Comments on "Why Functional Programming Matters" by Chalmers

Chalmers's memo "Why Functional Programming Matters" is essentially a well-written argumentative essay, which demonstrates the strong points of functional languages. Chalmers starts by introducing the usual definition of functional programming – no side effects, results independent of execution order. However, he explains that these just explain what a functional language won't do – not what it will. He then goes on to address one of the biggest issues in programming languages – modularity. Modularity involves breaking up a problem into smaller sub problems, solving each one, and then combining the solution. Chalmers claims that functional programming allows for better combination of sub problems in order to solve a bigger problem, and thus allows for better modularity in general.

Chalmers, then, uses numerous examples to display what he thinks are the functional tools most useful in gluing sub problems together. These tools are higher-order functions and lazy evaluation. As an example of powerful higher-order functions, he discusses the "reduce" function, which reduces a list to a single value by repeatedly applying a binary operator to it. This function is indeed very powerful – it can be used to sum up all elements of a list, multiply all elements of a list together, append two lists to each other, double all elements in a list, etcetera. All these functions would require independent implementation in an imperative language, while in a functional language they can be greatly modularized.

The second tool that Chalmers discusses is lazy evaluation. It can be used to glue whole programs together; since functional programs are just functions and we can easily compose them. If we have f (g (x)), with lazy evaluation g (x) will only run when necessary, thus allowing the composition of functions that continuously produce output. Chalmers then gives a few other examples of using lazy evaluation to calculate square roots, derivatives, and integrals or to efficiently implement the Alpha-Beta algorithm from AI. Though these examples are complex, they hinge on one essential thing, which is being able to work with infinite data structures. With lazy evaluation, only the necessary parts are evaluated.

Overall, I found the examples in this paper very exciting and original. Chalmers succeeded in impressing me with the power and ease of functional languages. However, the only functional language I know, Scheme, doesn't use lazy evaluation by default! Chalmers briefly mentions that some functional languages do not use lazy evaluation by default, but he does not explain why. We've seen in class that lazy evaluation has some unpleasant limitations, such as the possibility of unnecessarily evaluating a certain expression many times.

Though this week's presenter, XXX, demonstrated basic understanding of the paper, I believe that his presentation omitted some of the higher-level ideas. XXX meticulously went through the complex examples in the paper, explaining how they worked, but he did not adequately explain why these examples are important. He should have taken the "theme" of the paper – lazy evaluation and higher-order functions and presented it first, and then cited examples. Overall, I think he should be forgiven, since this week's paper was technical and quite difficult to read.