AquaLang: A Dataflow Programming Language (Appendix)

A INCREMENTALISATION

In this appendix, we describe a step-by-step example of incrementalisation in AquaLang.

Step 0. Initial program:

```
def stddev(bids: Vec[Bid]): f64 = {
  var sum = 0;
  var count = 0;
  for bid in bids {
      sum += bid.price;
      count += 1;
  }
  val mean = sum/count;
  var sumSqDiff = 0.0;
  for bid in bids {
      sumSqDiff += (bid.price - mean).pow(2);
  }
  val variance = sumSqDiff/count;
  variance.sqrt()
}

source(...).window(sliding(...), stddev).sink(...).run(...);
```

Step 1. Rewrite loops into folds, where the loop-carried variables become accumulators.

```
def stddev(bids: Vec[Bid]): f64 = {
  val (sum, count) = bids.fold(
      (0,0),
      fun((sum, count), bid) =
          (sum + bid.price, count + 1));
  val mean = sum/count;
  val sumSqDiff = bids.fold(
      0.0,
      fun(sumSqDiff, bid) =
          sumSqDiff + (bid.price - mean).pow(2));
  val variance = sumSqDiff/count;
  variance.sqrt()
}
```

Step 2. Rewrite the fold over pairs into a pair of folds:

```
val sum = bids.fold(0, fun(sum, bid) = sum + bid.price);
val count = bids.fold(0, fun(count, bid) = count + 1);
```

Step 3. Expand the power operation.

```
val sumSqDiff = bids.fold(0.0, fun(sumSqDiff, bid) =
  sumSqDiff
  + bid.price.pow(2) - 2*bid.price*mean + mean.pow(2));
```

Step 4. A fold of sums can be rewritten into a sum of folds. This is allowed since addition is associative (i.e., $a+(b+c) = (a+b)+c$) and commutative (i.e., $a+b = b+a$) such that $a_0+b_0+a_1+b_1+\cdots+a_n+b_n = (a_0+a_1+\cdots+a_n)+(b_0+b_1+\cdots+b_n)$.

```
val sumSq = bids.fold(0.0,
  fun(sumSq, bid) = sumSq + bid.price.pow(2));
val diff2Mean = bids.fold(0.0,
  fun(diff2Mean, bid) = diff2Mean - 2*bid.price*mean);
val sumMeanSq = bids.fold(0.0,
  fun(sumMeanSq, bid) = sumMeanSq + mean.pow(2));
val sumSqDiff = sumSq + diff2Mean + sumMeanSq;
```

Step 5. Extract the subtraction from the `diff2Mean` fold and leave an addition in its place. This is legal since addition is semi-associativie with respect to subtraction (i.e., $a-b-c = a-(b+c)$) and $-0 = 0$, such that $0-x_0-x_1-\cdots-x_n = -(0+x_0+x_1+\cdots+x_n)$.

```
val diff2Mean = -bids.fold(0.0,
  fun(diff2Mean, bid) = diff2Mean + 2*mean*bid.price);
```

Step 6. Extract the constant factor from the sum inside the `diff2Mean` and `sumMeanSq` folds. This is allowed since multiplication is distributive over addition (i.e., $ca+cb = c(a+b)$) and $c\cdot 0 = 0$, such that $0+cx0+cx_1+\cdots+cx_n = c(0+x0+x_1+\cdots+x_n)$.

```
val diff2Mean = -2*mean*bids.fold(0.0,
  fun(diff2Mean, bid) = diff2Mean + bid.price);
val sumMeanSq = mean.pow(2)*bids.fold(0.0,
  fun(sumMeanSq, bid) = sumMeanSq + 1);
```

Step 7. Simplify `diff2Mean` and `sumMeanSq` by factoring their folds to `sum` and `count` respectively.

```
val diff2Mean = -2*mean*sum;
val sumMeanSq = mean.pow(2)*count;
```

Step 8. Expand the mean:

```
val diff2Mean = -2*(sum/count)*sum;
val sumMeanSq = (sum/count).pow(2) * count;
```

Step 9. Simplify `diff2Mean` and `sumMeanSq` (assuming `count` is non-zero, which is true for all windows):

```
val diff2Mean = -2 * sum.pow(2)/count;
val sumMeanSq = sum.pow(2)/count;
```

Step 10. Substitute `diff2Mean` and `sumMeanSq`:

```
val sumSqDiff =
    sumSq - 2*sum.pow(2)/count + sum.pow(2)/count;
```

Step 11. Simplify `sumSqDiff`.

```
val sumSqDiff = sumSq - sum.pow(2)/count;
```

Step 12. Substitute `sumSqDiff`:

```
val variance = (sumSq - sum.pow(2)/count)/count;
```

Step 13. Simplify `variance`:

```
val variance = sumSq/count - (sum/count).pow(2);
```

Step 14. Substitute the mean through syntactic equality:

```
val variance = sumSq/count - mean.pow(2);
```

Step 15. Rewrite the sum of floats `sumSq` to be a sum of integers that is then cast to a float. This is made possible since `bid.price.pow(2)` is an integer.

```
val sumSq = bids.fold(0,
  fun(sumSq, bid) = sumSq + bid.price.pow(2)).toF64();
```

At this point, we have the program:

```
def stddev(bids: Vec[Bid]): f64 {
  val sum = bids.fold(0,
    fun(sum, bid) = sum + bid.price);
  val count = bids.fold(0,
    fun(count, bid) = count + 1);
  val mean = sum/count;
  val sumSq = bids.fold(0,
    fun(sumSq, bid) = sumSq + bid.price.pow(2)).toF64();
  val variance = sumSq/count - mean.pow(2);
  variance.sqrt()
}
```

Step 16. Rewrite each `fold` into a `foldMap`:

```
val sum = bids.foldMap(0,
    fun(bid) = bid.price,
    fun(sum1, sum2) = sum1 + sum2,
    fun(sum) = sum
);
val count = bids.foldMap(0,
    fun(bid) = 1,
    fun(count1, count2) = count1 + count2,
    fun(count) = count
);
val sumSq = bids.foldMap(0,
    fun(bid) = bid.price.pow(2),
    fun(sumSq1, sumSq2) = sumSq1 + sumSq2,
    fun(sumSq) = sumSq.toF64(),
);
```

Step 17. Fuse the `foldMaps` into a single `foldMap`:

```
val (sum, count, sumSq) =
  bids.foldMap(
    (0, 0, 0),
    fun(bid) =
        (bid.price, 1, bid.price.pow(2)),
    fun((sum1, count1, sumSq1), (sum2, count2, sumSq2)) =
        (sum1+sum2, count1+count2, sumSq1+sumSq2),
    fun((sum, count, sumSq)) =
        (sum, count, sumSq.toF64()));
```

Step 18. Inline `mean`, `variance` and `stddev` into the lower function of `foldMap`:

```
def stddev(bids: Vec[Bid]): f64 =
  bids.foldMap(
    (0, 0, 0),
    fun(bid) = (bid.price, 1, bid.price.pow(2)),
    fun((sum1,count1,sumSq1), (sum2,count2,sumSq2)) =
        (sum1+sum2, count1+count2, sumSq1+sumSq2),
    fun((sum,count,sumSq1)) = {
        val mean = sum/count;
        val variance = sumSq.toF64()/count - mean.pow(2);
        variance.sqrt()
    })
```

Step 19. Rewrite `window` into `incrWindow`:

```
source(...)
  .incrWindow(sliding(...),
    (0, 0, 0),
    fun(bid) = (bid.price, 1, bid.price.pow(2)),
    fun((sum1,count1,sumSq1), (sum2,count2,sumSq2)) =
        (sum1+sum2, count1+count2, sumSq1+sumSq2),
    fun((sum,count,sumSq)) = {
        val mean = sum/count;
        val variance = sumSq.toF64()/count - mean.pow(2);
        variance.sqrt() })
  .sink(...).run(...);
```