EECS 598 Paper Review by Kien Yue Kuo
Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions
By Dawnson Engler et al.

This paper introduces the idea of using meta-level compilation (MC) to write compiler extensions that automatically check code for rule violations. The authors did some testing using MC on real systems like Linux and OpenBSD and found quite an impressive number of errors.

The idea behind MC is to use static analysis in the compiler. This extends the compiler and use language-based patterns to recognize operations that are to be checked, implemented as a state-machine. It helps to identify problems like double freeing a pointer or sleeping while holding a lock. These problems are not checked by the compiler but it can be extended to warn the programmer of these errors.

One of the problems of using MC is the number of false positives. If the false positives overwhelm the number of real errors in the code, then the user will not use it. This may depend on the system, since some systems may use syntax that looks wrong to the compiler. This means the MC must be tweaked for different systems, but that is not such a big deal, since that might require some work by an experienced programmer, but since MC code is usually about a few hundred lines only, it would not take a long time to have such rules in place.

Meta-level compilation is a very useful tool, especially in complex systems. It is impossible for a human to keep all the requirements in mind, and write code that conforms to all the rules. Having the compiler do the checking at compile time can help to reduce the mental workload of the programmer, and catch errors caused by careless mistakes that are not caught by the compiler normally.

One major limitation of the extension (as of the paper’s writing) is that the checker only performs local analysis. For example, in checking for mutex in Linux, 48 false positives are caused by that. This is caused by warnings issued when functions exit while holding locks, but these can be eliminated with global analysis. However, global analysis will mean that the number of states to keep track of will be way larger, since branching in a program grows exponentially, and if this is performed on a complex system, the work required may be huge. Also, whether to use global analysis depends on how useful it is. If most analysis can be done locally, and only a few function calls require global analysis, it may not be worth the effort. Adding this functionality will be nice, and if should be left as an option for the user to choose whether or not to use it.

Meta-level compilation is a good idea and it may be beneficial for future compilers to integrate this functionality, since currently it is dynamically linked into their own compiler, xg++. It may improve performance since some internal data of the compiler when compiling the code can be used for the benefit of the extension.
In conclusion, meta-level compilation lets the compiler ‘know’ about some rules that the programmer must follow, and warn if there are any violations. So now all that is needed is to write a set of rules in their language, *metal*. Instead of keeping track of all the rules in every point of the program in the mind of the programmer, this work is now delegated to the compiler. If the programmer forgets to release a lock before returning on an error, the compiler now detects it. These rules can be reused, and may require only little tweaks for different systems, so meta-level compilation is a valuable tool for the programmer which should be more widely used.