rules of definite assignment) or a blank `final` variable (for rules of definite unassignment). Likewise, we will use *a*, *b*, *c*, and *e* to represent expressions, and *S* and *T* to represent statements. We will use the phrase *a* is *V* to mean that *a* is either the simple name of the *V*, or *V*'s simple name qualified by `this` (ignoring parentheses). We will use the phrase *a* is not *V* to mean the negation of *a* is *V*.

16.1 Definite Assignment and Expressions

Driftwood: The party of the first part shall be known in this contract as the party of the first part.
—Groucho Marx, *A Night at the Opera* (1935)

16.1.1 Boolean Constant Expressions

- *V* is [un]assigned after any constant expression whose value is `true` when false.
- *V* is [un]assigned after any constant expression whose value is `false` when true.

Because a constant expression whose value is `true` never has the value `false`, and a constant expression whose value is `false` never has the value `true`, the two preceding rules are vacuously satisfied. They are helpful in analyzing expressions involving the operators `&&` (§16.1.2), `||` (§16.1.3), `!` (§16.1.4), and `? :` (§16.1.5).

- *V* is [un]assigned after any constant expression whose value is `true` when true iff *V* is [un]assigned before the constant expression.
- *V* is [un]assigned after any constant expression whose value is `false` when false iff *V* is [un]assigned before the constant expression.
- *V* is [un]assigned after a boolean-valued constant expression *e* iff *V* is [un]assigned after *e* when true and *V* is [un]assigned after *e* when false. (This is equivalent to saying that *V* is [un]assigned after *e* iff *V* is [un]assigned before *e*.)

16.1.2 The Boolean Operator &&

- *V* is [un]assigned after *a* && *b* when true iff *V* is [un]assigned after *b* when true.
- *V* is [un]assigned after *a* && *b* when false iff *V* is [un]assigned after *a* when false and *V* is [un]assigned after *b* when false.
- *V* is [un]assigned before *a* iff *V* is [un]assigned before *a* && *b*.
- *V* is [un]assigned before *b* iff *V* is [un]assigned after *a* when true.
- *V* is [un]assigned after *a* && *b* iff *V* is [un]assigned after *a* && *b* when true and *V* is [un]assigned after *a* && *b* when false.

16.1.3 The Boolean Operator `||`

- *V* is [un]assigned after *a* `||` *b* when true iff *V* is [un]assigned after *a* when true and *V* is [un]assigned after *b* when true.
- *V* is [un]assigned after *a* `||` *b* when false iff *V* is [un]assigned after *b* when false.

- *V* is [un]assigned before *a* iff *V* is [un]assigned before *a* `||` *b*.
- *V* is [un]assigned before *b* iff *V* is [un]assigned after *a* when false.
- *V* is [un]assigned after *a* `||` *b* iff *V* is [un]assigned after *a* `||` *b* when true and *V* is [un]assigned after *a* `||` *b* when false.

16.1.4 The Boolean Operator `!`

- *V* is [un]assigned after `!`*a* when true iff *V* is [un]assigned after *a* when false.
- *V* is [un]assigned after `!`*a* when false iff *V* is [un]assigned after *a* when true.
- *V* is [un]assigned before *a* iff *V* is [un]assigned before `!`*a*.
- *V* is [un]assigned after `!`*a* iff *V* is [un]assigned after `!`*a* when true and *V* is [un]assigned after `!`*a* when false. (This is equivalent to saying that *V* is [un]assigned after `!`*a* iff *V* is [un]assigned after *a*.)

16.1.5 The Boolean Operator `? :`

Suppose that *b* and *c* are boolean-valued expressions.

- *V* is [un]assigned after *a* `?` *b* `:` *c* when true iff *V* is [un]assigned after *b* when true and *V* is [un]assigned after *c* when true.
- *V* is [un]assigned after *a* `?` *b* `:` *c* when false iff *V* is [un]assigned after *b* when false and *V* is [un]assigned after *c* when false.
- *V* is [un]assigned before *a* iff *V* is [un]assigned before *a* `?` *b* `:` *c*.
- *V* is [un]assigned before *b* iff *V* is [un]assigned after *a* when true.
- *V* is [un]assigned before *c* iff *V* is [un]assigned after *a* when false.
- *V* is [un]assigned after *a* `?` *b* `:` *c* iff *V* is [un]assigned after *a* `?` *b* `:` *c* when true and *V* is [un]assigned after *a* `?` *b* `:` *c* when false.

16.1.6 The Conditional Operator `? :`

Suppose that *b* and *c* are expressions that are not boolean-valued.

- *V* is [un]assigned after *a* `?` *b* `:` *c* iff *V* is [un]assigned after *b* and *V* is [un]assigned after *c*.
- *V* is [un]assigned before *a* iff *V* is [un]assigned before *a* `?` *b* `:` *c*.
- *V* is [un]assigned before *b* iff *V* is [un]assigned after *a* when true.

- *V* is [un]assigned before *c* iff *V* is [un]assigned after *a* when false.

16.1.7 Other Expressions of Type `boolean`

Suppose that *e* is a an expression of type boolean and is not a boolean constant expression, logical complement expression `!`*a*, conditional-and expression *a* `&&` *b*, conditional-or expression *a* `||` *b*, or conditional expression *a* `?` *b* `:` *c*.

- *V* is [un]assigned after *e* when true iff *V* is [un]assigned after *e*.
- *V* is [un]assigned after *e* when false iff *V* is [un]assigned after *e*.

16.1.8 Assignment Expressions

Driftwood: Would you like to hear it once more?
Fiorello: Just the first part.
Driftwood: What do you mean? The party of the first part?
Fiorello: No, the first part of the party of the first part.
—Groucho Marx and Chico Marx,
A Night at the Opera (1935)

Consider an assignment expression *a* `=` *b*, *a* `+=` *b*, *a* `-=` *b*, *a* `*=` *b*, *a* `/=` *b*, *a* `%=` *b*, *a* `<<=` *b*, *a* `>>=` *b*, *a* `>>>=` *b*, *a* `&=` *b*, *a* `|=` *b*, or *a* `^=` *b*.

- *V* is definitely assigned after the assignment expression iff either
  - *a* is *V* or
  - *V* is definitely assigned after *b*.
- *V* is definitely unassigned after the assignment expression iff *a* is not *V* and *V* is definitely unassigned after *b*.
- *V* is [un]assigned before *a* iff *V* is [un]assigned before the assignment expression.
- *V* is [un]assigned before *b* iff *V* is [un]assigned after *a*.

Note that if *a* is *V* and *V* is not definitely assigned before a compound assignment such as *a* `&=` *b*, then a compile-time error will necessarily occur. The first rule for definite assignment stated above includes the disjunct "*a* is *V*" even for compound assignment expressions, not just simple assignments, so that `V` will be considered to have been definitely assigned at later points in the code. Including the disjunct "*a* is `V`" does not affect the binary decision as to whether a program is acceptable or will result in a compile-time error, but it affects *how many* different points in the code may be regarded as erroneous, and so in practice it can improve the quality of error reporting. A similar remark applies to the inclusion of the conjunct "*a* is not `V`" in the first rule for definite unassignment stated above.

16.1.9 Operators ++ and --

- `V` is definitely assigned after ++*a*, --*a*, *a*++, or *a*-- iff either *a* is `V` or `V` is definitely assigned after the operand expression.
- `V` is definitely unassigned after ++*a*, --*a*, *a*++, or *a*-- iff *a* is not `V` and `V` is definitely unassigned after the operand expression.
- `V` is [un]assigned before *a* iff `V` is [un]assigned before ++*a*, --*a*, *a*++, or *a*--.

16.1.10 Other Expressions

Driftwood: All right. It says the, uh, the first part of the party of the first part, should be known in this contract as the first part of the party of the first part, should be known in this contract . . .
—Groucho Marx, *A Night at the Opera* (1935)

If an expression is not a boolean constant expression, and is not a preincrement expression ++*a*, predecrement expression --*a*, postincrement expression *a*++, postdecrement expression *a*--, logical complement expression !*a*, conditional-and expression *a* && *b*, conditional-or expression *a* || *b*, conditional expression *a* ? *b* : *c*, or assignment expression, then the following rules apply:

- If the expression has no subexpressions, `V` is [un]assigned after the expression iff `V` is [un]assigned before the expression. This case applies to literals, names, `this` (both qualified and unqualified), unqualified class instance creation expressions with no arguments, initialized array creation expressions whose initializers contain no expressions, unqualified superclass field access expressions, named method invocations with no arguments, and unqualified superclass method invocations with no arguments.
- If the expression has subexpressions, `V` is [un]assigned after the expression iff `V` is [un]assigned after its rightmost immediate subexpression.

There is a piece of subtle reasoning behind the assertion that a variable *V* can be known to be definitely unassigned after a method invocation. Taken by itself, at face value and without qualification, such an assertion is not always true, because an invoked method can perform assignments. But it must be remembered that, for the purposes of the Java programming language, the concept of definite unassignment is applied *only* to blank final variables. If *V* is a blank final local variable, then only the method to which its declaration belongs can perform assignments to *V*. If *V* is a blank final field, then only a constructor or an initializer for the class containing the declaration for *V* can perform assignments to *V*; no method can perform assignments to *V*. Finally, explicit constructor invocations (§8.8.5) are handled specially (§16.8); although they are syntactically similar to expression statements containing method invocations, they are not expression statements and therefore the rules of this section do not apply to explicit constructor invocations.

For any immediate subexpression *y* of an expression *x*, *V* is [un]assigned before *y* iff one of the following situations is true:

- *y* is the leftmost immediate subexpression of *x* and *V* is [un]assigned before *x*.
- *y* is the right-hand operand of a binary operator and *V* is [un]assigned after the left-hand operand.
- *x* is an array access, *y* is the subexpression within the brackets, and *V* is [un]assigned after the subexpression before the brackets.
- *x* is a primary method invocation expression, *y* is the first argument expression in the method invocation expression, and *V* is [un]assigned after the primary expression that computes the target object.
- *x* is a method invocation expression or a class instance creation expression; *y* is an argument expression, but not the first; and *V* is [un]assigned after the argument expression to the left of *y*.
- *x* is a qualified class instance creation expression, *y* is the first argument expression in the class instance creation expression, and *V* is [un]assigned after the primary expression that computes the qualifying object.
- *x* is an array instance creation expression; *y* is a dimension expression, but not the first; and *V* is [un]assigned after the dimension expression to the left of *y*.

- *x* is an array instance creation expression initialized via an array initializer; *y* is the array initializer in *x*; and *V* is [un]assigned after the dimension expression to the left of *y*.

16.2 Definite Assignment and Statements

Driftwood: The party of the second part shall be known in this contract as the party of the second part.
—Groucho Marx, *A Night at the Opera* (1935)

16.2.1 Empty Statements

- *V* is [un]assigned after an empty statement iff it is [un]assigned before the empty statement.

16.2.2 Blocks

- A blank final member field *V* is definitely assigned (and moreover is not definitely unassigned) before the block that is the body of any method in the scope of *V*.
- A local variable *V* is definitely unassigned (and moreover is not definitely assigned) before the block that is the body of the constructor, method, instance initializer or static initializer that declares *V*.
- Let *C* be a class declared within the scope of *V*. Then:
  - *V* is definitely assigned before the block that is the body of any constructor, method, instance initializer or static initializer declared in *C* iff *V* is definitely assigned before the declaration of *C*.

  Note that there are no rules that would allow us to conclude that *V* is definitely unassigned before the block that is the body of any constructor, method, instance initializer or static initializer declared in *C*. We can informally conclude that *V* is not definitely unassigned before the block that is the body of any constructor, method, instance initializer or static initializer declared in *C*, but there is no need for such a rule to be stated explicitly.
- *C* [un]assigned after an empty block iff it is [un]assigned before the empty block.
- *V* is [un]assigned after a nonempty block iff it is [un]assigned after the last statement in the block.

- *V* is [un]assigned before the first statement of the block iff it is [un]assigned before the block.
- *V* is [un]assigned before any other statement *S* of the block iff it is [un]assigned after the statement immediately preceding *S* in the block.

We say that *V* is definitely unassigned everywhere in a block *B* iff

- *V* is definitely unassigned before *B*.
- *V* is definitely assigned after *e* in every assignment expression *V* = *e*, *V* += *e*, *V* -= *e*, *V* *= *e*, *V* /= *e*, *V* %= *e*, *V* <<= *e*, *V* >>= *e*, *V* >>>= *e*, *V* &= *e*, *V* |= *e*, or *V* ^= *e* that occurs in *B*.
- *V* is definitely assigned before before every expression ++*V*, --*V*, *V*++, or *V*--. that occurs in *B*.

These conditions are counterintuitive and require some explanation. Consider a simple assignment V = e. If V is definitely assigned after e, then either:

1. The assignment occurs in dead code, and V is vacouusly definitely assigned. In this case, the assignment will not actually take place, and we can assume that V is not being assigned by the assignment expression.
2. V was already assigned by an earlier expression prior to e. In this case the current assignment will cause a compile-time error.

So, we can conclude that if the conditions are met by a program that causes no compile time error, then any assignments to V in B will not actually take place at run time.

16.2.3 Local Class Declaration Statements

- *V* is [un]assigned after a local class declaration statement iff it is [un]assigned before the local class declaration statement.

16.2.4 Local Variable Declaration Statements

- *V* is [un]assigned after a local variable declaration statement that contains no variable initializers iff it is [un]assigned before the local variable declaration statement.
- *V* is definitely assigned after a local variable declaration statement that contains at least one variable initializer iff either it is definitely assigned after the

last variable initializer in the local variable declaration statement or the last variable initializer in the declaration is in the declarator that declares *V*.

- *V* is definitely unassigned after a local variable declaration statement that contains at least one variable initializer iff it is definitely unassigned after the last variable initializer in the local variable declaration statement and the last variable initializer in the declaration is not in the declarator that declares *V*.
- *V* is [un]assigned before the first variable initializer in a local variable declaration statement iff it is [un]assigned before the local variable declaration statement.
- *V* is definitely assigned before any variable initializer *e* other than the first one in the local variable declaration statement iff either *V* is definitely assigned after the variable initializer to the left of *e* or the initializer expression to the left of *e* is in the declarator that declares *V*.
- *V* is definitely unassigned before any variable initializer *e* other than the first one in the local variable declaration statement iff *V* is definitely unassigned after the variable initializer to the left of *e* and the initializer expression to the left of *e* is not in the declarator that declares *V*.

16.2.5 Labeled Statements

- *V* is [un]assigned after a labeled statement `L:S` (where `L` is a label) iff *V* is [un]assigned after *S* and *V* is [un]assigned before every `break` statement that may exit the labeled statement `L:S`.
- *V* is [un]assigned before *S* iff *V* is [un]assigned before `L:S`.

16.2.6 Expression Statements

- *V* is [un]assigned after an expression statement *e*`;` iff it is [un]assigned after *e*.
- *V* is [un]assigned before *e* iff it is [un]assigned before *e*`;`.

16.2.7 `if` Statements

The following rules apply to a statement `if (`*e*`)` *S*:

- *V* is [un]assigned after `if (`*e*`)` *S* iff *V* is [un]assigned after *S* and *V* is [un]assigned after *e* when false.
- *V* is [un]assigned before *e* iff *V* is [un]assigned before `if (`*e*`)` *S*.

- *V* is [un]assigned before *S* iff *V* is [un]assigned after *e* when true.

The following rules apply to a statement `if (`*e*`)` *S* `else` *T*:

- *V* is [un]assigned after `if (`*e*`)` *S* `else` *T* iff *V* is [un]assigned after *S* and *V* is [un]assigned after *T*.
- *V* is [un]assigned before *e* iff *V* is [un]assigned before `if (`*e*`)` *S* `else` *T*.
- *V* is [un]assigned before *S* iff *V* is [un]assigned after *e* when true.
- *V* is [un]assigned before *T* iff *V* is [un]assigned after *e* when false.

16.2.8 `assert` Statements

The following rules apply both to a statement `assert` *e1* and to a statement `assert` *e1 :e2* :

- *V* is definitely [un]assigned before *e1* iff *V* is definitely [un]assigned before the `assert` statement.
- *V* is definitely assigned after the `assert` statement iff *V* is definitely assigned before the `assert` statement.
- *V* is definitely unassigned after the `assert` statement iff *V* is definitely unassigned before the `assert` statement and *V* is definitely unassigned after *e1* when true.

The following rule applies to a statement `assert` *e1: e2* :

- *V* is definitely [un]assigned before *e2* iff *V* is definitely [un]assigned after *e1* when false.

16.2.9 `switch` Statements

- *V* is [un]assigned after a `switch` statement iff all of the following are true:
  - Either there is a `default` label in the `switch` block or *V* is [un]assigned after the switch expression.
  - Either there are no switch labels in the `switch` block that do not begin a block-statement-group (that is, there are no switch labels immediately before the “`}`” that ends the switch block) or *V* is [un]assigned after the switch expression.

  - Either the `switch` block contains no block-statement-groups or *V* is [un]assigned after the last block-statement of the last block-statement-group.
  - *V* is [un]assigned before every `break` statement that may exit the `switch` statement.
- *V* is [un]assigned before the switch expression iff *V* is [un]assigned before the `switch` statement.

If a switch block contains at least one block-statement-group, then the following rules also apply:

- *V* is [un]assigned before the first block-statement of the first block-statement-group in the switch block iff *V* is [un]assigned after the switch expression.
- *V* is [un]assigned before the first block-statement of any block-statement-group other than the first iff *V* is [un]assigned after the switch expression and *V* is [un]assigned after the preceding block-statement.

16.2.10 `while` Statements

- *V* is [un]assigned after `while (`*e*`)` *S* iff *V* is [un]assigned after *e* when false and *V* is [un]assigned before every `break` statement for which the `while` statement is the break target.
- *V* is definitely assigned before *e* iff *V* is definitely assigned before the `while` statement.
- *V* is definitely unassigned before *e* iff all of the following conditions hold:
  - *V* is definitely unassigned before the `while` statement.
  - Assuming *V* is definitely unassigned before *e*, *V* is definitely unassigned after *S*.
  - Assuming *V* is definitely unassigned before *e*, *V* is definitely unassigned before every `continue` statement for which the `while` statement is the continue target.
- *V* is [un]assigned before *S* iff *V* is [un]assigned after *e* when true.

16.2.11 `do` Statements

- *V* is [un]assigned after `do` *S* `while` (*e*)`;` iff *V* is [un]assigned after *e* when false and *V* is [un]assigned before every `break` statement for which the `do` statement is the break target.
- *V* is definitely assigned before *S* iff *V* is definitely assigned before the `do` statement.
- *V* is definitely unassigned before *S* iff all of the following conditions hold:
  - *V* is definitely unassigned before the `do` statement.
  - Assuming *V* is definitely unassigned before *S*, *V* is definitely unassigned after *e* when true.
- *V* is [un]assigned before *e* iff *V* is [un]assigned after *S* and *V* is [un]assigned before every `continue` statement for which the `do` statement is the continue target.

16.2.12 `for` Statements

- *V* is [un]assigned after a `for` statement iff both of the following are true:
  - Either a condition expression is not present or *V* is [un]assigned after the condition expression when false.
  - *V* is [un]assigned before every `break` statement for which the `for` statement is the break target.
- *V* is [un]assigned before the initialization part of the `for` statement iff *V* is [un]assigned before the `for` statement.
- *V* is definitely assigned before the condition part of the `for` statement iff *V* is definitely assigned after the initialization part of the `for` statement.
- *V* is definitely unassigned before the condition part of the `for` statement iff all of the following conditions hold:
  - *V* is definitely unassigned after the initialization part of the `for` statement.
  - Assuming *V* is definitely unassigned before the condition part of the `for` statement, *V* is definitely unassigned after the contained statement.
  - Assuming *V* is definitely unassigned before the contained statement, *V* is definitely unassigned before every `continue` statement for which the `for` statement is the continue target.

- *V* is [un]assigned before the contained statement iff either of the following is true:
  - A condition expression is present and *V* is [un]assigned after the condition expression when true.
  - No condition expression is present and *V* is [un]assigned after the initialization part of the `for` statement.
- *V* is [un]assigned before the incrementation part of the `for` statement iff *V* is [un]assigned after the contained statement and *V* is [un]assigned before every `continue` statement for which the `for` statement is the continue target.

16.2.12.1 *Initialization Part*

- If the initialization part of the `for` statement is a local variable declaration statement, the rules of §16.2.4 apply.
- Otherwise, if the initialization part is empty, then *V* is [un]assigned after the initialization part iff *V* is [un]assigned before the initialization part.
- Otherwise, three rules apply:
  - *V* is [un]assigned after the initialization part iff *V* is [un]assigned after the last expression statement in the initialization part.
  - *V* is [un]assigned before the first expression statement in the initialization part iff *V* is [un]assigned before the initialization part.
  - *V* is [un]assigned before an expression statement *E* other than the first in the initialization part iff *V* is [un]assigned after the expression statement immediately preceding *E*.

16.2.12.2 *Incrementation Part*

- If the incrementation part of the `for` statement is empty, then *V* is [un]assigned after the incrementation part iff *V* is [un]assigned before the incrementation part.
- Otherwise, three rules apply:
  - *V* is [un]assigned after the incrementation part iff *V* is [un]assigned after the last expression statement in the incrementation part.
  - *V* is [un]assigned before the first expression statement in the incrementation part iff *V* is [un]assigned before the incrementation part.

- *V* is [un]assigned before an expression statement *E* other than the first in the incrementation part iff *V* is [un]assigned after the expression statement immediately preceding *E*.

16.2.13 `break`, `continue`, `return`, and `throw` Statements

Fiorello: Hey, look! Why can't the first part of the second party be the second part of the first party? Then you've got something!

—Chico Marx, *A Night at the Opera* (1935)

- By convention, we say that *V* is [un]assigned after any `break`, `continue`, `return`, or `throw` statement. The notion that a variable is "[un]assigned after" a statement or expression really means "is [un]assigned after the statement or expression completes normally". Because a `break`, `continue`, `return`, or `throw` statement never completes normally, it vacuously satisfies this notion.
- In a `return` statement with an expression *e* or a `throw` statement with an expression *e*, *V* is [un]assigned before *e* iff *V* is [un]assigned before the `return` or `throw` statement.

16.2.14 `synchronized` Statements

- *V* is [un]assigned after `synchronized (`*e*`)` *S* iff *V* is [un]assigned after *S*.
- *V* is [un]assigned before *e* iff *V* is [un]assigned before the statement `synchronized (`*e*`)` *S*.
- *V* is [un]assigned before *S* iff *V* is [un]assigned after *e*.

16.2.15 `try` Statements

These rules apply to every `try` statement, whether or not it has a `finally` block:

- *V* is [un]assigned before the `try` block iff *V* is [un]assigned before the `try` statement.
- *V* is definitely assigned before a `catch` block iff *V* is definitely assigned before the `try` block.
- *V* is definitely unassigned before a `catch` block iff all of the following conditions hold:
  - *V* is definitely unassigned after the `try` block.

- ◆ *V* is definitely unassigned before every `return` statement that belongs to the `try` block.
- ◆ *V* is definitely unassigned after *e* in every statement of the form `throw` *e* that belongs to the `try` block.
- ◆ *V* is definitely unassigned after *e1* for every statement of the form `assert` *e1*, that occurs in the try block.
- ◆ *V* is definitely unassigned after *e2* in every statement of the form `assert` *e1* : *e2* that occurs in the try block.
- ◆ *V* is definitely unassigned before every `break` statement that belongs to the `try` block and whose break target contains (or is) the `try` statement.
- ◆ *V* is definitely unassigned before every `continue` statement that belongs to the `try` block and whose continue target contains the `try` statement.

If a `try` statement does not have a `finally` block, then this rule also applies:

- *V* is [un]assigned after the `try` statement iff *V* is [un]assigned after the `try` block and *V* is [un]assigned after every `catch` block in the try statement.

If a `try` statement does have a `finally` block, then these rules also apply:

- *V* is definitely assigned after the `try` statement iff at least one of the following is true:
  - ◆ *V* is definitely assigned after the try block and *V* is definitely assigned after every `catch` block in the try statement.
  - ◆ *V* is definitely assigned after the `finally` block.
  - ◆ *V* is definitely unassigned after a `try` statement iff *V* is definitely unassigned after the `finally` block.
- *V* is definitely assigned before the `finally` block iff *V* is definitely assigned before the `try` statement.
- *V* is definitely unassigned before the `finally` block iff all of the following conditions hold:
  - ◆ *V* is definitely unassigned after the `try` block.
  - ◆ *V* is definitely unassigned before every `return` statement that belongs to the `try` block.
  - ◆ *V* is definitely unassigned after *e* in before every statement of the form `throw` *e* that belongs to the `try` block.

  - *V* is definitely unassigned after *e1* for every statement of the form `assert` *e1*, that occurs in the try block.
  - *V* is definitely unassigned after *e2* in every statement of the form `assert` *e1* : *e2* that occurs in the try block.
  - *V* is definitely unassigned before every `break` statement that belongs to the `try` block and whose break target contains (or is) the `try` statement.
  - *V* is definitely unassigned before every `continue` statement that belongs to the `try` block and whose continue target contains the `try` statement.
  - *V* is definitely unassigned after every `catch` block of the `try` statement.

16.3 Definite Assignment and Parameters

- A formal parameter *V* of a method or constructor is definitely assigned (and moreover is not definitely unassigned) before the body of the method or constructor.
- An exception parameter *V* of a `catch` clause is definitely assigned (and moreover is not definitely unassigned) before the body of the `catch` clause.

16.4 Definite Assignment and Array Initializers

What about assignments within the array initializer? And how does this change when we have COALs?

- *V* is [un]assigned after an empty array initializer iff it is [un]assigned before the empty array initializer.
- *V* is [un]assigned after a nonempty array initializer iff it is [un]assigned after the last variable initializer in the array initializer.
- *V* is [un]assigned before the first variable initializer of the array initializer iff it is [un]assigned before the array initializer.
- *V* is [un]assigned before any other variable initializer *I* of the array initializer iff it is [un]assigned after the variable initializer to the left of *I* in the array initializer.

16.5 Definite Assignment and Anonymous Classes

- *V* is definitely assigned before an anonymous class declaration (§15.9.5) that is declared within the scope of *V* iff *V* is definitely assigned after the class instance creation expression that declares the anonymous class.

16.6 Definite Assignment and Member Types

Let *C* be a class, and let *V* be a blank final member field of *C*. Then:

- *V* is definitely assigned (and moreover, not definitely unassigned) before the declaration of any member type of *C*.

Let *C* be a class declared within the scope of *V*. Then:

- *V* is definitely assigned before a member type (§8.5, §9.5) declaration of *C* iff *V* is definitely assigned before the declaration of *C*.

16.7 Definite Assignment and Static Initializers

Let *C* be a class declared within the scope of *V*. Then:

- *V* is definitely assigned before a static variable initializer of *C* iff *V* is definitely assigned before the declaration of *C*.

Note that there are no rules that would allow us to conclude that *V* is definitely unassigned before a static variable initializer. We can informally conclude that *V* is not definitely unassigned before any static variable initializer of *C*, but there is no need for such a rule to be stated explicitly.

Let *C* be a class, and let *V* be a blank `final static` member field of *C*, declared in *C*. Then:

- *V* is definitely unassigned (and moreover is not definitely assigned) before the leftmost `static` initializer or `static` variable initializer of *C*.
- *V* is [un]assigned before a `static` initializer or `static` variable initializer of *C* other than the leftmost iff *V* is [un]assigned after the preceding `static` initializer or `static` variable initializer of *C*.

Let *C* be a class, and let *V* be a blank `final static` member field of *C*, declared in a superclass of *C*. Then:

- `V` is definitely assigned (and moreover is not definitely unassigned) before the block that is the body of a static initializer of `C`.
- `V` is definitely assigned (and moreover is not definitely unassigned) before every static variable initializer of `C`.

16.8 Definite Assignment, Constructors, and Instance Initializers

Let `C` be a class declared within the scope of `V`. Then:

- `V` is definitely assigned before an instance variable initializer of `C` iff `V` is definitely assigned before the declaration of `C`.

Note that there are no rules that would allow us to conclude that `V` is definitely unassigned before an instance variable initializer. We can informally conclude that `V` is not definitely unassigned before any instance variable initializer of `C`, but there is no need for such a rule to be stated explicitly.

Let `C` be a class, and let `V` be a blank `final` non-`static` member field of `C`, declared in `C`. Then:

- `V` is definitely unassigned (and moreover is not definitely assigned) before the leftmost instance initializer or instance variable initializer of `C`.
- `V` is [un]assigned before an instance initializer or instance variable initializer of `C` other than the leftmost iff `V` is [un]assigned after the preceding instance initializer or instance variable initializer of `C`.

The following rules hold within the constructors of class `C`:

- `V` is definitely assigned (and moreover is not definitely unassigned) after an alternate constructor invocation (§8.8.7.1).
- `V` is definitely unassigned (and moreover is not definitely assigned) before an explicit or implicit superclass constructor invocation (§8.8.7.1).
- If `C` has no instance initializers or instance variable initializers, then `V` is not definitely assigned (and moreover is definitely unassigned) after an explicit or implicit superclass constructor invocation.
- If `C` has at least one instance initializer or instance variable initializer then `V` is [un]assigned after an explicit or implicit superclass constructor invocation iff `V` is [un]assigned after the rightmost instance initializer or instance variable initializer of `C`.

Let *C* be a class, and let *V* be a blank `final` member field of *C*, declared in a superclass of *C*. Then:

- *V* is definitely assigned (and moreover is not definitely unassigned) before the block that is the body of a constructor, or instance initializer of *C*.
- *V* is definitely assigned (and moreover is not definitely unassigned) before every instance variable initializer of *C*.