**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.]

*Horizontally matched* is an equivalence relationship between two evaluations. Within two invocations of an element access function F, given evaluations A and A′ of an expression X, *horizontally matched* is established by the following rules:

If there are no evaluations sequenced before A in the corresponding invocation of F, and likewise for A’, then A is horizontally matched with A’.

Given evaluations A and A’ of expression X, where A is horizontally matched with A’, and evaluations B and B’ of a (possibly different) expression Y, then B is horizontally matched with 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.
* and similar conditions hold for A′, B′, and S′. [*Note*: S and S’ are possibly different executions of the same statement.]

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.

The rules for establishing the horizontal match relationship matches evaluations in one application with corresponding evaluations in another 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* horizontally matched with *b’*. Intuitively, we would expect the kth evaluation *e* to be horizontally matched with 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 are common to both invocations cause a horizontal antecedent relationship to be re-established for *c* and *c*’.

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*. Letf and f′ denote the invocations of a function *F* in two contexts, where the context of f precedes the context for f′. Let A and B′ denote evaluations performed within distinct calls to f and f′ respectively. *Wavefront application* of *F* requires that A be sequenced before B′ if:

* If there exists evaluation A′ in invocation f′ and A is horizontally matched with A’ and A′ is sequenced before B′, or
* If there exists evaluation B in invocation f, and B is horizontally matched with B’ and A’ is sequenced before B’.

The two bullets describe the two triangles in Figure 1. Rule 1 can be summarized graphically as shown in Figure 1.

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 horizontally matched evaluations A and A′ in f and f′ respectively, 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 ordered\_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)