Seminar Configurable Systems

Prof. Zeller / Christian Lindig libscheme

Summary

Brent W. Benson Jr.'s article *libscheme: Scheme as a C Library* appeared 1994 in the Proceedings of the USENIX Symposium on Very High Level Languages. It presents Scheme as a language and its interpreter-based implementation *libscheme* as an alternative to the Tcl extension language.

Benson demonstrates that Scheme as a language provides a richer set of concepts than Tcl. Because these concepts are orthogonal, they can be combined and lead to short and powerful programs. Benson demonstrates (without being exhaustive) call by value, lexical scoping, nested procedures, and various data types. He argues that these are desirable in a scripting and extension language. Consequently he proposes his implementation *libscheme* of Scheme as a C library as an extension language for C programs.

What makes Scheme powerful as a language makes it difficult to implement: various data types, first-class functions, lexical scoping. An extension language must hide these complexities at the extension API that developers use. Benson acknowledges this by attributing Tcl's success largely to the ease with which it can be embedded into C applications. Despite a more complex language, he aims to achieve the same for Scheme.

Benson implements Scheme as an interpreter with a uniform representation of Scheme objects. Such an object is a C union that lives on the heap. Because of this, function activations are not managed as a stack but create (a lot of) garbage. This leads to a slower implementation compared to a Scheme interpreter that tries to use a stack on which it also keeps scalar data types. On the other hand, the uniformity of representation simplifies the extension API considerably.

Memory in the Scheme interpreter is managed by the conservative Boehm-Weiser garbage collector for C and Benson recommends to use it for the application code as well. Otherwise sharing the responsibility for freeing memory might be tricky: who is responsible for freeing memory that was allocated by the application and is referenced by code from the interpreter?

While the Tcl extension API provides one mechanism to register new functions, the Scheme API provides three mechanism to register new language constructs: for functions, types, and syntax. While the need to register new functions is obvious, the other two come as a consequence of Scheme's design: Scheme knows types that primitives and procedures check at runtime. Added complex functionality also requires new types and hence the need to create and register such types. Since Scheme uses call by value, procedures cannot implement new syntax and a special mechanism is required. This is different from Tcl, which passes arguments unevaluated to C. Hence, a single mechanism in Tcl can implement regular procedures as well as new syntax.

As its main contribution, the paper demonstrates that the extension interface to a powerful language like Scheme does not have to be more complicated than the interface to a much simpler language like Tcl. The paper hints at projects that are realized with *libscheme* but does not elaborate them. It would have lent credibility to the argument that Scheme is superior to Scheme by showing them in more detail. The paper would benefit from a stronger motivation why an extension language needs the power of Scheme and a less specialized example.