# Husky: A C++ Functional Programming Library

# Design Document

## Contents

## 1. Introduction

The functional programming paradigm treats computation as the evaluation of functions, avoiding state changes and mutable data. It intends to eliminate side effects (i.e. changes in state that do not depend on the function inputs), to make it easier to understand and predict the behavior of a program. Several purely functional languages have been developed over the years, with Haskell being the most famous one.

The rise of the functional paradigm has reached classic imperative languages as well. Java and C++, for example, have over the years incorporated some functional features, like the use of lambdas and the emulation of higher-order functions. Husky was developed with the intent of providing C++ programmers with several higher -order functions, using an interface closer to the functional programming style.

Husky does not pretend to replace the STL, but instead extends it. As such, the library supports STL standard containers, barring some exceptions (that will be discussed in the interface design). Several of the library's functions also can receive lambdas or functors as parameters, and the return types are defined accordingly. We have also used STL functions as much as possible, as they are certainly more optimized and tested than anything we could possibly develop.

This design document will explore the features present in Husky, such as the interface design, error handling and resource management. It will also discuss the future extensions planned, and how they would conform to the rest of the library and to C++ itself.

## 2. Feature set

During the course of the project, we had to redefine several times our feature set. Our initial intent was provide a complete outline for functional programming in C++, providing support for Monads and lazy evaluation. As we tried to implement such features, however, it became clear that they would not be doable within our timeframe, and where more suited themselves to be each an entire project by itself.

We have then focused our efforts on building an extensive library that provides both higher-order and container manipulation functions, inspired by Haskell's Prelude module. In the future, we intend to implement a lazy evaluated version of those functions, as lazy evaluation is one of the big features of Haskell, and allows it to provide functions dealing with infinite lists, for example.

The final Husky's feature set is divided into two big groups: higher-order functions and container manipulation functions. String manipulation functions such as words or lines are also included in the second group.

### 2.1 Higher-order Functions

Higher-order functions come from mathematics. They are functions that take one or more functions as input and/or output a function itself. In C++, the STL has been gaining more and more versions of some classic higher-order functions (e.g. std::accumulate, std::transform, etc.). These functions are the bread and butter of functional programming, and are pervasive in languages such as Python, Haskell and Scala.

We have implemented in Husky all the main higher-order functions, along with its many variants, using Haskell's Prelude as guideline. The complete list is indicated in the table below, and information regarding the use of each function can be found in the tutorial and in the manual documentation for the library.

| | |
|---|---|
| **maps** | *map, concatMap* |
| **filters** | *filter, takeWhile, dropWhile, span_container, break_container* |
| **folds** | *foldl, foldl1, foldr, foldr1, any_fold, all_fold,* |
| **scans** | *scanl, scanl1, scanr, scanr1* |
| **zips** | *zipWith, zipWith3* |
| **function composition** | *compose* |

*Table 1 Higher-order functions implemented by Husky*

## 2.2 Container manipulation functions

Lists are the basic container in Haskell. As such, there are many list manipulation functions, and they are constantly used in functional programs. As C++ gives support to many different containers, Husky tries to be as generic as possible when implementing similar functions. When a container must be defined (for return purposes, for example), we have decided to use the std::vector container, as the tests realized during this course have indicated that vector is the most optimized STL container.

The complete list of container manipulation functions implemented by Husky is on the table below. We consider the string only functions to be container manipulation as well, as strings in Haskell are only a special type of list.

| | |
|---|---|
| **concatenation** | *concat, concat1* |
| **list properties** | *head, last, tail, init, is_null, length, at, elem, notElem* |
| **list manipulation** | *reverse, take, drop, splitAt, cons* |
| **special folds** | *and_fold, or_fold, sum, product, maximum_of, minimum_of* |
| **zips** | *zip, zip3, unzip, unzip3* |
| **string manipulation** | *lines, words, unlines, unwords* |

*Table 2 Container manipulation functions implemented by Husky*

## 3. Interface design

One of the first dilemmas that we encountered while defining our interface was whether we should follow C++ algorithm approach, dealing directly with iterators, or if we should try to base our interfaces in the functional programming style, which deals with the containers themselves. We have decided to follow the second path, because one of the main objectives of our library is to provide functional programmers with familiar syntax and style when programming in C++.

We have also tried to follow as much as possible Haskell's nomenclature, in order to keep with that philosophy of easing the transition from one language to another. Exceptions had to be made for some functions, that had forbidden names (!! transformed into the '*at*' function, and *break* was renamed as *break_container*, for example).

We have also endeavored to maintain a consistent parameter order and nomenclature across our functions and templates. This makes the code more readable. Unfortunately some functions still got big prototypes, as some of the types depended explicitly on the return type of a parameter function, as in zipWith, for example.

The functions code was also designed in a way to maximize STL function calls, in order to keep the code as optimized as possible. This also has the side effect of providing an easier to understand code flow, as STL functions are extensively documented, in contrast to manually written code. We have also preferred C++11 style and features, such as use of the *auto* keyword, range-for loops and lambdas, for example. We have also used a C++14 feature to implement the *compose* function, using *auto* inside lambdas.

As the STL functions and containers already have a satisfactory error handling and resource management system, we have relied on them to provide these functionalities to our library. Whenever a function or container throws an exception, our corresponding function will throw it as well. Higher-order functions also have some form of template type-checking. The *zipWith* function, for example, requires a parameter function that can be applied to the types of the two containers that will be zipped, and generates a std::vector of the return type of said function.

The resource management is also delegated to the STL containers. As one of the bastions of functional programming is the absence of side effects, all of our functions do not alter their parameters in any way, creating new containers instead. This was a conscious decision, as we are trying to emulate as much as we can the functional programming paradigm. Parameters were also declared *const*, to emphasize this point. We have also preferred passing by reference as much as possible, as containers can be very heavy, leading to slow copying times.

More detailed information regarding each implementation can be found on the manual pages. One of the things that made testing against C++ libraries particularly difficult for us was the lack of proper documentation, mainly in FC++. We had to read the library's source code, in order to understand its uses. As we are providing both manuals and a tutorial, we hope that other programmers don't have the same difficulties as we had, and that they can integrate our library as easily as possible. In order to get access to Husky functionalities, a programmer can copy our header file to their project or simply reference the library folder in their includes, and include the header file. We recommend the later procedure, in order to keep projects modularized.

## 4. Future implementations

There are many possible extensions for Husky. As functional programming has a good deal of idiosyncrasies and particularities, there would always be something new to add to our library. Our more immediate possible improvements would be to implement complete support to currying and lazy evaluation. Another possible big extension would be to fully implement Monads, which would allow for a more integrated with Haskell experience.

Currying translates a function of multiple arguments into a sequence of functions with a single argument. It is directly related to partial application, which is the process of fixing a number of arguments to a function, producing another function of smaller arity. In C++, partial application is present in the form of the std::bind function, but there is no curry support. We intend to provide it as a future feature of our library, as implementing it would heavily need C++14 additions.

Another big staple of Haskell is lazy evaluation, a strategy that delays the evaluation of an expression until its value is needed, and which also avoids repeated evaluations. It leads to performance increases by avoiding needless calculations, as well as the ability to construct and manipulate infinite data structures. We intend in the future to implement lazy evaluated versions of Husky's functions, and extensions to STL containers that would allow this optimization, without losing all of C++ standard containers' capacity.

Finally, another intended extension of Husky is to implement at least a basic support to monadic programming. Functional programs can use monads to encapsulate a program's state. They also allow the structuration of sequenced operations and are the only possible way to include side effects in Haskell, for example. As monads are very characteristic of functional programming, it would be a challenge to implement all its functionalities in an imperative language such as C++.

## 5. Conclusion

With C++11 and soon C++14, we can see that many functional programming concepts are making their way to C++ Standard Library. As such, we have developed Husky to add even more functional tools for those who are either interested in using a different paradigm without leaving C++, or making the jump from Haskell to a more imperative style.

We have endeavored to follow as much as possible the STL paradigm for libraries, relying on already tried and tested auxiliary functions instead of trying to replace them with our own. Our intention is for Husky to be used alongside STL, as an easy and intuitive library that has satisfactory efficiency. We have also provided documentation and a tutorial for this purpose.

In the future, we hope to extend Husky's capabilities even more, in order to provide developers with a true functional programming outline in C++. The addition of Haskell's staples such as currying, lazy evaluation and monads will be a great step in that direction. We also intend to provide less cumbersome function prototypes and C++14 Concepts support in the future.