Annotations[1]

A1 ·1/-1/-3:-1 It is striking that this characterization is expressed in terms of *reification*, which at least in the first instance appears to be an ontological notion, rather than semantically, in terms of an ability for a reflection system to reason "about" its own structures, operations, and behavior. For purposes of this paper the semantical point was relegated to a foonote (·4/n3; cf. A4, below). As discussed in §··· of the Cover, however, I believe that reification, too, is fundamentally a semantical phenomenon.

A1.5 ·2/1/-2:-1 Cf. annotation A... of ch. 4, at ·.../....

A3 ·4/1/4:5 In describing structures in terms of "programs and data" we were imposing only a single serial reduction[2] between the overall behavior of a user's program and the underlying language processor, rather than two: one between the overall user process A and the process B engendered by the user's program dealing with their data structures, and then a second, in turn, on B, describing it as constituted by the program (i.e., the code as passive) and the active language processor. The reason for not imposing the more complex dual registration in the reflective case arose in part because of the complexities that result in dealing with reflective code, which from one point of view is data (to the reflective processor), and from another point of view is "interior" to the process B that deals with the user's object-level data structures.

A4 ·4/n3 Cf. A1, above.

A5 ·7/1/-3:-1 It would have better if the last sentence of this paragraph had been written: "The relationship is this: if we say that G is running at level *k*, we mean that a program at level *k* is run by G directly, without the intervention of any higher levels of RPP."

A6 ·7/2/-1 The previous try would "not succeed" just in case G were to encounter a reflective request, which it would not be equipped to handle.

A7 ·8/1/8:9 Cf. the discussion in the ch. 2 «...where?...» of the constant theme, which permeates 3Lisp and the presented model of reflection in general, of eliding or even fusing semantical notions of description and procedural notions of implementation.

A8 ·10/0/3 Absorption is introduced on p·· of §8 of "The Correspondence Continuum," ch. 12; cf. also annotation A34 of "Reflection and Seman-

1. References are in the form *page/paragraph/line; with ranges (of any type) indicated as x:y. For details see the explanation on p.·*

2. For an explanation of serial reductions see. ch. ··, §··, p. ··.

tics in Lisp," ch. 4, re ·31/1/-9:-1 in that paper, where there is also a brief introductory characterization. Note also the subsequent use of the notion here in ·10/2.

A9 ·14/0/3 '□' and '□' are primitive (built-in) notational abbreviations for the simple extensional procedures UP and DOWN, respectively.[3] Thus '□*EXP*' and '□*EXP*' are fully equivalent—both procedurally and declaratively—to '(UP *EXP*)' to '(DOWN *EXP*)', respectively.

A10 ·17/-2 The issues of what otherwise implicit aspects of a computation should be "explicitised" upon reflection, and of how to define dialects in which some but not others could be rendered explicit in ways that would dovetail with other reflective code that made different aspects explicit, were background concerns throughout the work on 3Lisp. Cf. the discussion in annotation A43 re passage ·47/0/-5:-1 of "Reflection and Semantics in Lisp," ch. 4, which talks about discussions of this issue in my research group at Xerox Palo Alto Research Center (PARC) in the 1980s, and the subsequent emergence of aspect-oriented programming from members of that group.

I believe that the issue remains open and appropriate for further research, but also that treating it adequately will require something on the order of the fan calculus discussed in the Introduction.

A11 ·17/-1/3:4 Lisp 1.5 and other standard Lisps do not need to quote lambda expressions when used, as it might be said, "in function position"—e.g., in such a construct as ((LAMBDA (X) (+ X 1)). In all such dialects, however,[4] they do have to be quoted when they are passed as arguments or results—e.g., in the expression (APPLY '(LAMBDA (X) (+ X 1)) 3)—or, as a philosopher might put it, when the functions they designate are *mentioned* or *objectified*.

A12 ·18/0/4 FEXPRs in MacLisp and NLAMBDAs in Interlisp were so-called "special forms," used to define functions which did not automatically evaluate their arguments. Cf.:[4.5]

http://www.nhplace.com/kent/Papers/Special-Forms.html

As indicated in the above report, FEXPRs and NLAMBDAs were widely disparaged—in no small part, I believe, because, while inchoately re-

3. UP and DOWN were called NAME and REFERENT, respectively, in Smith [1982], of which parts are included here in ch. 3.

4. I.e., in all dialects of Lisp—at least at the time this was written—other than Scheme, 2Lisp, and 3Lisp.

4.5. «...cite; date...»

flective, they were not provided within a context in which reflective or meta-level access to "code as data" was adequately understood, disciplined, or controlled.

A13 ·34/0/-3:-1 «...cite, describe, provide a pointer to, Jun's Ruby implementation»

A14 ·35/-1/-2 In the technical report version of this paper (cf. p. ··), this line was erroneously printed as

(&&call state cont □(□proc! . □args!))

however the ACM version was correct.

A15 ·40/0/-3 As mentioned earlier «...where?...», the implementation included as an appendix in (Smith 1984), the original dissertation, did not handle this issue properly. Cf. annotation A83, re ·136/0/-1, in ch. 3b.

A16 ·45/-1/4 An SECD machine[5] is an abstract virtual machine, originally designed by Peter Landin, designed to evaluate λ-calculus expressions, which became a standard target for compilers of functional programming languages.

5. An acronym for "Stack, Environment, Code, Dump," names of its internal registers.