SMITH845
Smith, B.C.;
Reflection and Semantics in Lisp;
Proceedings of ACM Symposium on Principles of Programing Languages
(May 1984).

This paper steps through the design of a dialect of Lisp containing primitives that give a programmer the ability to manipulate control structures in a manner similar to how most programming languages have allowed the manipulation os data structures. The model used in this Lisp dialect, known as 3-Lisp, is that of having an infinite tower of processors, each processor executes the processor one level below itself. Arguments are made that this infinite hierarchy is in fact finitely and efficiently implementable, and code for a 3-Lisp processor written in 3-Lisp is presented.

Although the idea of reflection or introspection is very old, it is only recently that formalisms based on the concept have been presented. In the last 20 years there had been intermittent attempts at such formalization, but the only practical results before 3-Lisp was Weyhrauch's FOL (First Order Logic) language. Brian Smith is applying the ideas of reflective hierarchies of processors in a procedural context, namely as an embedding in a language that is traditionally thought of as self-descriptive (Lisp). As he points out, the self-descriptive abilities of Lisp are still limited to manipulating data structures; what is missing, and the contribution of Smith's reflective 3-Lisp is the ability to create new control structures within the language. This capability is equivalent to what one could do if the Lisp processor (interpretor) was written entirely in Lisp. In practice this is a circular definition of the language, and therefore impossible.

In order to build 3-Lisp, the author argues that an intermediate step is desirable due to the semantic ambiguities of Lisp notation. The 2-Lisp dialect of Lisp is constructed in a very orthogonal manner, drawing clear distinctions between the use and the mention of a language object; Smith calls 2-Lisp a "semantically rationalized" dialect. The important difference from normal Lisp is that there are precise notations for making these distinctions, and so the issue of semantic ambiguity does not cloud the design of 3-Lisp.

With 2-Lisp in hand as a precise tool for self-description, a "reflective tower" of 2-Lisp interpreters is constructed, each running the interpreter below it in the hierarchy. The code at each level of the tower is identical. 3-Lisp then, is essentially the reflective tower with an infinite number of levels (i.e. in the limit).

The code at each level of the reflective tower is based on a simple 2-Lisp interpreter (corresponding to a read-eval-print loop in normal Lisp), but extended to allow a user program access to the control structure information. This escape capability is accomplished through "reflective procedures" that are run one level higher that the level of their invocation. Reflective procedures have all the facilities of the language they are implementing (sic) available for use.

The paper gives examples of how traditionally hardwired functions lik binding predicates, returns, quoting, and new function types can be implemented in 3-Lisp with judicious use of reflective procedures. With the appropriate hooks, thes (or very similar) capabilities are available in traditional Lisp, however one example that demonstrates the power of the model is the implementation of the THROW/CATCH construct in 3-Lisp. The capability of 3-Lisp that makes it possible, is that reflective procedures have access to the environment, and, in this case more importantly, the continuation of their invoking process. The THROW/CATCH pair essentially just says to return to the continuation of the CATCH instead of the continuation of the throw. The continuation can be thought of as the embodiment of the future execution path from some point in the execution.

The ability to pass continuations and environments around as parameters definitely provide much of 3-Lisps power. One objection to this kind of computational model is that the programmer needs explicit knowledge of it to be able to take advantage of its power. The answer to this objection is that the power of a language that manipulates control structure increases as the information about that control structure becomes more fine-grained, and so it is not unreasonable to expect a correspondingly better understanding of the model of computation by the programmer.

The thing that makes the reflection model implementable is that almost all the levels of the reflective tower, except for a few at the bottom that the user code accesses directly, will always be in the same state. This means that most of the reflective tower can be simulated by a single, slightly expanded, reflective processor. In fact, a couple of implementations of 3-Lisp exist, but they are naturally slower than their host language (Lisp) because they essentially provide another level of interpretation between the machine language and the task.

The ideas with reflective programming certainly seem very different from the traditional restrictions in programming languages, namely that only data structures can be manipulated. What the author does with Lisp (developing 3-Lisp) is not something unique to Lisp. The methods are generally applicable although their formal statement will be less clear. The ideas in 3-Lisp will show up in special-purpose AI languages because of the elegance of being able to encode data and strategic (control) knowledge similarly and using the same tools to develop the two aspects of a problem.