**Wavefront Application**

For the purpose defining wavefront application, the initialization of a temporary object is considered a subexpression of the expression that necessitates the temporary object.

An evaluation C of an expression is *sequenced-between* two evaluations A and B (of possibly different expressions) if A is sequenced before C and C is sequenced before B.

An execution of a statement S *contains* an evaluation A of an expression E if evaluation of A occurs on behalf of S or a substatement of S. [*Note*: Expression E might be in the invocation of a function while S is executing.]

*Vertical antecedent* is an irreflexive, nonsymmetric, nontransitive relationship between two evaluations. For evaluation A and a different evaluation B (of possibly the same expression), both within the same invocation of an element access function, evaluation A is a *vertical antecedent* of evaluation B if:

* evaluation A is sequenced before evaluation B, and
* if there are evaluations sequenced between A and B, there exists a statement execution S or expression evaluation that:
  + contains A, and
  + contains all said evaluations between A and B,
  + but does not contain B, and
* control reached B from A without executing any of the following:
  + a goto statement executed within execution S that jumps to a statement outside of S, or
  + a switch statement executed within S that transfers control into a substatement of a nested selection or iteration statement, or
  + a longjmp.

The first and second major bullets above describes the notion related to “immediately precedes”. If A and B are part of the *same* statement, then A is a vertical antecedent of B only if there is nothing sequenced between them. If A and B are part of *different* statements, then A is a vertical antecedent of B if, by popping out zero or more levels of nesting, you find a point where the statement containing A immediately precedes B. This is the point of re-convergence after a control-flow divergence.

The third major bullet is needed to handle cases where re-convergence is difficult or impossible to establish. In those cases, the guarantees degenerate to those provided by the unsequenced\_execution\_policy until convergence is re-established at the end of the block containing both the jump statement and the jumped-to statement.

Each invocation of an element access function performed by an algorithm with an input range is associated with a value in that range, called its *context*.  *Horizontal antecedent* is an irreflexive, nonsymmetric, transitive relationship between two evaluations with different contexts. Within two applications of an element access function, given evaluations B and B′ of an expression X, B is a *horizontal antecedent* of B′ if:

* Context of B precedes context of B′ and neither evaluation has a vertical antecedent, or
* There exist evaluations A and A’ of a (possibly different) expression Y that are the vertical antecedents of evaluations B and B′ respectively, and A is a horizontal antecedent of A’.

[*Note:* Horizontal antecedent does not imply a sequenced-before relationship. --  *end note*]

The rules for establishing the horizontal antecedent relationship match evaluations in one application with corresponding evaluations in a logically-later application of the element access function. The nature of the rules are such that even nested loops work correctly. For example, given[[1]](#footnote-2):

*b*;  
 while ( *e* )  
 *stmt*;  
 *c*;

where *b* is a horizontal antecedent of *b’*. Intuitively, we would expect the kth evaluation *e* to be the horizontal antecedent of the kth evaluation of *e*′, assuming both evaluations happen. Even if one of the invocations executes *e* more times than the other, the last evaluations of *e* and *e*’ that is common to both invocations are vertical antecedents of *c* and *c*’, so the horizontal antecedent relationship is re-established for *c* and *c*’.

Let *f* be a function called for each argument list in a sequence of argument lists. Let A and B′ denote two evaluations. *Wavefront application* of *f* requires that A is sequenced before B′ if:

* There exists an evaluation B such that A is sequenced before B and B is a horizontal antecedent of B′, or
* There exists an evaluation A′ such that A′ is sequenced before B′ and A is a horizontal antecedent of A′.

The two bullets describe the two triangles in Figure 1. Note that the vertical relationships are *sequenced before*, not *vertical antecedent.*

A

A′

B

B′

′

**Figure 1**

**Optional clause for supporting scatter pattern**

The *direct side effects* of a an expression X are those caused by evaluating X, but not including side effects caused by evaluating its sub-expressions. For any two evaluations A and A′ such that A is a horizontal antecedent of A′, all direct side effects in A are sequenced before all direct side effects in A′.

This clause allows for code such as:

A[B[i]] = expr(i);

to produce deterministic results even if B[i] contains duplicate elements (sometimes called the *overlapping scatter pattern*). If this clause is adopted, we will also want a library function, unordered\_update, having a syntax similar to ordered\_update, that relaxes this guarantee and allows the generation of faster code on architectures with scatter instructions that do not support ordered writes. The unordered\_update function should be used only when B[i] is known not to contain duplicates.

1. The notation here is informal. In the code, the letters *b, c, e* denote expressions. In the discussion text they refer to the evaluations of those expressions. [↑](#footnote-ref-2)