Adding Probabilities in the Hakaru to C Compiler (HKC)

Zach Sullivan

July 30, 2016

Probabilities in Hakaru

Because Hakaru is a probabilistic language, we provide a type just for probabilities. The type `prob` has extra safety from underflow. The main way we accomplish this is by storing its value as double precision floating point numbers in the log-domain. We can do basic arithmetic on our probabilities.

The "LogSumExp Trick"

Because our probability types are stored in the log-domain, we need to compute

$$\mathrm{LSE}(x) = \log\left(\sum_{i=0}^{n} e^{x_i}\right)$$

often called "LogSumExp." The advantage of storing probabilities in the log-domain is lost if we simply remove them from the log-domain before doing calculations on them. We would also add a bunch of `log` and `exp` operations.

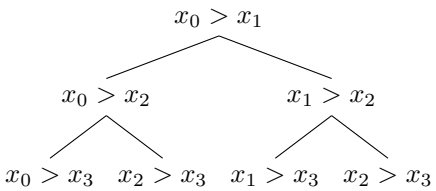
$$\mathrm{LSE}'(x) = \hat{x} + \log\left(\sum_{i=0}^{n} e^{x_i - \hat{x}}\right)$$
$$\textbf{where } \hat{x} = \max(x)$$

This trick preserves the value of the largest of the numbers being summed.

Max Comparison Tree

LogSumExp safety creates a particular challenge when generating code for our compiler. HKC compiles Hakaru to C, where we have no max function that works on an arbitrary number of arguments. The solution is to create a tree of comparisons using C's ternary conditional expression to find the maximum of n number of arguments.

Here is an example of a max comparison tree when the length of array x is 4.



Code Generation

After generating a max comparison tree, we can create leaves for our tree that are different LogSumExp summations. This is rather trivial given a particular max index.

A Hakaru program that sums 4 probabilities will generate the following C expression, where `p_a`, `p_b`, `p_c`, and `p_d` are variables holding probabilities:

```
p_a > p_b
  ? p_a > p_c
    ? p_a > p_d
      ? p_a + log1p(expm1(p_c - p_a) + (expm1(p_d - p_a) + expm1(p_b - p_a)) + 3)
      : p_d + log1p(expm1(p_b - p_d) + (expm1(p_c - p_d) + expm1(p_a - p_d)) + 3)
    : ( p_c > p_d
      ? p_c + log1p(expm1(p_b - p_c) + (expm1(p_d - p_c) + expm1(p_a - p_c)) + 3)
      : p_d + log1p(expm1(p_b - p_d) + (expm1(p_c - p_d) + expm1(p_a - p_d)) + 3))
  : (p_b > p_c
    ? p_b > p_d
      ? p_b + log1p(expm1(p_c - p_b) + (expm1(p_d - p_b) + expm1(p_a - p_b)) + 3)
      : p_d + log1p(expm1(p_b - p_d) + (expm1(p_c - p_d) + expm1(p_a - p_d)) + 3)
    : (p_c > p_d
      ? p_c + log1p(expm1(p_b - p_c) + (expm1(p_d - p_c) + expm1(p_a - p_c)) + 3)
      : p_d + log1p(expm1(p_b - p_d) + (expm1(p_c - p_d) + expm1(p_a - p_d)) + 3)));
```

We can use `log1p` because where x_i is the maximum $e^{x_i - \hat{x}} = 1$. We use `log1p` and `expm1` because they can be more accurate for small values.

Future Improvements

The LogSumExp code generation will be the same for each n number of arguments. If we have several LogSumExp operations of size n in the same program, then we will be generating the same code multiple times. Creating different LogSumExp C functions for different numbers of arguments will reduce code size.

Kahan summation is another improvement that keeps track of the accumulated error in each addition, which would add more robustness to our operations.