EQUATIONAL LOGIC

as a

PROGRAMMING LANGUAGE

Michael J. O'Donnell

EQUATIONAL LOGIC

as a

PROGRAMMING LANGUAGE

Michael J. O'Donnell

The MIT Press
Cambridge, Massachusetts
London, England

Preface.

This book describes an ongoing equational programming project that started in 1975. Principal investigators on the project are Christoph Hoffmann and Michael O'Donnell. Paul Chew, Paul Golick, Giovanni Sacco, and Robert Strandh participated as graduate students. I am responsible for the presentation at hand, and the opinions expressed in it, but different portions of the work described involve each of the people listed above. I use the pronoun "we" throughout the remainder, to indicate unspecified subsets of that group. Specific contributions that can be attributed to one individual are acknowledged by name, but much of the quality of the work is due to untraceable interactions between several people, and should be credited to the group.

The equational programming project never had a definite pseudocommercial goal, although we always hoped to find genuinely useful applications. Rather than seeking a style of computing to support a particular application, we took a clean, simple, and elegant style of computing, with particularly elementary semantics, and asked what it is good for. As a result, we adhered very strictly to the original concept of computing with equations, even when certain extensions had obvious pragmatic value. On the other hand, we were quite willing to change the application. Originally, we envisioned equations as formal descriptions of interpreters for other programming languages. When we discovered that such applications led to outrageous overhead, but that programs defined directly by equations ran quite competitively with LISP, we switched application from interpreter generation to programming with equations.

We do not apologize for our fanaticism about the foundations of equational programming, and our cavalier attitude toward applications. We believe that good

mathematics is useful, but not always for the reasons that motivated its creation (non-Euclidean geometry is a positive example, the calculus a negative one). Also, while recognizing the need for programming languages that support important applications immediately, we believe that scientific progress in the principles of programming and programming languages is impeded by too quick a reach for applications. The utility of LISP, for example, is unquestionable, but the very adjustments to LISP that give it success in many applications make it a very imprecise vehicle for understanding the utility of declarative programming. We would rather discover that pure equational programming, as we envision it, is *unsuitable* for a particular application, than to expand the concept in a way that makes it harder to trace the conceptual underpinnings of its success or failure.

Without committing to any particular type of application, we must experiment with a variety of applications, else our approach to programming is pure speculation. For this purpose, we need an implementation. The implementation must perform well enough that some people can be persuaded to use it. We interpret this constraint to mean that it must compete in speed with LISP. Parsers, programming support, and the other baggage possessed by all programming languages, must be good enough not to get in the way, but the main effort should go toward demonstrating the feasibility of the novel aspects, rather than solving well understood problems once again.

The equational programming project has achieved an implementation of an interpreter for equational programs. The implementation runs under Berkeley UNIX* 4.1 and 4.2, and is available from the author for experimental use. The current distribution is not well enough supported to qualify as a reliable tool for

*UNIX is a trademark of AT&T.

important applications, but we have hopes of producing such a stronger implementation in the next few years. Sections 1 through 10 constitute a user's manual for the current implementation. The remainder of the text covers a variety of topics relating to the theory supporting equational programming, the algorithmic and organizational problems solved in its implementation, and the special characteristics of equational programming that qualify it for particular applications. Some sections discuss work in progress. The intent is to give a solid intuition for all the identifiable aspects of the project, from its esoteric theoretical foundations in logic to its concrete implementation as a system of programs, and its potential applications.

Various portions of the work were supported by a Purdue University XL grant, by the National Science Foundation under grants MCS-7801812 and MCS-8217996, and by the National Security Agency under grant 84H0006. The Purdue University Department of Computer Sciences provided essential computing resources for most of the implementation effort. I am grateful to Robert Strandh and Christoph Hoffmann for critical readings of the manuscript, and to AT&T Bell Laboratories for providing phototypesetting facilities. Typesetting was accomplished using the *troff* program under UNIX.

Table of Contents

1. Introduction (adapted from HO82b)

Computer scientists have spent a large amount of research effort developing the semantics of programming languages. Although we understand how to implement Algol-style procedural programming languages efficiently, it seems to be very difficult to say what the programs mean. The problem may come from choosing an implementation of a language before giving the semantics that define correctness of the implementation. In the development of the equation interpreter, we reversed the process by taking clean, simple, intuitive semantics, and then looking for correct, efficient implementations.

We suggest the following scenario as a good setting for the intuitive semantics of computation. Our scenario covers many, but not all, applications of computing (e.g., real-time applications are not included).

> A person is communicating with a machine. The person gives a sequence of assertions followed by a question. The machine responds with an answer or by never answering.

The problem of semantics is to define, in a rigorous and understandable way, what it means for the machine's response to be correct. A natural informal definition of correctness is that any answer that the machine gives must be a logical consequence of the person's assertions, and that failure to give an answer must mean that there is no answer that follows logically from the assertions. If the language for giving assertions is capable of describing all the computable functions, the undecidability of the halting problem prevents the machine from always detecting those cases where there is no answer. In such cases, the machine never halts. The style of semantics based on logical consequence leads most naturally to a style of programming similar to that in the descriptive or applicative languages such as

LISP, Lucid, Prolog, Hope, OBJ, SASL and Functional Programming languages, although Algol-style programming may also be supported in such a way. Computations under logical-consequence semantics roughly correspond to "lazy evaluation" of LISP [HM76, FW76].

Semantics based on logical consequence is much simpler than many other styles of programming language semantics. In particular, the understanding of logical-consequence semantics does *not* require construction of particular models through lattice theory or category theory, as do the semantic treatments based on the work of Scott and Strachey or those in the abstract-data-types literature using initial or final algebras. If a program is given as a set of assertions, then the logical consequences of the program are merely all those additional assertions that *must* be true whenever the assertions of the program are true. More precisely, an equation A=B is a logical consequence of a set E of equations if and only if, in *every* algebraic interpretation for which every equation in E is true, A=B is also true (see [O'D77] Chapter 2 and Section 14 of this text for a more technical treatment). There is no way to determine which one of the many models of the program assertions was really intended by the programmer: we simply compute for him all the information we possibly can from what we are given. For those who prefer to think of a single model, term algebras or initial algebras may be used to construct one model for which the true equations are precisely the logical consequences of a given set of equations.

We use the language of equational logic to write the assertions of a program. Other logical languages are available, such as the first-order predicate calculus, used in Prolog [Ko79a]. We have chosen to emphasize the reconciliation of strict adherence to logical consequences with good run-time performance, at the expense

of generality of the language. Current implementations of Prolog do not always discover all of the logical consequences of a program, and may waste much time searching through irrelevant derivations. With our language of equations, we lose some of the expressive power of Prolog, but we always discover all of the logical consequences of a program, and avoid searching irrelevant ones except in cases that inherently require parallel computation. Hoffmann and O'Donnell survey the issues involved in computing with equations in [HO82b]. Section 17 discusses the question of relevant vs. irrelevant consequences of equations more specifically.

Specializing our computing scenario to equational languages:

> The person gives a sequence of equations followed by a question, "What is E?" for some expression E. The machine responds with an equation "E=F," where F is a simple expression.

For our equation interpreter, the "simple expressions" above must be the *normal forms*: expressions containing no instance of a left-hand side of an equation. This assumption allows the equations to be used as rewriting rules, directing the replacement of instances of left-hand sides by the corresponding right-hand sides. Sections 2 and 3 explain how to use the equation interpreter to act out the scenario above. Our equational computing scenario is a special case of a similar scenario developed independently by the philosophers Belnap and Steel for a logic of questions and answers [BS76].

The equation interpreter accepts equations as input, and automatically produces a program to perform the computations described by the equations. In order to achieve reasonable efficiency, we impose some fairly liberal restrictions on the form of equations given. Section 5 describes these restrictions, and Sections 6-8 and 10 present features of the interpreter. Section 15 describes the computational

power of the interpreter in terms of the procedural concepts of parallelism, nondeterminism, and pipelining.

Typical applications for which the equation interpreter should be useful are:

1. We may write quick and easy programs for the sorts of arithmetic and list-manipulating functions that are commonly programmed in languages such as LISP. The "lazy evaluation" implied by logical-consequence semantics allows us to describe infinite objects in such a program, as long as only finite portions are actually used in the output. The advantages of this capability, discussed in [FW76, HM76], are similar to the advantages of pipelining between coroutines in a procedural language. Definitions of large or infinite objects may also be used to implement a kind of automatic dynamic programming (see Section 15.4).

2. We may define programming languages by equations, and the equation processor will produce interpreters. Thus, we may experiment with the design of a programming language before investing the substantial effort required to produce a compiler or even a hand-coded interpreter.

3. Equations describing abstract data types may be used to produce correct implementations automatically, as suggested by [GS78, Wa76], and implemented independently in the OBJ language [FGJM85].

4. Theorems of the form A=B may sometimes be proved by receiving the same answer to the questions "What is A?" and "What is B?" [KB70, HO88] discuss such theorem provers. REVE [Le83, FG84] is a system for developing theorem-proving applications of equations.

5. Non-context-free syntactic checking, and semantics, such as compiler code-generation, may be described formally by equations and used, along with the conventional formal parsers, to automatically produce compilers (see Section 13).

The equation interpreter is intended for use by two different classes of user, in somewhat different styles. The first sort of user is interested in computing results for direct human consumption, using well-established facilities. This sort of user should stay fairly close to the paradigm presented in Section 2, should take the syntactic descriptions as fixed descriptions of a programming language, and should skip Section 20, as well as other sections that do not relate to the problem at hand. The second sort of user is building a new computing product, that will itself be used directly or indirectly to produce humanly readable results. This sort of user will almost certainly need to modify or redesign some of the syntactic processors, and will need to read Sections 13 and 20 rather closely in order to understand how to combine equationally-produced interpreters with other sorts of programs. The second sort of user is encouraged to think of the equation interpreter as a tool, analogous to a formal parser constructor, for building whichever parts of his product are conveniently described by equations. These equational programs may then be combined with programs produced by other language processors to perform those tasks not conveniently implemented by equations. The aim in using equations should be to achieve the same sort of self-documentation and ease of modification that may be achieved by formal grammars, in solving problems where context-free manipulations are not sufficiently powerful.

2. Using the Equation Interpreter Under UNIX (*ep* and *ei*)

Use of the equation interpreter involves two separate steps: preprocessing and interpreting. The preprocessing step, like a programming language compiler, analyzes the given equations and produces machine code. The interpreting step, which may be run any number of times once preprocessing is done, reduces a given term to normal form.

Normal use of the equation interpreter requires the user to create a directory containing 4 files used by the interpreter. The 4 files to be created are:

1. *definitions* - containing the equations;
2. *pre.in* - an input parser for the preprocessor;
3. *int.in* - an input parser for the interpreter;
4. *int.out* - an output pretty-printer for the interpreter.

The file *definitions*, discussed in Section 3, is usually typed in literally by the user. The files *pre.in*, *int.in* and *int.out*, which must be executable, are usually produced automatically by the command *loadsyntax*, as discussed in Section 4.

To invoke the preprocessor, type the following command to the shell

ep Equnsdir

where *Equnsdir* is the directory in which you have created the 4 files above. If no directory is given, the current directory is used. *Ep* will use *Equnsdir* as the home for several temporary files, and produce in *Equnsdir* an executable file named interpreter. Because of the creation and removal of temporary files, the user should avoid placing any extraneous files in *Equnsdir*. Two of the files produced by *ep* are not removed: *def.deep* and *def.in*. These files are not strictly necessary for operation of the interpreter, and may be removed in the interest of space

conservation, but they are useful in building up complex definitions from simpler ones (Section 14) and in producing certain diagnostic output (Section 10). To invoke the interpreter, type the command:

ei Equnsdir

A term found on standard input will be reduced, and its normal form placed on the standard output.

A paradigmatic session with the equation interpreter has the following form:

mkdir Equnsdir
loadsyntax Equnsdir
edit *Equnsdir/definitions* using your favorite editor
ep Equnsdir
edit *input* using your favorite editor
ei Equnsdir <input

The sophisticated user of UNIX may invoke *ei* from his favorite interactive editor, such as *ned* or *emacs*, in order to be able to simultaneously manipulate the input and output.

In more advanced applications, if several equation interpreters are run in a pipeline, repeated invocation of the syntactic processors may be avoided by invoking the interpreters directly, instead of using *ei*. For example, if *Equ.*1, *Equ.*2, *Equ.*3 are all directories in which equational interpreters have been compiled, the following command pipes standard input through all three interpreters:

Equ.1/int.in | Equ.1/interpreter | Equ.2/interpreter |
Equ.3/interpreter | Equ.3/int.out;

Use of *ei* for the same purpose would involve 4 extra invocations of syntactic processors, introducing wasted computation and, worse, the possibility that superficial aspects of the syntax, such as quoting conventions, may affect the results. If

*Equ.*1, *Equ.*2, and *Equ.*3 are not all produced using the same syntax, careful consideration of the relationship between the different syntaxes will be needed to make sense of such a pipe.

After specifying the directory containing definitions, the user may give the size of the workspace to be used in the interpreter. This size defaults to $2^{15}-1=32767$: the largest that can be addressed in one 16-bit word with a sign bit. The workspace size limits the size of the largest expression occurring as an intermediate step in any reduction of an input to normal form. The effect of the limit is blurred somewhat by sharing of equivalent subexpressions, and by allocation of space for declared symbols even when they do not actually take part in a particular computation. For example, to reduce the interpreter workspace to half of the default, type

ep Equnsdir 16384

The largest workspace usable in the current implementation is $2^{31}-2=2147483646$. The limiting factor is the Berkeley Pascal compiler, which will not process a constant bigger than $2^{31}-1=2147483647$, and which produces mysteriously incorrect assembly code for an allocation of exactly that much. On current VAX Unix implementations, the shell may often refuse to run sizes much larger than the default because of insufficient main memory. In such a case, the user will see a message from the shell saying "*not enough core*" or "*too big*".