CHAPTER 1

Constrained systems are more structured

This is especially a discussion of how Backus's FP language is more tractible, by being less powerful and lacking self-reference.

There is a duality between objects and their structure; the more general or powerful an object, the less you can say about it, while the more constrained an object, the more you can say about it: the more structure it has.

One might quip "slavery is freedom".

Saunders Mac Lane writes (in a comment in "Categories for the Working Mathematician"): "...good general theory does not search for the maximum generality, but for the right generality."; I would say: "sometimes, say more about less".

1. Computer examples

A good example is programming languages versus markup languages.

At one end, XML is very simple (it is a markup language); basically, it's a syntax for trees. As such, you can analyze them and work on them very easily:

- it is very easy to write an XML parser
- it is very easy to check if two XML documents are equal

They can be completely understood by reading them and parsing them: you do not need to do any further computation.

At the other end, any Turing/von Neumann-type language is very complex.

- it is relatively hard to write a C compiler
- it is impossible to algorithmically determine properties of a program: you cannot determine if two programs are equal (produce the same output) or halt

You cannot understand a program without running it (at least mentally). This is particularly pronounced with reflective languages, like

self-modifying code: you can't analyze the behavior of such a program by simply reading its code – not only its behavior but its *structure* can only be determined by running it! As the example of the halting problem demonstrates, there are limits on what you can determine about programs via a program.

In between, constrained languages (like regular expressions or SQL) are powerful enough to be quite useful, but constrained enough to do meaningful changes and analysis: one can transform SQL queries, for instance, or rewrite regular expressions.

1.1. Backus's Function-level programming. John Backus's function-level programming (and his FP language) is a fascinating example: traditional (von Neumann style) programming languages are "too powerful": they are a very general class of programs and thus you can't do much with them: you can't analyze them, you can't optimize them well, you can't parallelize them and so forth.

Instead, in Backus's language FP, you start with atoms, and then build functions from them. The rules for combining atoms are specified in a fixed set of higher order functions ("functional forms"); you can program new functional forms in a *separate* meta-language, Formal FP.

This constraint means that, for a fixed set of functional forms, the possible programs are a module over the algebra of functional forms. This is much more constrained than a general programming language, and hence easier to study and more engineerable, precisely because it lacks self-reference (it has a hierarchy of meta-programming, instead of meta-meta = meta).

Backus's Turing award lecture, "Can Programming Be Liberated from the von Neumann Style?", is interesting itself; I've outlined my main interest in it above though.