Why Functional Programming Matters

Functional programming languages offer a wide variety of benefits to modern day computer science as software grows more and more complex. A lot of those benefits stem from the simplicity that functional languages provide through the absence of assignment, side effects, and control of flow. On the other hand one of the main benefit of structured programming languages is the ability to modularize code or "glue solutions together" thereby introducing better productivity, faster development of subsequent programs, and reduce time spent debugging. This paper will prove that even functional programming languages can allow modularization through two ways: 1) enabling simple function to be glued together and 2) enabling whole programs to be glued together.

Functional programming provides us with the ability to "glue" functions together using higher order functions such as 'foldr' and 'map' stacked on top of standard functions like 'sum' or 'count'. Approaching a function like 'sum' with modularization in mind results in the creation of 'foldr' which can be used to write many other functions on lists with little to no additional programming effort. This example in the paper regarding list processing is also easily extended to other complex data structures like trees use a newer version of 'foldr' named 'foldtree'. Functional programming allows for functions that can't be modularized in structured programming languages to be broken up into a combination of a higher order function (i.e. foldr, map) and a specialized function (i.e. sum, count). The other benefit of this approach is that it is easily extended to new datatypes like in the example of extended foldr to foldtree when switching context from lists to trees.

In this paper we define a program as just a function from input to output. In functional programming it's common to chain functions together via nesting. In a structured programming language it would've been common store output from one program (function) in a temporary file and have another program (function) read that temp file as input but in functional programming that's not necessary because chaining functions together forces them to run in strict synchronization. Let's take an example g (f (x)) where program/function takes x as input and then the output from f is fed to program/function g. Program/function f is started only when g is run and tries to read input and only runs for as long as it takes for f to deliver the output to g. Program/function then is automatically suspended and g continues its processing and f will only resume if g fetches more input. As a result this allows termination conditions in functions to be completely separate from loop bodies which is another powerful modularization and this method of program/function evaluation is called "lazy evaluation" because the evaluation of the nested program/function f only runs as little as possible to satisfy g's input requirements. The question then becomes "why can't we extend the benefits of lazy evaluation to nonfunctional languages?" the answer turns out is quite simple. Lazy evaluation would require programmers to give up direct control flow of the application and that would make life more difficult for programmers in structured non-functional languages where programs are not set up to be arbitrarily executed in randomly ordered sections.

Lazy evaluation provides a lot of benefits in the field of numerical analysis + algorithms. The first example discussed in the paper demonstrates the lazy evaluation of a combination of the 'repeat' and a user defined 'next' function to estimate square root. We can see that repeat itself is an infinite output higher order function but embedding (next n) as the first argument

put a finite bound of n using the strict synchronization execution discussed earlier. Another is how lazy evaluation can also be used to perform calculus operations like differentiation which is defined by 'within eps (improve (differentiate h0 f x))' where differentiate is a user defined function or for integration using 'within eps (integrate f a b)'. The last example discussed in this paper is in the artificial intelligence space where we can use the two methods of "glue" along with these higher order functions to predict the likelihood of the AI winning a game of tic-tac-toe against a human opponent. The game moves are set up in a tree-like data structure and the game board itself is a matrix so we can leverage our tree and matrix higher order functions referenced earlier in the paper. The algorithm described in this scenario is also combining the use of higher order functions with lazy evaluations of 'minimum' and 'maximum' to recurse the "game trees" to find which nodes result in the closest value to '1' meaning AI won. The pruning function only calculates a finite number of game trees for the algo to evaluate and as a result creates a much more efficient program since the entire universe of possible game trees never have to sit in memory unless the prune function calls for them. The absence of lazy evaluation in this example would result in a monolithic application that is not only hard to code but even harder to modify or debug.

In conclusion, we know that modularity is the key to any successful programming language in the modern era because of the benefits they provide in scaling developer productivity. We can't only rely on advancements in compilation rules and mechanisms a language has to provide that "glue" – in this case the glue is higher-order function and lazy evaluation. As a result we see that functional programs are so much smaller and easier to code than popular structured languages. Experienced functional programmers know that they can

target messy / complex parts of program and modularize them into smaller parts using methods such as higher order functions and lazy evaluation.

Works Cited

http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html -- Why Functional Programming Matters