Haskell before Haskell. Curry's contribution to programming (1946–1950)*

Liesbeth De Mol[1]**, Maarten Bullynck[2], and Martin Carlé[3]

[1] Center for Logic and Philosophy of Science, University of Ghent, Blandijnberg 2, 9000 Gent, Belgium
elizabeth.demol@ugent.be
[2] Université Paris VIII, Vincennes Saint-Denis
maarten.bullynck@kuttaka.org
[3] Κυψέλη, 15 Οδός Λευκάδος, 11362 Αθήνα, Greece
mc@aiguphonie.com

Abstract. This paper discusses Curry's work on how to implement the problem of inverse interpolation on the ENIAC (1946) and his subsequent work on developing a theory of program composition (1948-1950). It is shown that Curry anticipated automatic programming and that his logical work influenced his composition of programs.

Key words: ENIAC, Haskell B. Curry, combinators, program composition, history of programming

1 Introduction

In 1946, the ENIAC (Electronic Numerical Integrator and Computer) was revealed to the public. The machine was financed by the U.S. army and thus it was mainly used for military purposes like e.g. the computation of firing tables. It was the first U.S. electronic, digital and (basically) general-purpose computer. ENIAC did not have any programming interface. It had to be completely rewired for each new program. Hence, a principal bottleneck was the planning of a computation which could take weeks and it became obvious that this problem needed to be tackled at some point.
In the middle of 1945 a Computations Committee was set up at Aberdeen Proving Ground to prepare for programming the new computing machines [11]. The committee had four members: F.L. Alt, L.B. Cunningham, H.B. Curry and D.H.

* This paper is a contribution to the ENIAC NOMOI project. We would like to thank G. Alberts for having given us the opportunity to present and discuss parts of this paper at CHOC (Colloquium History of Computing), University of Amsterdam. This is the authors' version of a paper that was previously published as: L. De Mol, M. Bullynck, M. Carlé, *Haskell before Haskell. Curry's contribution to programming (1946–1950)*, F. Ferreira, B. Löwe, E. Mayordomo, L.-M. Gomze (eds.), Computability in Europe 2010, Lecture Notes in Computer science, vol. 6158, 108–117. The original publicaion is available at www.springerlink.com/index/826829K30064K537.pdf
** Postdoctoral Fellow of the Fund for Scientific Research – Flanders (FWO)

Lehmer. Curry and Lehmer prepared for testing the ENIAC, Cunningham was interested in the standard punch card section and Alt worked with Bell and IBM relay calculators. Lehmer's test program for the ENIAC was already studied in [9]. Here, we will deal with Haskell B. Curry's implementation of a very concrete problem on the ENIAC, i.e., inverse interpolation, and how this led him to develop what he called a theory of programming.

Of course, Curry is best known as a logician, especially for his work on combinators [1]. The fact that, on the one hand, a logician got involved with ENIAC, and, on the other hand, started to think about developing more efficient ways to program a machine, makes this a very early example of consilience between logic, computing and engineering. Curry's work in this context has materialized into three reports and one short paper. The first report [5], in collaboration with Willa Wyatt, describes the set-up of inverse interpolation for the ENIAC. The second and third report [2,3] develop the theory of program composition and apply it to the problem of inverse interpolation. A summary of these two reports is given in [4]. Despite the fact that the reports [2,3] were never classified, this work went almost completely unnoticed in the history of programming as well as in the actual history and historiography. Only Knuth and Pardo have discussed it to some extent [8, pp. 211–213].

2 A study of inverse interpolation of the ENIAC

Together with Willa Wyatt, one of the female ENIAC programmers, Curry prepared a detailed technical report that presented a study of inverse interpolation of [sic] the ENIAC. The report was declassified in 1999. This problem is explained as follows: "Suppose we have a table giving values of a function $x(t)$ [...] for equally spaced values of the argument t. It is required to tabulate t for equally spaced values of x." [5, p.6] This problem is highly relevant for the computation of firing tables. Indeed, given the coordinates of the target, it allows to compute the angle of departure of a bomb as well as the fuse time.

The set-up is amply described in the report, with over 40 detailed figures of wirings for parts of the program and many technical notes on exploiting hardware specificities of the ENIAC. Notwithstanding this concreteness, Curry and Wyatt have more general ambitions, viz. "the problem of inverse interpolation is studied with reference to the programming on the ENIAC *as a problem in its own right* [m.i.]." [5, p.6] Thus, even though the practical goal was to compute firing tables, the aim was to consider the problem and its implementation in its generality: "[The] basic scheme was not designed specifically for a particular problem, but as a basis from which modifications could be made for various such problems." [5, p.54] For example, the scheme could be modified to provide for composite interpolation. In total, four modifications are discussed in detail and 21 more are suggested in the conclusion.

In their analysis Curry and Wyatt give a method for structuring a program. They distinguish between *stages* and *processes*. Processes are major subprograms that can be repeated over and over again. A process consists of pieces called stages.

Each stage is a program sequence with an input and one or more outputs. About these stages Curry and Wyatt point out : "The stages can be programmed as independent units, with a uniform notation as to program lines, and then put together; and since each stage uses only a relatively small amount of the equipment the programming can be done on sheets of paper of ordinary size." [5, p.10] Thus, hand-writing replaces hand-wiring since modifications of the problem can be obtained by changing only these stages without the need to modify the complete (structure of the) program. In this sense, the concept of stages adds to the generality of their study.
Curry's experience with putting the inverse interpolation on the ENIAC was the starting point for a series of further investigations into programming. Moreover, he would use the interpolation problem as a prototypical example for developing and illustrating his ideas. As Curry wrote later: "This [interpolation] problem is almost ideal for the study of programming; because, although it is simple enough to be examined in detail by hand methods; yet it is complex enough to contain a variety of kinds of program compositions." [4, p. 102]

3 On the composition of programs

In 1949 and 1950 Curry wrote two lengthy reports for the Naval Ordnance that proposed a theory of programming different from the Goldstine-von Neumann (GvN hereafter) approach [6] (see Sec. 3.4) that "evolved with reference to inverse interpolation" [2, p.7]. The motivation and purpose of the reports is made absolutely clear [2, p.5]:

> In the present state of development of automatic digital computing machinery, a principal bottleneck is the planning of the computation [...] Ways of shortening this process, and of systemizing it, so that a part of the work can be done by the technically trained personnel or by machines, are much desired. The present report is an attack on this problem from the standpoint of composition of computing schedules [...] This problem is here attacked theoretically by using techniques similar to those used in some phases of mathematical logic.

Note that Curry explicitly considered the possibility of mechanizing the technique of program composition and thus made the firsts steps towards automatic programming, i.e., compiling. As G.W. Patterson stated in a 1957 (!) review on [4]: "automatic proramming is anticipated by the author" [10, p. 103]

3.1 Definitions and assumptions

Unlike modern programming languages which are machine-independent, Curry chose to build up his theory on the basis of a concrete machine, viz. the IAS computer that had von Neumann architecture. He leaves "the question of ultimate generalization [i.e. machine-independence] until later", but does make an idealization of the IAS machine. Curry introduces several definitions related to

his idealized IAS machine. Furthermore, he makes certain assumptions that are used to deal with practical problems of programming.
The machine has 3 main parts: a memory, an arithmetic unit (consisting mainly of accumulators A) and a control (keeping track of the active location in memory). The memory consists of locations and each location can store a word and is identified by its location number. There are two types of words: *quantities* and *orders*. An order consists of three main parts: an operator and two location numbers, a datum location and an exit location. There are four "species" of orders: arithmetical, transfer, control and stop orders. Roughly speaking, a transfer order moves a word from one position to another, a control order changes location numbers. A *program* is an assignment of $n+1$ words to the first $n+1$ locations. A program that exists exclusively of orders resp. quantities is called an order program resp. quantity program. A *normal program* X is a program where the orders and quantities are strictly separated into an order program A and a quantity program C with $X = AC$.
Note that it is impossible to tell from the appearance of a word, if it is a quantity or an order. Curry considers this as an important problem [2, p.98]:

> [...] from the standpoint of practical calculation, there is an absolute separation between [quantities and orders]. Accordingly, the first stage in a study of programming is to impose restrictions on programs in order that the words in all the configurations of the resulting calculation can be uniquely classified into orders and quantities.

Curry introduces a classification of the kinds of programs allowed. In this context, the *mixed arithmetic order* is crucial. This is an arithmetical operation that involves an order as datum. For example, an important use of mixed arithmetic orders is looking up consecutive data in a table . Here, it is employed to effectively calculate with location numbers. To enable this, Curry adds the *table condition*, i.e. one may add one to a location number to get a next value. Ultimately, this results in the notion of a *regular program*, which is either a primary program or a secondary program that satisfies the table condition whereby a primary program never involves mixed arithmetic orders, but a secondary program at least one. In any case, the calculation has to terminate.

3.2 Steps of program composition

Throughout the two main reports, Curry provides several steps for the process of program composition. As noted by Curry, these steps are suitable for mechanization. In this respect, the techniques provided in these steps can be considered as techniques for compiling.

a. Transformations and replacement The first step in program composition as discussed by Curry concerns the definition of the different transformations needed in order to attain compositions on the machine level. Let $X = M_0M_1M_2...M_p; Y = N_0N_1N_2...N_q; Z = L_0L_1L_2...L_r$ be three regular programs

with N_0, M_0, L_0 initiating orders, $T(k) = k'$ with $k \leq m$, $k' \leq n$ some numerical function. Given a program X such that $p = m$ then $\{T\}(X)$ computes the program Y with $q = n$ such that:

$$\{T\}(X) = \begin{cases} N_0 = M_0 \\ N_i = M_{k_i} \text{ if there are } \{k_1, ..., k_i, ..., k_t\} \text{ for which } T(k_j) = i, i > 0 \ (*) \\ \qquad\qquad \text{and if } t > 1, \exists f \text{ such that } f(k_1, ..., k_t) = k_i \\ N_i = J \quad \text{if there is no } k \text{ such that } T(k) \text{ is defined} \end{cases}$$

where J is a blank. $\{T\}(X)$ is called a *transformation of the first kind*. It boils down to a reshuffling of the words in X, where it is not necessary that every word in X reappears in Y. Note that the function f is needed in order for $\{T\}$ to be uniquely defined.
A *transformation of the second kind* (T)(X) gives the Y such that $q = p$ and every word $N_i \in Y$ is derived from a word $M_i \in X$ by replacing every location number k in every order M_i of X by $T(k)$. If M_i is a quantity then $N_i = M_i$. This results in changing the datum and exit numbers in the orders to correspond with the reshuffling from the transformation of the first kind.
Given program X and Y and θ a set of integers then the transformation $\frac{\theta}{Y}x$ called a *replacement* gives the program Z of length $r + 1$ where $r = p$ if $p \geq q$ else $r = q$ and for each $L_i \in Z$, $0 \leq i \leq r$:

$$L_i = \begin{cases} M_0 \text{ if } i = 0 M_i \text{ if } i \notin \theta, i \leq p \\ N_i \text{ if } i \leq q \text{ and } (i \in \theta \text{ or } i > p) \\ J \text{ if } i \in \theta, i > q \end{cases}$$

Thus, a replacement is a program made up from two programs by putting, in certain locations of one program, words from corresponding locations in the other program. Curry then gives definitions for transformations of the second kind with replacement, defined as $\{\frac{\theta,T}{Y}\} = \frac{\theta}{Y}(\{T\}(X))$ and transformations of the third kind. This last class concerns transformations that result from combining transformations of the first and second kind with replacements:

$$\begin{aligned} [T](x) &= \{T\}(T)(x) \\ [\tfrac{T}{Y}] &= \{\tfrac{T}{Y}\}(T)(x) \\ [\theta T](x) &= \{\tfrac{\theta T}{0}\}(T)(x) \\ S(x) = [\tfrac{\theta T}{Y}](x) &= \{\tfrac{\theta T}{Y}\}(T)(x) \end{aligned}$$

Note that 0 is a void program.
Curry goes on to show that it is possible to convert every regular program into a normal program under certain restrictions.

Compositions Using these three kinds of transformations, Curry embarks on the study of diverse program compositions. He considers the following compositions: simple substitution, multiple substitution, reduction to a single quantity program, loop programs and finally complex programs. This includes to find the right combination of the different transformations and to determine the numerical functions $T(k)$ used in the transformations. Here, we will just discuss how

a simple substitution can be achieved.
Let $Y = BC$ and $X = AC$ be normal programs, with α, β, γ the respective lengths of A, B, C, m is the location number of a word M in A where the program Y is to be substituted and n the location number of the starting location of B. The following numerical functions are needed:

$$T_1(k) = \begin{cases} k & \text{for } 0 < k < m \\ m+n-1 & \text{for } k = m \\ k+\beta-1 & \text{for } m < k \leq \alpha+\gamma \end{cases}$$

$$T_2(k) = \begin{cases} m+k-n & \text{for } n \leq k \leq n+\beta \\ \alpha+k-n & \text{for } n+\beta < k \leq n+\beta+\gamma \end{cases}$$

Then with θ the set of k's with $n \leq k < m+\beta$, the simple substitution of Y in X at M is given by $Z = [\frac{\theta T_1}{[T_2](Y)}](x)$. Consider that, if M is an output of X, the simple substitution of Y in X results in the program composition Z, denoted as $Z = X \to Y$.

Basic programs In [3], Curry starts with an analysis of programs into their simplest possible programs called basic programs. Curry gives several reasons for the significance of determining basic programs [4, p.100]:

> (1) Experience in logic and in mathematics shows that an insight into principles is often best obtained by a consideration of cases too simple for practical use [...] (2) It is quite possible that the technique of program composition can completely replace the elaborate methods of Goldstine and von Neumann [...] (3) The technique of program composition can be mechanized; if it should prove desirable to set up programs [...] by machinery, presumably this may be done by analyzing them clear down to the basic programs

A basic program consists of a single order plus their necessary outputs and data [3, p.22]. Therefore, two important concepts are *locatum* and *term*. A locatum is a variable that designates a word in the machine, a term is a word constructed by the machine from its locata at any stage. If ξ is a term and λ a locatum, then $\{\xi : \lambda\}$ is a program that calculates the term ξ and stores it in the locatum λ, i.e. yielding what we would nowadays call an assignment. Given a predicate Φ constructed by logical connectives from equalities and inequalities of terms, then $\{\Phi\}$ designates a discrimination which tests whether Φ is true or not. The assignment program can be analyzed in more basic programs using the equation:

$$\{\phi(\xi) : \lambda\} = \{\xi : \mu\} \to \{\phi(\mu) : \lambda\}$$

where ϕ is some function.
Curry defines several functions and orders to list all the basic programs. First, he gives a list of arithmetic functions:

$$\pi_0(t) = +t \quad \pi_1(t) = -t$$
$$\pi_2(t) = +|t| \; \pi_3(t) = -|t|$$

Besides these arithmetic functions, Curry defines two mixed arithmetic orders, i.e., $d(*)$ and $e(*)$ where $d(*)$ is an order that reads the location number of its own datum into the accumulator and $e(*)$ an order that reads the location number of its own exit number into the accumulator. Similarly, $d(x)$ designates the location number of the datum of order x, and $e(x)$ the location number of the exit number of the order x. Some more additional orders are the conditional and unconditional jump and the stop order.
On the basis of these most basic orders Curry defines his final set of basic orders, of which some are the result of simple compositions of the original set of basic orders. This is done for practical reasons. In this way, Curry replicates all orders of the GvN coding plus adds some more that he deems useful. E.g.,'clear', absent in the GvN coding, is $\{0 : A\}$; a conditional jump $\{A < 0\}$; $\{A + \pi_2(R) : A\}$, add the absolute value of the register R to the accumulator A; and the partial substitution order, S_p in the GvN coding, becomes $\{A : d(x)\}$.

Analysis and synthesis of expressions How can one convert expressions such as $(x + 1)(y + 1)(z + 1)$ to a composition of basic programs? This problem is treated in the last three chapters of [3]. Ch. III deals with arithmetic programs, Ch. IV with discrimination programs and Ch. V with secondary programs.
In Ch. III Curry gives all the necessary rules, which are based on inductive definitions, for transforming fairly straightforward arithmetic expressions (expressions that do not need control orders or discriminations) to a composition of basic programs. We will not go into the details of this here. However, to clarify what is done, the rules allow to convert the expression $(x + 1)(y + 1)(z + 1)$ to:

$$\{x : A\} \to \{A + 1 : A\} \to \{A : w\} \to \{y : A\} \to \{A + 1 : A\} \to \{A : R\} \to \\ \{wR : A\} \to \{A : w\} \to \{Z : A\} \to \{A + 1 : A\} \to \{A : R\} \to \{wR : A\}$$

Curry only gives a partial treatment of discrimination and secondary programs. For discrimination programs, this includes procedures to handle elementary propositions that are used as predicates. For secondary programs, some specific programs, such as 'tabulate', are synthesized.

3.3 Notation

As is clear from 3.2, Curry develops a notation for programs different from the flow charts discussed by Goldstine and von Neumann. We already introduced the notation for compositions and assignments. Another notation is that for conditionals, i.e., $X \to Y \ \& \ Z$ means that X is either followed by Y or Z depending on the output of X.
Curry's notation allows to code several expressions. Here is the notation for $n!$:

$$n! = \{1 : A\} \to \{A : x\} \to \{A : i\} \to (Y \to (It(m, i) \to I \ \& \ O_2))$$
$$Y = \{ix : x\}$$
$$It(m, i) = \{i : A\} \to \{A + 1 : A\} \to \{A : i\} \to \{m : A\} \to \{A - i : A\} \to \{A < 0\}$$

where I represents the input to Y, A is an accumulator of the arithmetical unit and O_2 is some output channel.

3.4 A Comparison with the Goldstine-von Neumann approach

Originally, Curry feared that the third part of the GvN reports [6, Part III] *Combining routines* might overlap with his own work on composition of programs [2, p. 6]. Yet by 1950, Curry had seen this report and promptly concluded that his approach differed significantly from GvN's [3, pp.3–4]. Perhaps, the chief difference in their respective approaches is the fact that Curry's is the more systematic one. There are several indications of this finding. For example, the classification of programs (Sec. 3.1) or their analysis into basic programs (Sec. 3.2) with the explicit goal of synthesizing and analyzing expressions. Curry also notes that the GvN approach is not suitable for automatic programming:

> [GvN give] a preparatory program for carrying out on the main machine a rather complicated kind of program composition. But one comment seems to be in order in regard to this arrangement. The scheme allows certain data to be inserted directly into the machine by means of a typewriter-like device. Such an arrangement is very desirable for trouble-shooting and computations of a tentative sort, but for final computations of major importance it would seem preferable to proceed entirely from a program or programs recorded in permanent form, not subject to erasure, such that the computation can be repeated automatically [...] on the basis of the record.' [3, p. 4]

To this Curry adds the following footnote in order to strengthen his point: "It is said that during the war an error in one of the firing tables was caused by using the wrong lead screw in the differential analyser. Such an error would have been impossible if the calculation had been completely programmed." Clearly, this remark indicates that Curry was not only highly aware of the significance of a digital approach but also of the possible merits of higher-level programming and the prospects of automated program optimization.
A second important point made by Curry, concerns the fact that he considered his notation as more suited for automatic programming than the flow chart notation of the GvN approach [2, p.7]:

> The present theory develops in fact a notation for program construction which is more compact than the "flow charts" of [6, Part I]. Flow charts will be used [...] primarily as an expository device. By means of this notation a composite program can be exhibited as a function of its components in such a way that the actual formation of the composite program can be carried out by a suitable machine.

In general, one may say that the cardinal difference of Curry's approach is due to the fact that it is based on a systematic logical theory, whereas this is not the case for the GvN-approach.

3.5 Logic and the theory of program composition

In [3, p.5], Curry makes explicit that he regards his approach as a logician's:

> The objective was to create a programming technique based on a systematic logical theory. Such a theory has the same advantages here that it has in other fields of human endeavor. Toward that objective a beginning has been made.

Curry's guiding ideas while doing research in mathematical logic are the same as those that guided his theory of composing programs [1, p. 49]:

> [I]t is evident that one can formalize in various ways and that some of these ways constitute a more profound analysis than others. Although from some points of view one way of formalization is as good as any other, yet a certain interest attaches to the problem of simplification [...] In fact we are concerned with constructing systems of an extremely rudimentary character, which analyze processes ordinarily taken for granted.

Curry's analysis of basic orders and programs into the composition of more elementary orders (cfr. Sec. 3.2, pp. 6–7) is a clear example of this approach. In fact, Curry's analysis is at some points directly informed by his theory of combinators. In Sec. 3.2, p. 4 we explained the several kinds of transformations necessary to attain composition on the machine level. Remarkably, Curry makes use of functions rather than of combinators. However, he does rely on certain concepts of combinators in his explanation of how regular programs can be transformed into normal programs. This is related to the condition $(*)$ used in the definition of transformations of the first kind. There, a projection function is needed to select a value k_j. Then, the function $T(k_s) = i$ is called K-free, if it has at least one solution, W-free, if it has no multiple solutions, and C-free, if T is monotone increasing, whereby $K = \lambda.xy.x, W = \lambda xy.xyy$ and $C = \lambda xyz.xzy$.
On top of this analysis, Curry superposes a calculus of composition. While dealing with the synthesis of programs, Curry arrives at the problem that a program can be synthesized in more than one way. Hence, a calculus of equivalences becomes accessible. [4, p.100]:

> When these processes [of composition] are combined with one another, there will evidently be equivalences among the combinations. There will thus be a calculus of program composition. This calculus will resembles, in many respects the ordinary calculus of logic. It can be shown, for example, that the operation "→" is associative. But the exact nature of the calculus has not, so far as I know, been worked out.

Curry provides several equivalent programs and points out that finding shorter programs equivalent to longer ones is important with respect to memory which was very expensive at the time. His approach to study programs and their variations antedates by nearly a decade the Russian study of equivalent programs [7].

4 Discussion

The ENIAC was not just one of the first electronic computers, it was also the place where ideas originated that are still relevant today. While Lehmer's work

on ENIAC announced the use of the computer as a tool for experimental mathematics, it was Curry's work to anticipate parts of a theory of programming. Surely, Curry was aware that his suggestions would create a certain distance between man and machine, a distance which was almost completely absent with ENIAC, and only marginally present in the GvN approach. Thus, he developed his theory of compositions in a way that allowed for mechanization and consequently automatic programming. As a result, a theory of programming emerged from the practical problems related to hand-wiring thereby also transgressing the limitations of a notation of programs still bound to hand-writing and manual composition. This, however, does not mean that Curry completely abstracted away from the concrete context he started from. On the contrary, his work is a good example of the mutual interactions between logic and the machinery of computing. To draw a conclusion, Curry's work reveals an interesting balance between, on the one hand, the highest awareness of the practicalities involved with the hardware and concrete problems such as inverse interpolation, and, on the other hand, the general character of a theory of compositions that becomes a calculus in its own right but still is evidently rooted in these practicalities.

References

1. Haskell B. Curry. The combinatory foundations of mathematical logic. *The Journal of Symbolic Logic*, 7(2):49–64, 1942.
2. Haskell B. Curry. On the composition of programs for automatic computing. Technical Report 9805, Naval Ordnance Laboratory, 1949.
3. Haskell B. Curry. A program composition technique as applied to inverse interpolation. Technical Report 10337, Naval Ordnance Laboratory, 1950.
4. Haskell B. Curry. The logic of program composition. In *Applications scientifiques de la logique mathématique, Actes du 2e Coll. Int. de Logique Mathématique, Paris, 25-30 août 1952, Institut Henri Poincaré*, pages 97–102, Paris, 1954. Gauthier-Villars.
5. Haskell B. Curry and Willa A. Wyatt. A study of inverse interpolation of the Eniac. Technical Report 615, Ballistic Research Laboratories, Aberdeen Proving Ground, Maryland, 1946.
6. Herman H. Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument. vol. 2, part I,II and III, 1947-48. Report prepared for U. S. Army Ord. Dept. under Contract W-36-034-ORD-7481.
7. Iuri I. Ianov. On the equivalence and transformation of program schemes. *Communications of the ACM*, 1(10):8–12, 1958.
8. Donald E. Knuth and Luis T. Pardo. Early development of programming languages. In J.Howlett, N. Metropolis, and G.-C. Rota, editors, *A History of Computing in the Twentieth Century*, pages 197–274, New York, 1980. Academia Press.
9. Liesbeth De Mol and Maarten Bullynck. A week-end off: The First Extensive Number-Theoretical computation on the ENIAC. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *Logic and Theory of Algorithms, CIE2008*, volume 5028 of *LNCS*, pages 158–167. Springer, 2008.
10. George W. Patterson. Review of "The logic of program composition by H.B. Curry". *The Journal of Symbolic Logic*, 22(1):102–103, 1957.
11. Henry S. Tropp. Franz Alt interview, September 12, 1972.