Corrado Böhm

[Image of Corrado Böhm]
1951:

a compiler that can compile the language in which it is written

a meta-circular compiler

five years before FORTRAN was defined

seven years before LISP was defined
1951:

Corrado Böhm
1951:

a compiler that can compile the language in which it is written
1951:
a compiler that can compile the language in which it is written
a meta-circular compiler
1951:
a compiler that can compile the language in which it is written
a meta-circular compiler
five years before FORTRAN was defined
1951:

a compiler that can compile the language in which it is written

a meta-circular compiler

five years before FORTRAN was defined

seven years before LISP was defined
Prom. Nr. 2186

CALCULATRICES DIGITALES
DU DÉCHIFFRAGE DE FORMULES LOGICO-MATHÉMATIQUES
PAR LA MACHINE MÊME
DANS LA CONCEPTION DU PROGRAMME

THÈSE
PRÉSENTÉE
À L’ÉCOLE POLYTECHNIQUE FÉDÉRALE, ZURICH,
POUR L’OBTENTION DU
GRADE DE DOCTEUR ÉS SCIENCES MATHEMATIQUES
PAR
CORRADO BÖHM, ing. élect. dipl. EPUL
de Milan (Italie)

Rapporteur : Prof. Dr. E. Stiefel
Co-rapporteur : Prof. Dr. P. Bernays
3.4. Deux exemples de description.

1) Algorithme d’Euclide: recherche du plus grand diviseur commun $m$ de deux nombres donnés (v. fig. 2).

A. Soient $a$, $b$ deux nombres donnés.
Appelons $M$ le plus grand et $m$ le plus petit.

B. Le reste de la division de $M$ par $m$ soit $r$.
   Si $r = 0$ poursuivons sous $C$, autrement sous $D$.

C. $m$ est le résultat. Fin.

D. Appelons $M$ le nombre $m$, et appelons $m$ le nombre $r$; poursuivons de nouveau sous $B$. 

\[
\begin{align*}
\pi' & \rightarrow A \\
? & \rightarrow a \\
? & \rightarrow b \\
a \cup b & \rightarrow M \\
a \cap b & \rightarrow m \\
B & \rightarrow \pi \\
\pi' & \rightarrow B \\
\pi' & \rightarrow C \\
m & \rightarrow ? \\
\Omega & \rightarrow \pi \\
\pi' & \rightarrow D \\
m & \rightarrow M \\
r & \rightarrow m \\
B & \rightarrow \pi
\end{align*}
\]
A History of Computing in the Twentieth Century

Edited by
N. Metropolis,
J. Howlett,
and
Gian-Carlo Rota.
THE EARLY DEVELOPMENT OF PROGRAMMING LANGUAGES

by

Donald E. Knuth
Luis Trabb Pardo

STAN-CS-76-562
AUGUST 1976

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY
Böhm's dissertation [BO 52] was especially remarkable because he not only described a complete compiler, he also defined that compiler in its own language! And the language was interesting in itself, because every statement (including input statements, output statements, and control statements) was a special case of an assignment statement.

Böhm's one-pass compiler was capable of generating instructions rapidly, as the input was being read from paper tape.
The complete program for his compiler consisted of 114 assignments, broken down as follows:

(i) 59 statements to handle formulas with parentheses
(ii) 51 statements to handle formulas with operator precedence
(iii) 4 statements to decide between (i) and (ii).

There was also a loading routine, described by 16 assignment statements; so the compiler amounted to only 130 statements in all, including 33 statements which were merely labels \((\pi' \rightarrow \ldots)\). This brevity is especially surprising when we realize that a good deal of the program was devoted solely to checking the input for correct syntax;
Böhm's parsing technique, on the other hand, was of order $n$, generating instructions in what amounts to a linked binary tree while the formula was being read in; to some extent, his algorithm anticipated modern list-processing techniques, which were first made explicit by Newell, Shaw, and Simon about 1956 (cf. [KN 68, p. 457]).
1966:
1966:

Bohm-Jacopini theorem
1966:

Böhm-Jacopini theorem

the theoretical basis of structured programming
Flow Diagrams, Turing Machines
And Languages With Only Two
Formation Rules

Corrado Böhm and Giuseppe Jacopini
International Computation Centre and Istituto Nazionale
per le Applicazioni del Calcolo, Roma, Italy

In the first part of the paper, flow diagrams are introduced to represent inter alia mappings of a set into itself. Although not every diagram is decomposable into a finite number of given base diagrams, this becomes true at a semantical level due to a suitable extension of the given set and of the basic mappings defined in it. Two normalization methods of flow diagrams are given. The first has three base diagrams; the second, only two.

In the second part of the paper, the second method is applied to the theory of Turing machines. With every Turing machine provided with a two-way half-tape, there is associated a similar machine, doing essentially the same job, but working on a tape obtained from the first one by interspersing alternate blank squares. The new machine belongs to the family, elsewhere introduced, generated by composition and iteration from the two machines \( \lambda \) and \( R \). That family is a proper subfamily of the whole family of Turing machines.

1. Introduction and Summary

The set of block or flow diagrams is a two-dimensional programming language, which was used at the beginning of automatic computing and which now enjoys a certain favor. As far as is known, a systematic theory of this language does not exist. At the most, there are some papers by Peter [1], Gorn [2], Hermes [3], Ciampa [4], Riguet [5], Janov [6], Asser [7], where flow diagrams are introduced with different purposes and defined in connection with the descriptions of algorithms or programs.

In this paper, flow diagrams are introduced by the ostensive method; this is done to avoid definitions which certainly would not be of much use. In the first part (written by G. Jacopini), methods of normalization of diagrams are studied, which allow them to be decomposed into base diagrams of three types (first result) or of two types (second result). In the second part of the paper (by C. Böhm), some results of a previous paper are reported [8] and the results of the first part of this paper are then used to prove that every Turing machine is reducible into, or in a determined sense is equivalent to, a program written in a language which admits as formation rules only composition and iteration.

2. Normalization of Flow Diagrams

It is a well-known fact that a flow diagram is suitable for representing programs, computers, Turing machines, etc. Diagrams are usually composed of boxes mutually connected by oriented lines. The boxes are of functional type (see Figure 1) when they represent elementary operations to be carried out on an unspecified object \( x \) of a set \( X \), the former of which may be imagined concretely as the set of the digits contained in the memory of a computer, the tape configuration of a Turing machine, etc.

There are other boxes of predicative type (see Figure 2) which do not operate on an object but decide on the next operation to be carried out, according to whether or not a certain property of \( x \in X \) occurs. Examples of diagrams are: \( \Sigma(\alpha, \beta, \gamma, \alpha, \beta, c) \) [Figure 3] and \( \Theta(\alpha, \beta, \gamma, \delta, \epsilon, \alpha, \beta, c, d, e) \) [see Figure 4]. It is easy to see a difference between them. Inside the diagram \( \Sigma \), some parts which may be considered as a diagram can be isolated in such a way that if \( \Pi(\alpha, \beta), \Omega(\alpha, a), \Delta(\alpha, a, b) \) denote, respectively, the diagrams of Figures 5-7, it is natural to write

\[
\Sigma(\alpha, \beta, \gamma, \alpha, \beta, c) = \Omega(\alpha, \Delta(\beta, \Omega(\gamma, a), \Pi(b, c))).
\]

Nothing of this kind can be done for what concerns \( \Theta \); the same happens for the entire infinite class of similar diagrams

\[
\Omega(\Theta, \Omega, \Omega, \ldots, \Omega, \ldots),
\]

whose formation rule can be easily imagined.

Let us say that while \( \Sigma \) is decomposable according to subdiagrams \( \Pi, \Omega \) and \( \Delta \), the diagrams of the type \( \Theta \) are not decomposable. From the last consideration, which should be obvious to anyone who tries to isolate with a
Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

Corrado Böhm
Technology Programming Logics

λ
25 January 2013 15 / 25

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all “higher level” programming languages (i.e. everything except, perhaps, plain machine code).

At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer’s activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that is in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the “making” of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our power to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible. Let us consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion we assume that these actions are independent of each other), the program text is a suitable text in the text a “textual index.”

When we include conditional clauses (if B then A), alternative clauses (if B then A1 else A2), choice clauses as introduced by C. A. R. Hoare (case[] of (A1, A2, ..., A,n), conditional expressions as introduced by J. McCarthy (B1 = E1, B2 = E2, ..., Bn = En), the facts remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the depth of procedure calling.

Let us now consider repetition clauses (like, while B repeat A or repeat A until B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don’t want to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as “induction” makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called “dynamic index,” inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer’s control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n, say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one.

The unbribled use of the go to statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood. With the go to statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (a kind of activity index).

The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points where n, the number of persons in the room minus one.

The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one’s program. One can regard and appreciate the clauses considered as bridging its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g., abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to
recently I discovered why the use of the `go to` statement has such disastrous effects, and I became convinced that the `go to` statement should be abolished from all “higher level” programming languages (i.e. everything except, perhaps, plain machine code).

... have proved the (logical) superfluousness of the `go to` statement.
1968:

Böhm's theorem on the equivalence of programs represented by $\lambda$-terms in $\beta$-$\eta$-normal form is decidable. Two syntactically different $\beta$-$\eta$-normal forms can be separated by applying them to the same sequence of $\lambda$-terms. Two terms of $\lambda$-calculus having syntactically different normal forms with respect to $\beta$-$\eta$-reduction cannot be consistently equated.
1968: Böhm’s theorem
1968: Böhm’s theorem

Equivalence of programs represented by $\lambda$-terms in $\beta$-$\eta$-normal form is decidable
1968: Böhm’s theorem

equivalence of programs represented by $\lambda$-terms in $\beta$-$\eta$-normal form is decidable

two syntactically different $\beta$-$\eta$-normal forms can be separated by applying them to the same sequence of $\lambda$-terms
1968: Böhm’s theorem

equivalence of programs represented by $\lambda$-terms in $\beta$-$\eta$-normal form is decidable

two syntactically different $\beta$-$\eta$-normal forms can be separated by applying them to the same sequence of $\lambda$-terms

two terms of $\lambda$-calculus having syntactically different normal forms with respect to $\beta$-$\eta$-reduction cannot be consistently equated
Alcune proprietà delle forme $\beta\eta$-normali nel $\lambda K$-calcolo
Part I: TOWARDS THE THEORY

Chapter 1: Introduction.

Chapter 2: Conversion.

Chapter 3: Reduction.

Chapter 4: Theories.

Chapter 5: Models.

Part II: CONVERSION

Chapter 6: Classical Lambda Calculus.

Chapter 7: The Theory of Combinators.

Chapter 8: Classical Lambda Calculus (Continued)

Chapter 10: Bohm Trees.

Part III: REDUCTION

Chapter 11: Fundamental Theorems.

Chapter 13: Reduction Strategies.

Appendix A: Typed Lambda Calculus.
Böhm-out technique

\[
\text{SELECT}(k, i) = \begin{array}{c}
1 \\
\cdots \\
i \\
\cdots \\
k
\end{array}
\]

\[
\text{ROTATE}(k) = \begin{array}{c}
1 \\
\cdots \\
k \\
a
\end{array}
\]

\[
\text{ROTATE}(k) = \begin{array}{c}
1 \\
\cdots \\
k \\
a
\end{array}
\]
Böhm-out technique
Böhm-out technique
Böhm-out technique

\[ a \leftarrow \text{ROTATE}(3) \]
Böhm-out technique
Böhm-out technique

\[ b \leftarrow \text{SELECT}(3, 2) \]
Böhm-out technique
Böhm-out technique

e ← \text{SELECT}(3, 1)
Böhm-out technique

\[ \triangle c \quad \triangle d \]
Böhm’s theorem is constructive
Corrado Böhm with Combinators

Corrado Böhm

courtesy of Carol Hindley

Technology Programming Logics λ

25 January 2013 25 / 25
Corrado Böhm with Combinators

courtesy of Carol Hindley