CSCC24 E6 - C++ $\lambda, map, filter, fold$

Sean Shekhtman

Due: April 9th, 2024

1 Introduction

C++ is a superset of the C programming language, a language that notably does not support lambda expressions. Furthermore, C doesn't provide libraries for *map*, *filter*, or *fold*, however they are easily implementable. (See map.c for an example implementation.)

Upon moving onto the world of C++, we will see that lambda expressions have more optional fields than we have commonly used in CSCC24, *map*, *filter*, *fold* can be considered as special cases of std::transform(), std::copy_if(), std::accumulate(), respectively.

There is a notable exception however, for converting an accumulate() function, which is a general *foldl* function, into a *foldr* function.

2 Lambda Expressions

Lambda expressions in C++ have the following format:
[CAPTURE_CLAUSE] (PARAMETER_LIST) $\rightarrow$ RETURN_TYPE {LAMBDA_BODY}

Note that I will be ignoring optional parameters that are unimportant to the discussion such as the *mutable specification* and *exception-specification*.

The *capture clause* represents a closure. (i.e., every variable in the capture clause is *captured* in the closure and will be dependent on the values of the variables at the time of the creation of the lambda.) Note that this must be done by passing variables in the *capture clause* by value (as opposed to pass-by-reference.) Passing variables by value and by reference is denoted by [=], and [&], respectively.
If the '&' is not specified before a variable name, then it is assumed to be passed-by-value (i.e., closed.)

The *parameter list* is equivalent to the parameter list in Haskell, with the exception that the inputs are not automatically put in curried form. (e.g., [](int

x, int y) → { return "blah"; }; is of the form: (x,y) -→ z). A form of *pseudo-currying* is still possible manually, though one would have to implement the partially-applied lambda expressions themselves.

3 Map

A form of *map* is easily implementable in C++ using the std::transform() function from the *algorithm* library. The function has the following format:
transform(InputIt first1, InputIt last1, OutputIt d_first, UnaryOp unary_op);
where 'first1' is the start of the iterable, 'last1' is the end of the iterable, 'd_first' is the start of the iterable to be outputted to, and 'unary_op' is some unary function.

To conform to the *map* we've learnt in CSCC24, we always set first1 and last1 to the first and last element of the iterable, respectively. last1 is arbitrary as it merely specifies where to place the result of transform(). This will be the case for all instances of 'first', 'last', 'first1', 'last1' in all future functions mentioned as well. 'unary_op' is equivalent to the function 'f' used in *map*. (There isn't much explaining necessary here.)

4 Filter

Similarly to *map*, *filter* is also easily implementable in C++ using the std::copy_if() function from the *algorithm* library. The function has the following format:
copy_if(InputIt first, InputIt last, OutputIt d_first, UnaryPred pred);
where 'first', 'last', 'd_first' are identical to 'first1', 'last1', 'd_first' in std::transform() as described earlier, respectively.

The parameter responsible for the filtering in copy_if is 'pred', a unary predicate, which determines whether the current element of the iterable will be copied to 'd_first'.

5 Fold

In contrast to *map* and *fold*, *fold* has the most interesting implementation details. *fold* is comparable to the C++ function std::accumulate() from the numerics library. The function has the following format:
accumulate(InputIt first, InputIt last, T init, BinaryOperation op);
where 'first', 'last' are identical to 'first1', 'last1' in std::transform(), respectively.

The 'init' parameter is equivalent to the *fold* 'id' in Haskell (i.e., it specifies an initial value to be used to fold/reduce the elements of some iterable.) Unsurprisingly, 'op' is the binary operation used to reduce the iterable down to some

value, as with ‘op‘ in Haskell’s implementation of *fold*.

accumulate() is equivalent to *foldl* by default, however, there exists a method to transform this function into foldr. The method, as you will see, is *very* interesting.

First, we must iterate through the iterable in reverse order. This is achieved by employing two reverse iterator functions, namely std::rbegin() and std::rend(), which return a reverse iterator to the beginning and end of an iterable, respectively.

Second, we replace the ‘init‘ paramter with what I refer to as a *reverse init*. For example, in the case of non-associative arithmetic operations such as subtraction in which *foldl* and *foldr* may yield different outputs, a suitable reverse init would be the second-last element of the iterable. This also has the added effect of having to do one less evaluation since we’re replacing the identity with an element from the iterable. Interestingly, I’m unsure as to whether this choice of reverse init will work for all cases.
As this will require a great deal of investigation (and possibly a proof), I will omit any analysis for a general choice of reverse init. (Trust me, I’ve pondered long and hard. Although, it’s a very interesting question to me!)

Lastly, the binary operation ‘op‘ must be evaluated in reverse. This is done due to the difference in the order of evaluation of *foldl* and *foldr* (i.e., given a binary operation op(x,y), foldl will evaluate from left-to-right x ‘op‘ y, while foldr will evaluate from right-to-left y ‘op‘ x.) This would mean that one must modify the binary operation ‘op‘, such that it is evaluated from right-to-left.

6 Conclusion

In most cases, C++ allows for λ, *map*, *filter*, *fold* to be implemented with relative ease, although at times certain parameters and/or functions must be adjusted, as seen by the sections on *lambdas* and*fold*. Indeed, for lambda expressions, partial evaluation is impossible by default and one must create the partially-applied functions oneself, and for *fold*, one must directly modify all of the parameters of std::accumulate() as well as create a ”reversed” binary function ‘op‘.

It is often the case that the C++ functions gave *more* control (in a sense), than say, Haskell’s implementations of these functions. For instance, nearly all of the functions allow the programmer to specify the start and end ”index” to apply some operation (recall ‘first‘, ‘last‘, ‘first1‘ in the previous sections.)

Nonetheless, it has been an interesting and intellectually stimulating journey going through how C++ implements the functions discussed in this paper.

I hope, dear reader, from the bottom of my heart, that you too enjoyed exploring the differences in implementation details between C++ and Haskell for the aforementioned functions.

Thank you for reading this paper until the end!

7 References

https://learn.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=msvc-170
https://en.cppreference.com/w/cpp/algorithm/transform
https://en.cppreference.com/w/cpp/algorithm/copy
https://en.cppreference.com/w/cpp/algorithm/accumulate
https://en.wikipedia.org/wiki/Fold_(higher-order_function)#In_various_languages