Haskell before Haskell. An alternative lesson in practical logics of the ENIAC.

Liesbeth de Mol, Martin Carlé, Maarten Bullynck

Haskell before Haskell. An alternative lesson in practical logics of the ENIAC.[*]

Liesbeth De Mol[†] Martin Carlé[‡] Maarten Bullynck[§]

December 29, 2012

Abstract

This article expands on 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's hands-on experience with the ENIAC on the one side and his acquaintance with systems of formal logic on the other, were conducive to conceive a compact "notation for program construction"which in turn would be instrumental to a mechanical synthesis of programs. Since Curry's systematic programming technique pronounces a critique of the Goldstine-von Neumann style of coding, his "calculus of program composition"not only anticipates automatic programming but also proposes explicit hardware optimisations largely unperceived by computer history until Backus' famous ACM Turing Award lecture (1977). The cohesion of these findings asks for an integrative historiographical approach. An appendix gives, for the first time, a full description of Curry's arithmetic compiler.

This article frames Haskell B. Curry's development of a general approach to programming. Curry's work grew out of his experience with the ENIAC in 1946, took shape by his background as a logician and finally materialised into two lengthy reports (1948 and 1949) that describe what Curry called the 'composition of programs'. Following up to the concise exposition of Curry's approach that we gave in [32], we now elaborate on technical details, such as diagrams, tables and compiler-like procedures described in Curry's reports. The ends of studying such details is not solely about adding to a historical picture but is a prerequisite in order to fundamentally question sedimented historical narratives. Capital in this respect, is our in-depth discussion of the transition from Curry's concrete 'hands-on' experience with the ENIAC towards a general theory of combining programs. Our results contest the decisive role traditionally

[*]This paper is a contribution to the ENIAC NOMOI project.

[†]Center for Logic and Philosophy of Science, University of Ghent, Blandijnberg 2, 9000 Gent, Belgium, e-mail: elizabeth.demol@ugent.be. This research was supported by the Fund for Scientific Research – Flanders (FWO)

[‡]Κυψέλη, 15 Οδός Λευκάδος, 11362 Αθήνα, Greece, e-mail: mc@aiguphonie.com

[§]Université Paris VIII, Vincennes Saint-Denis, e-mail: maarten.bullynck@kuttaka.org ascribed to von Neumann, Goldstine and Burks in laying the logical foundations of computer science. It turns out that their series of technical reports describing methods of tackling the "coding and planning of problems" [19] for computers with the help of 'flow-charts' is neither highly systematic, nor unrivalled or, for that reason, canonical.

1 The ENIAC Experience

The ENIAC (Electronic Numerical Integrator and Computer) is one of the first computers worldwide. It was revealed to the public in 1946 and publicised widely in the scientific and popular press. In the journal *Nature* it was described as "an electronic computing machine" [22], in *Popular Science* the ENIAC was announced somewhat catchier as "lightning strikes mathematics" [35]. Until the mid-fifties, the ENIAC would remain the fountainhead of the new general-purpose digital electronic devices that we nowadays call computers. Although originally conceived as a calculator for ballistic tables, the outstanding role of ENIAC as a de facto operating and publicly known computing machine created an unparalleled pole of interest which attracted diverse disciplines to research and explore the 'arrival of computation' after the Second World War. This circumstance enticed historians to either overemphasise the advent of this very machine as the inauguration of the era of digital computing [5, p. 15] or to deconstruct the "standard narrative, which begins with the abacus and then follows a by now familiar sequence of devices - mechanical and electromechanical, analog and digital - all converging on ENIAC" [28, p. 8]. However, by reexamining rather technical details we may reach another view on the significance of this early computer which overcomes the always difficult historiographical opposition of a machine-centred history versus its discursification into a patchwork of diverse disciplines. That is, an 'integrative' approach which is sensitive to both the hardware details of the machine and the human's knowledge, needs and experience. This approach perfectly agrees with the insight that a computer "could not dictate how it was to be used", but may not postpone investigations "until people figured out what to do with it"[27, p. 123]. "Impact" sets in before that. For instance when truly studying the ENIAC in its actual use, alternative ways of engaging with the conditions imposed by the hardware at hand are disclosed. In sum, we might think of computers as the locus of 'habitus' which shapes dispositives.

"ENIAC, as is well known, was the first general-purpose electronic digital computer to be designed, built and successfully used"[17, p. 13]. No less known is that it had not the alluring simplicity of the EDVAC design, nor had it any programming interface. Instead, it exposed the architecture of a highly parallel and modular machine and for each new program, cables had to be plugged in the right devices, adaptors used on the right connections, dials and switches set for the right values etc. Thus, it need not surprise that the planning of a computation, not only the translation of the mathematics into a general scheme but also the realisation of the scheme as a combination of cables and switch-settings, could take weeks and had to be done with utmost care. This frequent and time-consuming re-wiring of the machine—or the so-called 'local' or 'direct-programming' of the ENIAC—constituted one of the severe 'bottlenecks' of doing things with 'programmable calculators', and that this fundamental problem had to be tackled at some point already sprang to the minds of ENIAC's inventors and "users" while building and using the machine. Hence the ENIAC not only flashed upon mathematics but also posed a provocation to the logics of programming.

To examine the machine and face these issues, a Computations Committee was set up at Ballistic Research Laboratory (BRL) at the Aberdeen Proving Ground in the middle of 1945 [36]. The committee had four members: F.L. Alt, L.B. Cunningham, H.B. Curry and D.H. 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 [31] and [3]. Here, we will deal with Haskell B. Curry's work on the problem of inverse interpolation, a principal problem in the calculation of ballistic tables, and how this led him to develop a theory of programming.

Curry is best known as a logician. He is remembered for his work on combinators [10], the verb 'to curry' and the Curry-Howard-isomorphism. The 'leitmotiv' of Curry's research on mathematical logic may be summarised by the following quote from his retiring address as the president of the Association for Symbolic Logic [10, p. 49]:

> [I]t is evident that one can formalise 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 formalisation 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 analyse processes ordinarily taken for granted.

This guiding principle will be seen at work in Curry's involvement with the ENIAC while conducting the way from a specific practice of wiring to a general theory of programming.

In combining experience with ENIAC, researching for more efficient and systematic ways to program a machine and by developing a simple most compact symbolism, Curry's contribution can be regarded as a very early example of the 20th century consilience between logic, computing and engineering.

During the years 1946 to 1950, Curry wrote three reports and one short paper. The first report [16] is a collaboration with Willa Wyatt. It describes the set-up of inverse interpolation for the ENIAC. The second and third report [11, 12] develop the theory of program composition and apply it to the problem of inverse interpolation. A summary of these two reports is given in [13]. Despite the fact that the reports [11, 12] were never classified, this work went almost completely unnoticed in the history of programming as well as in the actual history and historiographies. Only Knuth and Pardo in their history of programming languages briefly considered Curry's work [25, pp. 211–213]. Backus [2] however, only made frequent references to Curry's logical work, but equalised his efforts with those attributed to Church's lambda calculus or McCarthy's pure Lisp.

Curry himself does not seem to have ever returned to the topic after 1950. In 1950, he received a two-year Fulbright Grant to work together with Robert Feys from the university of Louvain (Belgium). This resulted in the famous *Combinatory Logic* (Vol. 1, 1958) co-authored by Curry and Feys. Although his reviews for AMS Mathematical Reviews (now MathSciNet) seem to suggest that Curry tried to keep up with the developments in programming or the use of logic in computers, he never again published on programming and concentrated mainly on his work in combinatory logic and on his philosophical work.

1.1 Curry and Wyatt: A study of programming on the ENIAC (1946)

In 1946–1947, Curry together with Willa Wyatt and Max Lotkin, drew up a series of BRL technical reports in preparation of using the ENIAC to compute ballistic tables. These were *A study of inverse interpolation on the ENIAC* (1946, Curry with Willa Wyatt); *A study of fourth order interpolation on the ENIAC* (1946, Curry with Max Lotkin) and *Inversion on the ENIAC using osculatory interpolation* (1947, Max Lotkin alone). The first report is the most interesting, since "the problem of inverse interpolation is studied with reference to the programming on the ENIAC *as a problem in its own right* [our italics]." [16, p. 6] The report was written in collaboration with Willa Wyatt, one of the all-female team of ENIAC programmers [17, p. 26], who surely provided a lot of the technical details and variations. The Curry-Wyatt report was declassified in 1999.

The main problem of the report is stated 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." [16, p.6] Given the coordinates of the target, the tables computed with the method of the report would allow to find the right angle of departure of a missile as well as the appropriate fuse time. The general problem of iteration was analysed at length and Curry finally opted for an algorithm that may not converge extremely rapidly but instead would be more stable and need less parameters. Card readings and printings (at the beginning and ending of each calculation cycle) eat up 3000 addition times, while in comparison even the slow convergent calculation takes only 60 addition times, so that in any case, "the bulk of the time [...] is taken up by card feeding." Therefore, a simpler and more stable algorithm has to take precedence over an optimally fast one —for as a lesson of the ENIAC: "a far more important consideration than speed of convergence is simplicity of programming" [16, p. 14].[1]

[1] This is completely in tune with Lehmer's observation that because of its speed, the ENIAC often calls for an "idiot approach" [3, p. 135–136].

The set-up of the computation is detailed with over 40 figures of wirings for parts of the program and contains various remarks on exploiting the peculiarities of the ENIAC's hardware. Although the practical goal was simply to compute firing tables, Curry and Wyatt aimed at a more general solution and discussed number of modifications of the problem. Hence they planned the scheme for a maximum of functions to be possibly interpolated at once, viz. to compute four values , x, y, z and w at a time. As a consequence, "the Eniac is jammed full" [16, p. 7]. The modifications had to cope with this very 'classical ENIAC-problem' of economising on a part of hardware *here*, just to be able to use it *there*. So there was always a tradeoff between the sophistication of computational procedure and the number of values or functions calculated at a time. The amount of hardware tied up in the realisation of a specific program thus afforded to make clear choices which put a limit to both the generality and effectivity of any one solution. Especially accumulators—being at the same time the main memory units and local programming controls—were always at a premium in the ENIAC.

As a result, the analysis of the report distinguishes between *stages* and *processes* of computations, where *processes* are major, repeatable subprograms grouping the logical progress within the entire problem solving procedure, whereas *stages* represent smaller, 'primitive' entities of ever-recurring computations. For example the inverse interpolation program consists of 6 major processes:

1. Set up
2. Preparation for the primary interpolation
3. Iteration
4. Secondary interpolation for y
5. Secondary interpolation for z
6. Secondary interpolation for w and closure

The first and second parts of processes 4, 5 and 6 run concurrently with 2 and 3 respectively. The sequencing is thus mainly: 1 – 2 (4a, 5a, 6a) – 3 (4b, 5b, 6b) – 6c (closure). The branching points of the main program are in processes 1 and 3. On the ENIAC, branching is done by discriminations on the sign of a number. In Curry and Wyatt's program, the master programmer controls the discriminations (or the conditionals, to use a more modern word).
Further, each *process* is broken up into *stages*. Each stage is a program sequence with an input and one or more outputs. About the analysis into stages Curry and Wyatt pointed out one important practical advantage: "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." [16, p. 10] Hence, arranging hand-written sheets of previously wired stages can replace the wiring of a complete process from scratch. The

'logic of stages' thus allows for modifications of the problem to be obtained simply by reshuffling sheets/stages without having to alter the total structure of the program. In this sense, the concept of stages forms an integral part to the generality of their case study on inverse interpolation. Moreover, "when an operation involves elements which recur more frequently than others, the more often recurring elements can be grouped into a stage by themselves, which stage can run concurrently with other stages." [16, p. 31] Clearly, since basic elements of a complex computation make up the stages and frequency indicates where stages may be introduced or repeated, the concept of stages indicates where computations can be run in parallel (concurrently). In other words, the technique developed by Curry and Wyatt not just managed but actually featured the parallel hardware-design of the original machine. However, significant improvements in speed and processing of this sort were disabled from 1948 onwards when a permanent, monolithic wiring, the so-called 'converter code' [7], serialised ENIAC. This code subjected the machine to von Neumann's serial philosophy of 'one operation at a time'—in fact an act of appropriation which "spoiled the ENIAC"[26].

Yet before this incidence of conversion, stages were explicated by the wiring diagrams themselves which served as markers in the complex set-up and as departure points for further programming. This can be gathered from the wiring diagram where each of the 12 controls of an accumulator has an input and an output that refers to a specific stage. Concerning the management of hardware resources, Curry and Wyatt noted that the wiring diagram showed "the stage numbers for a particular control" and also "what controls are still available for further programming" [16, p. 40]. Since every stage is an "autonomous piece of computation", any change can still be applied to stages, obviously within the limits of the ENIAC's hardware not used up already by a given possibly parallel process.

Curry's experience with putting the inverse interpolation on the ENIAC was a point of initiation for a series of further investigations into programming, such that he could reuse the interpolation problem as a prototypical example in his later reports. Curry was convinced that "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." [13, p. 102] This claim is amply illustrated at the end of the 1946-report. Curry and Wyatt list no less than 21 possible modifications of their program [16, p. 54–57]. These modifications can be grouped roughly in four categories:

1. Adding more parameters to the problem and 'complexifying' the mathematical procedure
2. Dropping (a) parameter(s) of the problem and simplifying the mathematical procedure
3. Alternative ways to wire processes, especially alternatives for doing (complex) discriminations

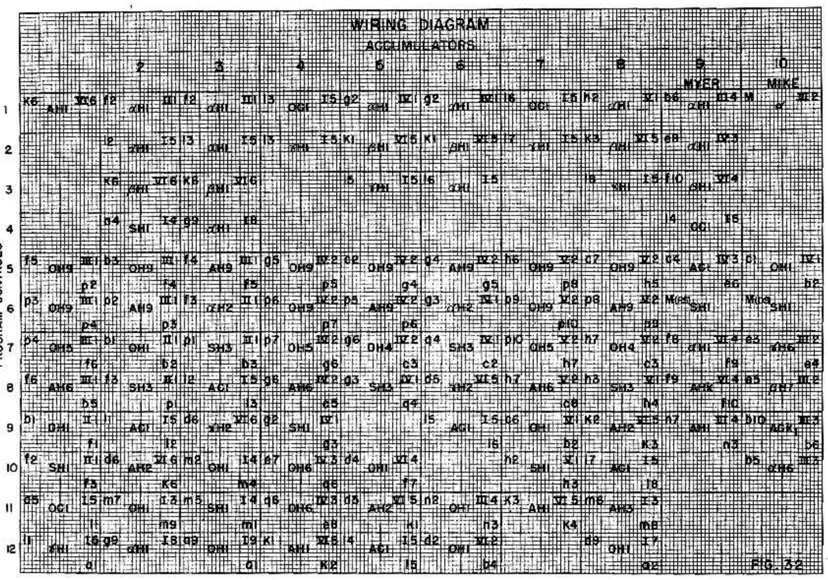

Figure 1: Curry and Wyatt's wiring diagram for the first 10 accumulators of ENIAC. Each column corresponds to an accumulator. Each accumulator has 12 controls (8 transceivers and 4 receivers) where each connects to specific stages. The twelve lines correspond to the controls.

4. Adding error checking and catching routines

The first and last group of modifications depend on feeding extra information in the basic scheme through external cards or through the (unused) function tables and by adding extra wiring while (re)using parts of the hardware not tied up by the basic scheme. As examples of the first group, one may quote the change of an interpolation formula requiring extra card information and a different wiring of controls of the accumulators (pp. 11–12 and 55); or, as another example, the transformation of a fixed parameter λ into a variable parameter representing the range of variation determined by "two transceivers free in [Accumulator N] and five in [Accumulator M]" (p. 44). The first example mentioned alters a stage and its set-up whereas the second adds a stage to the general structure.

The important idea of adding routines for internal data-checking and error-handling is a prominent one, a concept recurring in Curry's work from 1948–1950. In order to create an error channel, Curry links up several discriminations that test whether certain error conditions are met, i.e if some "error signals" were received [16, p. 23]. These signals arrive at specific positions of a free stepper located at the master programmer. When the ENIAC stops on error, the position of that stepper will indicate a certain kind of error significantly easing the affair of debugging. Also here, a stepper not yet tied up in the main scheme could be used.

It is worth noting that alternative ways of forking discriminations on the ENIAC have a non-trivial impact on wiring. While the general structure of divisions into stages remains unchanged, the consumption of output terminals and transceivers of accumulators used to perform the typical sign-discrimination (see above) could be diminished by transferring some of these classical ENIAC-discriminations to the steppers of the master programmer, thereby economising the use of working memory.[2] In traditional ENIAC parlance the, general principle applied here is "sav[ing] a dummy at the expense of an addition time" [16, p. 27]. Since the ENIAC, in fact, outran its contemporary competitors (the IBM or BTL relay calculators) by a speed-factor of 100, availability of working memory took precedence over clock cycles, such that the described wiring tricks and tradeoffs were important and determined largely the usability range of the machine. This principle, as we have seen, is also at work when adopting and simplifying sophisticated mathematical procedures, e.g. when sacrificing fast convergence behaviour for a smaller amount of parameters to compute. Consequentially, the technique of 'freeing up dummies' occurs systematically in the report of 1946 whenever modifications could not be entered into the basic scheme for lack of memory. However, what appears to be a mere mannerism of the ENIAC, will prove to be instructive for Curry's further investigations into a theory of program composition.

[2]More details on these two kinds of wiring conditional branching on the Eniac, see [3, Sec. 2.4]. If a transceiver of an accumulator is used for discrimination, or, in general, for steering another program, it is called a dummy (program).

1.2 Goldstine and von Neumann: From ENIAC to EDVAC (1946–1950)

Although Haskell Curry was involved with the ENIAC since 1946, it is not the name 'Curry' or 'Haskell' that spontaneously arises if one thinks of a logician in relation with computers in the years 1946–1950. That would rather be John von Neumann, 'Johnny the MANIAC—father of all JOHNNIACs'. As H.H. Goldstine recounted, he met von Neumann in a train station in 1944, talked with him about the ENIAC in progress and got the famous mathematician interested [20, p. 182]. Von Neumann's later involvement with ENIAC and computers is by now a standard part of the history of computers and of computing (see e.g. [1]). Yet for a significant comparison, it is mainly von Neumann's (and Goldstine's) study of setting up a process on a computer that is of relevance here.

Soon after the completion of the ENIAC, it became clear that the machine would not only be used to compute ballistic tables, but also for many other problems. Setting up a problem on the ENIAC was, unfortunately, not a simple process and took a lot of time. Therefore, in the years 1947–1948, a rewiring of the ENIAC was effectuated, changing the machine from a locally programmed, parallel machine into a centrally programmed, sequential machine [33]. This 'reconstruction' of the ENIAC was the outcome of a group process, involving the team of ENIAC engineers, the team of mathematicians from the Ballistic Research Laboratory and Los Alamos that wanted to set up programs on the ENIAC, and the all-female team of programmers of the ENIAC. Through their cooperation an instruction code for the ENIAC was developed and the ENIAC was rewired accordingly. The main ideas for this instruction code came from Adèle Goldstine (wife of Herman, programmer of the ENIAC and author of its technical description), Richard F. Clippinger (of Ballistic Research Laboratory) and John von Neumann.

A telling anecdote concerning 'instruction codes' illustrates the interplay of practical and theoretical issues, or even more to the matter, of programmers and mathematicians: For a complete list of instructions that von Neumann had proposed, a halt order was plainly missing. However, Betty Holberton, one of the programmers, convinced him of the need to include it. For a certain reason von Neumann held up the opinion, "you don't need it", yet Holberton replied, "but we are not all Von Neumann's, we will make mistakes" [37, p. 78]. Indeed, a halt order is useful for stopping the machine and analysing what was going wrong. In von Neumann's view the machine only ought to stop "naturally" that is after the program has been executed successfully.

According to von Neumann, once the translation of a problem into mathematics is achieved, the actual coding does not present any real challenge and no real problems are deemed to arise [19, part I]. Error handling is to be done before the execution of a program. Thus, this anecdote is not only an illustration of the impact of practical, everyday experiences with the ENIAC but of more theoretical and historical concern: it is characteristic of von Neumann's approach to programming. His ethos ultimately led to the above coupe of converting the ENIAC into a serial machine.

Following the experience with ENIAC, von Neumann and Goldstine quickly embarked on the conception and usage of a new machine, the EDVAC[3], that would be a sequential, stored-program computer from the start. This materialised into a series of reports describing the architecture of the new machine (hence the name 'von Neumann architecture') and how to set up programs on this new machine. The reports that interest us here, are the three volumes of *Planning and coding of problems for an electronic computing instrument* [19] which appeared in 1947 (vol. I, general principles and flowcharts; vol. II, coding examples for numerical and combinatorial problems) and 1948 (vol. III, combining routines) respectively.

Table 1: Table of basic orders [19]

Nr	**Symb.**	**Description**
1	x	Clear A and add number located at x into it
2.	x-	Clear A and subtract number at position x into it
3.	x M	Clear A and add the absolute value of the number located at x into it
4.	x -M	Clear A and subtract the absolute value of the number at position x into it
5.	x h	Add number located at x into the A
6.	x h-	Subtract number at position x into the A
7.	x hM	Add the absolute value of the number located at x into the A
8.	x h-M	Subtract the absolute value of the number at position x into the A
9.	xR	Clear R and add number located at position x into it
10.	A	Clear the A and shift the number held in the register into it
11.	x X	Clear the A and multiply the number located at position x by the number in the register
12.	x ÷	Clear register and divide the number in the A by the number located at x leaving the remainder in A and the quotient in R
13.	x C	Shift the control to the left of the order pair at position x
14.	x C'	Shift the control to the right of the order pair at position x
15.	x Cc	If the number in A ≥ 0 shift the control as in x C
16.	x Cc	If the number in A ≥ 0 shift the control as in x C'
17.	x S	Transfer the number in A to position x
18.	x Sp	Replace the left-hand 12 digits of the left-hand order located at position x by the 12 digits 9 to 20 in A
19.	x Sp'	Replace the left-hand 12 digits of the right-hand order located at position x by the 12 digits 29 to 40 in A
20.	R	Replace the content $\epsilon_0\epsilon_1\epsilon_2\ldots\epsilon_{39}$ of A by $\epsilon_0\epsilon_0\epsilon_1\epsilon_2\ldots\epsilon_{39}$

Continued on next page

[3]The EDVAC also called IAS (Institute for Advanced Study) machine when referring to the actual machine built at Princeton.

Table 1 – continued from previous page		
Nr	**Symbol**	**Description**
21.	L	Replace the content $\epsilon_0\epsilon_1\epsilon_2\ldots\epsilon_{39}$ and $\eta_0\eta_1\eta_2\ldots\eta_{39}$ of A and R by $\epsilon_0\epsilon_2\epsilon_3\ldots\epsilon_{39}0$ and $\eta_1\eta_2\eta_3\ldots\eta_{39}\epsilon_1$

In the first volume of *Planning and coding*, Goldstine and von Neumann give a table of the machine instructions of the IAS machine (see Table I). Of special interest in this table are the Sp and Sp' orders, called 'partial substitutions' by von Neumann. These orders shift the first half of a word to the second half (or vice versa). In the case the word contains an order (of length half a word) and an address (also half a word), these orders can exchange orders for addresses or vice versa. For von Neumann, the importance of these partial substitutions can hardly be over-estimated, because they allow a program to change its own code: "the machine's ability to modify its own orders [...] is absolutely necessary for a flexible code" [18, p. 31].[4] Goldstine and von Neumann then explain how to transform a problem into a program and how flowcharts can be used to do the coding:

> Since coding is not a static process of translation, but rather the technique of providing a dynamic background to control the automatic evolution of a meaning, it has to be viewed as a logical problem and one that represents a new branch of formal logics [...] this is not a mere question of translation (of a mathematical text into a code), but rather a question of providing a control scheme for a highly dynamical process, all parts of which may undergo repeated and relevant changes in the course of this process. [...] We therefore propose to begin the planning of a coded sequence by laying out a schematic of the course of C through that sequence, i.e. through the required region of the selectron memory. This schematic is the flow diagram of C. [19, vol. 2, p. 1 and 4]

The second volume of *Planning and coding* applied these general principles to other frequently occurring mathematical problems. provided detailed flowcharts for a variety of concrete problems. For the last and third volume that appeared in 1948, Goldstine and von Neumann had promised to explain how to combine several of these routines in one program. Indeed, they wanted to "avoid the need for recoding [a routine] each time when it occurs" and have a routine that can "insert [already] coded sequences as wholes" into a new program. In short, a meta-routine for substituting a subroutine in the main routine [19, vol. 3,

[3]In volume 1, orders are ordered in pairs in the memory because orders are "less than half as long as a forty binary digit number, and hence the orders are stored in the Selectron memory in pairs."

[4]This is a recurring theme in von Neumann's writings on computing, especially when talking about (self-reproducing) automata.

p. 2]. The solution to this task is a 'preparatory routine' that allows to copy a given routine to a given memory location in the main routine, automatically altering the memory positions of the program. This 'preparatory routine' only works under certain restrictions, in particular, the subroutine may not alter any of the memory locations of the main routine.

Estimating Goldstine and von Neumann's work on programming, one should say that they propose more a set of heuristic tools (such as flowcharts) and a demo-set of example routines than a theory of programming. The translation of mathematics is seen as the most important step and its implementation on a machine is treated as a derived, secondary problem—but never, as Curry states, "in it's own right". Even the 'preparatory routine' is not developed in its full generality and remains rather involved and complicated. Also typical is the absence of error routines, most obvious by the missing halt order. All in all, their strategy stuck close to the machine, not gaining enough abstraction to arrive at a more general approach worth to be called *programming*. Although the impact of the Goldstine-von Neumann reports on the history of computing and programming was considerably large, if not decisive, one cannot say that this helped constituting a systematic logical approach. That such an approach to programming was not impossible at that time, however, is shown by Curry's work.

2 On the composition of programs

In 1949 and 1950 Curry wrote two lengthy reports [11, 12] for the Naval Ordnance that proposed a theory of programming that is very different from the Goldstine-von Neumann (GvN hereafter) approach [19] (see Sec. 1.2). In fact, it will be shown that Curry's approach is much more advanced than the GvN reports as far as programming is concerned, providing a detailed theory of program compositions that is ultimately understood as a calculus of compositions and thus more akin to Backus' idea of an algebra of programs [2].
But why did Curry develop his theory of compositions and what was its purpose? This was clearly enunciated in [11, 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.

Indeed, Curry considered the preparation of programs for the machine as the principal bottleneck for the development of computing machines, a concern that directly related to Curry's ENIAC experience with its direct and local programming method. In contrast with GvN, Curry not only developed a notation for programs but also a *general* theory of program composition that could be mechanised. He thus made the firsts steps towards automatic programming, i.e., compiling. As G.W. Patterson stated in a 1957 (!) review on [13]: "automatic programming is anticipated by the author" [34, p. 103]

The sheer practical necessity to theoretically 'attack' the problem of program compositions was not the only influence of Curry's ENIAC experience. Also the 'logical nature' of the inverse interpolation problem contributed significantly to Curry's 'strategy' [11, p.7]:

> The present attack [goes] back for its fundamental philosophy to the Aberdeen report [16]. In fact, it was evolved with reference to inverse interpolation. That problem [has] shown itself to be well suited for the purpose. It is simple enough so that it is scarcely economical for a big machine; yet it has a structure showing several different kinds of compositions

Curry considers two different phases in programming a given problem, in his particular case, the inverse interpolation problem [12, p.2]:

> The first step [is] to analyze the inverse interpolation problem into its main constituent parts, and then to study the kinds of composition necessary to reconstruct the program from these main parts. [B]eyond this point there is the consideration of how these major parts [m]ay be compounded from simpler parts. This analysis, and the corresponding synthesis, occupies the later chapters of [12]

The study of the different kinds of compositions and how they can be synthesised in *general* forms the major bulk of [11] and will be discussed in Sec. 2.2. The application of the theory to the inverse interpolation problem was postponed to Chapter 1 of [12], even though it was "conceived as a part of [the] first memorandum". The analysis of the major parts into simpler parts as well as the corresponding synthesis is addressed in [12] and will be discussed in Sec. 2.3.

Note, in what follows, all technical terminology is Curry's unless indicated otherwise. Our own vocabulary, especially when slightly anachronistic, is made recognisable by single quotes.

2.1 Definitions and assumptions

Unlike modern programming languages that have as a design goal to be as machine-independent as possible, Curry chose to build up his theory with reference to a concrete machine, viz. the IAS computer with von Neumann architecture as described in [4]. He left "the question of ultimate generalization [i.e. machine-independence] until later", but did make considerable idealisations of the IAS machine. Curry introduced several definitions in targeting this idealised

IAS machine and made assumptions that were used to deal with practical problems of programming.
The target 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* was defined by Curry as an assignment of $n + 1$ words to the first $n + 1$ locations.[5] A program that exists exclusively of orders resp. quantities is called an order program resp. a 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 considered this as an important problem [11, 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 introduced 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. This corresponds, of course, to von Neumanns 'partial substitutions'. 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 added the *table condition*, i.e. it is allowed to add an integer to a location number to get a next value, but only within a limited range (the range of the table in which you look up values). Ultimately, this resulted in the notion of a *regular program*, which is either a primary program or a secondary program that satisfies the table condition where a primary program has no mixed arithmetic orders, but a secondary program at least one. In any case, the calculation has to terminate. This careful categorisation of programs allowed Curry to 'tame' or at least control the impact of mixed arithmetic orders (or 'partial substitutions'). These categorisations and restrictions contrast strongly with von Neumann's enthusiasm about changing the code of a program while running, and appears more in alliance with the caution the German computer pioneer Konrad Zuse expressed.[6]

[5]Note that Curry uses the word "program" throughout in both [11, 12]. This might be seen as a confirmation of Grier's hypothesis that the verb "to program" originated in ENIAC circles [21]. Note also that the word "assignment", as used by Curry, does not have the technical content this term has nowadays in computer science.

[6]Cf. the following quote from Zuse: "The idea of general calculating or information processing, as we say today, induced me to consider that the program, too, is information and

2.2 Steps of program composition part I

Next we discuss Curry's 'first step' in the programming of a problem, a study of the different kinds of compositions and the techniques he provided to 'reconstruct' a program from its subprograms. As Curry explained in the introduction of [11]:

> Suppose that we wish to perform a computation which is a complex of simple processes that have already been planned. Suppose that for each of these component processes we have a plan recorded in the form of what is here called a <u>program</u>, by means of a system of symbolization called a <u>code</u>. It is required to form a program for the composite computation.

2.2.1 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 $X = M_0M_1M_2...M_p$ and $Y = N_0N_1N_2...N_q$ be two regular programs with N_0, M_0 initiating orders, $T(k) = k'$ with $k \leq p$, $k' \leq q$ some numerical function. Given a program X then $\{T\}(X)$ computes the program Y such that:

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

where J is a blank. $\{T\}(X)$ is called a *transformation of the first kind.* This 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 programs X and Y and θ a set of integers then the transformation $\frac{\theta}{Y}X$

can be processed by itself or by another program.[...] In hardware it means that we not only have a controlling line going from left to right, but also from right to left. I had the feeling that this line could influence the whole computer development in a very efficient but also very dangerous way. Setting up this connection could mean making a contract with the devil. Therefore, I hesitated to do so, being unable to overlook all the consequences, the good as well as the bad. [...] My colleagues on the other side had no scruples about the problem I just mentioned. John von Neumann and others constructed a machine with a storage for all kinds of information including the program. [...] My own design for future machines on paper were more structured with instructions stored independently and special units for the handling of addresses and subroutines nested in several levels." [38, p. 616] called a *replacement* gives a program Z of length $r+1$ where $r=p$ if $p \geq q$ else $r=q$ and for each word $L_i \in Z$, $0 \leq i \leq r$:

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

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 gave 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:

$$\begin{array}{rcl} [T](x) & = & \{T\}(T)(x) \\ [\frac{T}{Y}] & = & \{\frac{T}{Y}\}(T)(x) \\ [\theta T](x) & = & \{\frac{\theta T}{0}\}(T)(x) \\ [\frac{\theta T}{Y}](x) & = & \{\frac{\theta T}{Y}\}(T)(x) \end{array}$$

Note that 0 is a void program.

This part of the theory of program composition is in fact heavily inspired by the theory of combinators on which Curry wrote his PhD in 1930 [8]. In particular, if one looks at the condition $(*)$ in the definition of $\{T\}(k)$, this a projection function needed to select a value k_j. If all k_j's are the same then:

> there is a unique normal variator (i.e. combinator corresponding to a variation of "Umwandlung") U such that $UN_1N_2 \ldots N_q \Rightarrow M_1M_2 \ldots M_p$ where $\Rightarrow$ is a reduction in the sense of combinatory logic, and the words are regarded merely as names for "entities" which can be manipulated according to the rules of the theory of combinators. [11, p. 27]

With the help of the language of combinators, Curry then rewrote the function $T(k)$ as $B^{a_n}K^{b_n} \ldots B^{a_2}K^{b_2}.B^{a_1}K^{b_1}$. Here B is the combinator of composition $(Bxyz = Bx(yz))$, K selects the argument x $(Kxy = x)$. This $T(k)$ transforms k in k if $0 < k \leq a_1$; in $k + b_1$ if $a_1 < k \leq a_2 \ldots$ in $k + b_1 + \ldots + b_n$ if $a_n < k$. The transformation $T(k)$ will accordingly be called K-free if $(*)$ has at least one solution, W-free if it has no multiple solutions and C-free, if T is monotone increasing.[7]

2.2.2 Compositions

Using these transformations of the first, second and third kind, Curry embarked on the study of diverse program compositions. He considered the following compositions: simple substitution, multiple substitution, reduction to a single

[7]Note that K, W and C are all combinators corresponding respectively to $Kxy = x$, $Wxy = Wxyy$ and $Cxyz = xzy$.

quantity program, loop programs and finally complex programs. This means: finding the right combination of the different transformations and determining 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:

$$Z = X \rightarrow Y$$

Figure 2 gives an intuitive idea of how the transformations and replacement work together in this concrete case.

To 'close' this formalism and make it into a proper algebraic structure, Curry introduced a neutral element for composition: In case B is only a stop order, then Y will be written as 0 and $X \rightarrow 0 = X$ [11, p. 38].

The discussion proceeds with the composition of multiple programs or multiple substitution. Let $X = A_0C$ and $Y_i = A_iC$ be regular programs then $Z = X \rightarrow Y_1 \& \ldots \& Y_n$ is defined as the program obtained by substituting Y_1 for O_1; ...; Y_n for O_n, with the O_i outputs of X. Curry remarked that, "with a proper renumbering of outputs" we have $Z = (\ldots((X \rightarrow Y_1) \rightarrow Y_2) \ldots \rightarrow Y_n)$ (or to put it in modern terms Z can be "curried"!). After multiple substitution (and the characterisation of the respective T-functions) follows the rather important treatment of the reduction to a single quantity program. Given $X = A_0C_0$ and $Y_i = A_iC_i$, a quantity program D can be constructed where each quantity with location number k in every C_i can be identified with some quantity in D by the functions $S_i(k)$. Taking α_i for the length of A_i, now one can define a T-function $T'_i(k) = \alpha_i + S_i(k - \alpha_i)$, or equivalently in the notation of combinatory logic, $T'_i = B^{\alpha_i} J_i$. Combining these T'_i functions with the respective T''_i-functions of multiple substitution of all A_i's on D, Curry arrived at an explicit description of $T_i (= T''_i T'_i)$-functions that describe the reduction to a single quantity program $Z = BD$. This characterisation of Z by the T's allows for "a practical procedure for constructing the program Z" [11, p. 43].

> In the first column of a sheet of columnar paper we write the successive locations of Z [...] In the second column we write the transformation T_0 by going through X and writing h in the kth row

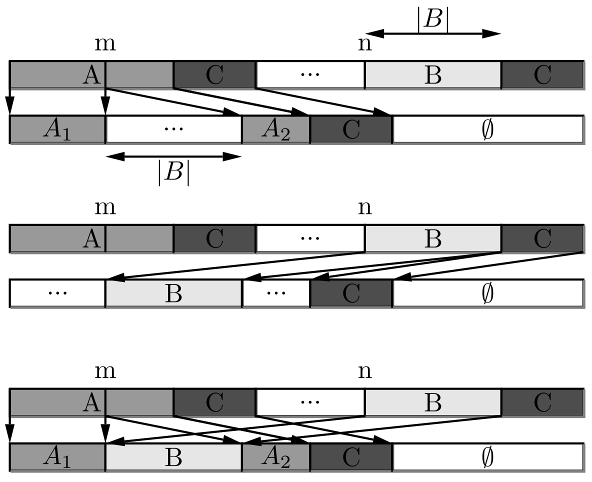

Figure 2: From top to bottom: The $T_1(X)$ transformation; the $T_2(Y)$ transformation; and finally the substitution $[\frac{\Theta T_1}{[T_2](Y)}](X)$ that substitutes Y in X at position m.

> whenever $T_0(h) = k$; if h is one of the m_i, we put in parentheses. In the $(i+2)$nd column we do the same for T_i. It is then easy to write out Z. Given any $k \leq$ [begin position of Z] there will be only one column in which there is an unparenthesized h. That column shows which program, X or Y_i, should be used. We now look up this order in the given program and make the appropriate transformation. This can be done by finding h in the appropriate column and replacing it by the corresponding k.

This manual pencil-and-paper procedure is useful for the construction of programs as Curry illustrated when he later resynthesises his inverse interpolation program using composition and basic programs. This tabular procedure is similar to our Fig. 2, but rotated over 90 degrees, with the columns representing programs. An example of the procedure is given in the table on page 28 in the next section. Of course, if a fully automatic procedure would have been feasible in 1950, the mere collation and comparison of T functions would have sufficed to find the construction of the composed program.

Finally, a last class of important program compositions is characterised, the loop, "programs which double back on themselves" or "a kind of substitution of part of X in itself" [11, p. 44–45]. To do this, one needs e.g. a T-function for which $T(m_i) = T(n_i) = j$ with $n_i < m_i$. At location m_i there is a jump or unconditional control shift to n_i.[8]

Curry concluded this report by observing that the "same programs can be constructed in different ways, [i]n other words, there are equivalences among

[8]These T-functions are no longer W-free or C-free (though still K-free), but can, "to boot if we want it" (p. 44), be regarded as such to start the program.

our programs." [11, p. 49] Indeed, as his introduction of a neutral element of composition (cfr. p. 17) made clear, Curry was aware that a calculus or an algebra of programs under the operation "composition" (or "$\rightarrow$") could be defined and studied. One such property of the algebra of composition is $X \rightarrow 0 = X$ but also associativity can be proven: $(X \rightarrow Y) \rightarrow Z = X \rightarrow (Y \rightarrow Z)$ [11, p. 50]. This aspect of Curry's composition of programs preempts the work of the Russians I. Ianov [23] and A.A. Markov Jr [29].

Given that Curry was inspired by his combinators (cfr. p. 16) or that he envisioned an algebra of program schemes, nowadays one is inclined to ask: did Curry use ideas or concepts that stem from Turing's, Church's or Post's analyses of computability?[9] The answer seems to be a plain *No*, simply because Curry's main occupation was how to synthesise a complex program from basic ones, rather than the determination of general characteristics of what could be done in principal by a machine. Curry was more interested in issues that directly relate to a theory of programming rather than a theory of computability. Curry's main example was not an abstract universal procedure, but a practical, though complex computation: the inverse interpolation routine. This maybe partly due to the addressee, namely the Naval Ordnance for which these reports were written. These people were surely more familiar with mathematical of differential equations than with mathematical logic, and probably more interested in what these new machines are capable of in the field of ballistics.

2.3 Steps of program compositions part II

Having established the different types of composition, Curry considered the problem of how the major constituent parts of a program could be analysed into simpler programs and how these simpler programs can be synthesised into the main programs. Here, his experience with the 'logic of stages' (see p. 6) when programming the ENIAC was capital.

2.3.1 Basic programs

Since Curry's 'fundamental philosophy (cf. p. 3 above) aimed at an analysis of programs into their "most rudimentary components" relative to a given machine, these components were called *basic programs* [12, p. 3]:

> [The] analysis can, in principle at least, be carried clear down until the ultimate constituents are the simplest possible programs. These programs, which are here called basic programs, each consist of a simple order plus the necessary outputs and data. Of course, it is a platitude that the practical man would not be interested in composition techniques for programs of such simplicity, but it is a common experience in mathematics that one can deepen one's insight into the most profound and abstract theories by considering trivially simple examples

[9]We are indebted to Peter van Emde Boas for raising this question.

As is evident, this approach was inspired by Curry's interest in systems of symbolic logic that are of "an extremely rudimentary character" (cf. p. 3). Later, in a short 1952 note, Curry showed that this philosophy contrasted with the GvN-approach [13, p. 100]:

> Von Neumann and Goldstine have pointed out that [we] should not use the technique of program composition to make the simpler sorts of programs, – these would be programmed directly –, but only to avoid repetitions in forming programs of some complexity. Nevertheless, there are three reasons for pushing the technique clear back to formation of the simplest possible programs from the basic programs, viz.: (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 its necessary outputs and data. Two important concepts in this context are *locatum* and *term*. A locatum is a variable that designates a word in the machine. It can take different values in different stages of a computation and in similar computations based on different data. Unspecified locata are written as λ, μ, memory locata as x, y, z, u, v, w.[10] A term is a word constructed by the machine from its locata at any stage. The constants 0, 1 and locata are *basic terms*, a general, unspecified term will be written as ξ, η, ζ. To be able to represent terms as functions of other terms, Curry introduced *indeterminate terms* or variables, denoted by s, t, that can appear in functions, written as ϕ, ψ, ω.

If ξ is a term and λ a locatum, then:

$$\{\xi : \lambda\} \tag{1}$$

is a program that calculates the term ξ and stores it in the locatum λ. This is what we would call in modern computer science terms an 'assignment'. If a number of terms is connected with logical operators, equalities and inequalities, one obtains a predicate Φ and $\{\Phi\}$ will then designate a discrimination which tests whether Φ is true or not. The fundamental principle for the analysis into basic programs is:

$$\{\phi(\xi) : \lambda\} = \{\xi : \mu\} \rightarrow \{\phi(\mu) : \lambda\} \tag{2}$$

where ϕ is a function which depends on a set L of locata and ξ is a term constructed from locata in L. The equation (2) can be read as a definition of the left hand side by its righthand-side. This principle allows for the analysis

[10]Note that words in the accumulator (A) or in the register (R) are usually indicated by A resp. R.

of a program into basic programs by recursive definitions that rely on forms of (2). Note however that this definitional character of (2) is only acceptable if the locata in L are not changed by $\{\xi : \mu\}$. Because this criterion recurs often, we will indicate it as criterion ($\diamond$). Indeed, if, for instance, $\phi(x) = A + x$, $\xi = x, \lambda = \mu = A$ (with A the content of the accumulator) then the left hand side results in $A + x$ whereas the right hand side results in $2x$.

Now, what kind of programs are basic programs? An obvious choice would be those commands on the GvN list of basic IAS machine orders (Table 1). Indeed, Curry started from this list, but inserted important additions. There were several reasons for doing so. To begin, Curry pointed out that there are certain gaps in he GvN list, viz. some orders have no symmetric counterpart. But, the more important reason for introducing new basic programs was rooted in Curry's machine-awareness: some programs should be basic because they are used very frequently and, if they would be basic, would speed-up the programming process and save memory during the actual automated analysis into basic programs (see e.g. Sec. 2.3.2 and Appendix). Remember that, besides the 'human bottleneck' of the programming, memory was the other principal limiting factor, being 'more expensive' than time in those early machines: "The most important considerations are efficient use of the memory and efficiency in programming. The size of the memory determines the kind of problems the machine can handle." [12, p. 97]

Yet, for Curry, it was not only the actual machine architecture which is subject to a lot of technical contingencies that should determine the choice of basic programs. The logical analysis and the general problem of synthesising programs also had a word in the determination of what should be part of the machine's architecture or hard-wired into the machine. The relation between the machine and its programming should not only be a one-way, but a two-way relationship, where the affordances of programming should have an influence on the design of the machine.

Curry was acutely aware that this was an important issue. Already in his preface to the 1950-report, he strongly advised that "considerations affecting the design of the machine are likely to arise, so that it is advantageous that such studies [as Curry's] be prosecuted before the designs are completely frozen." [12, p. 5]. In his factual analysis, Curry took into account practical, machine-bound properties of speed and memory. Intriguingly, this was achieved by a theory that took distance from the actual machine, thereby not only introducing abstractions and possible generalisations but also optimisations.

Grounded in this philosophy, Curry came up with a new list of basic programs (Table 2). The most important difference between Curry's and GvN's list of basic orders are the receptive programs. Roughly speaking, receptive programs are programs in which the accumulator is a 'passive' receiver of some value. These values are the outcomes of the functions $\pi_i(t)$:

$$\begin{array}{ll} \pi_0(t) = +t & \pi_1(t) = -t \\ \pi_2(t) = +|t| & \pi_3(t) = -|t| \\ \pi_4(t) = A + t & \pi_5(t) = A - t \\ \pi_6(t) = A + |t| & \pi_7(t) = A - |t| \end{array}$$

Clearly, all $\pi_i(t)$ with $i > 3$ can be rewritten as $A + \pi_i(t)$ with $0 \leq i < 4$. Consider now receptive programs of the form

$$\{\pi_i(\xi) : A\} \tag{3}$$

with ξ a definite basic term. Thus, if $i < 4$ then (3) comes down to replacing the content of A by that of ξ, if $i > 3$ then (3) amounts to adding or subtracting some number from the content of A. If ξ is simply a locatum in the memory, then (3) with $0 \leq i < 8$ are the first 8 basic orders in the Table 1. Going beyond this table, Curry also considered the cases of (3) where:

$$\begin{array}{ll} \text{(a) } \xi = R & \text{(b) } \xi = A \\ \text{(c) } \xi = 0 & \text{(d) } \xi = 1 \end{array}$$

Curry discussed each of these cases (a)-(d) separately in order to see whether they suggested orders that should be included as basic programs.

(a) $\xi = R$. The program with $i = 0$ (put R in A) is included in Table 1, viz. number 10 of the list. Curry saw no reason not to include the other possible programs for this case with $i > 0$ (i.e. put $-R$, $|R|$ or $-|R|$ in A), they appear as numbers 8, 9 and 10 in his list of basic orders (Table 2).

(b) $\xi = A$. For $i > 3$ this comes down to clearing or doubling the content of the accumulator. For $\pi_0(A)$ nothing is done to A. Conclusion: only the programs for $\pi_{i=1,2,3}(A)$ have to be retained (4, 5 and 6 in Curry's list).

(c) $\xi = 0$. For $\{\pi_{i>3}(0) : A\}$ nothing changes to the content of A, for $\{\pi_{i\leq 3}(0) : A\}$ the accumulator is cleared. One order suffices to do that, $\{\pi_0(0) : A\}$, order 1 in Curry's list, absent in GvN's Table 1. The inclusion of this order makes it possible to clear the accumulator without the need of an extra memory position which stores 0.

(d) $\xi = 1$. The cases $\{\pi_{i=0,1}(1) : A\}$ are retained as orders 2 and 3 in Curry's list of basic programs. Although absent in the GvN-list, Curry pointed out that these orders would occur very frequently, e.g. as a counter for keeping track in iterations. Again, if it would be hardwired into the machine, it could save memory and time needed to access the memory.

This analysis motivated Curry to add no less than 14 basic programs to the GvN list (Table 1). Two more basic receptive programs were included that allowed for the logical "freak situation" [11, p. 17] as he phrased the very violation of 'type determinations', that is, when an order is manipulated as a datum. These programs are written as $d(*)$ and $e(*)$ respectively. The order $d(*)$ reads the location number of its own datum and thus makes available the location number of the order in the control. It is this kind of operation that allows to e.g. 'scroll' through a series of values in the memory. The table condition for regular programs that involve mixed arithmetic orders (see p. 14) limits the possibilities for this order, it can only be used for $i = 0, 4$: $\{\pi_{i=0,4}(*) : A\}$. The order $e(*)$ reads the location number of its own exit number into the accumulator. One last important basic program Curry added was the stop order.

Table 2 shows the complete list of Curry's basic orders. The second column gives the symbol for the program, the third the details of the program, i.e., the order, the datum and exit location. The last column makes the comparison with the GvN list (Table 1). The orders that are not included in the GvN list are indicated by a. Note that here, $\pi_i(c)$ stands for functions $\pi_i(t)$ with $i \leq 3$. I.e., the contents of A is first cleared before adding a new number to it. Similarly, $\pi_i(h)$ stands for functions $\pi_i(t), i > 3$ where a number is added to the content of A. The symbol U (unity) stands for $\xi = 1$, Z (zero) for $\xi = 0$.

Table 2: Table of Curry's basic programs

Nr. for i =				**Symbol**	**Program**			**effects**			**GvN for $i =$**			
0	**1**	**2**	**3**		o	d	e	A	R	X	**0**	**1**	**2**	**3**
1				$\{0 : A\}$	c	Z	2	0	-	-	a			
2	3			$\{\pi_i(1) : A\}$	$\pi_i(c)$	U	2	$\pi_i(1)$	-	-	a	a		
	4	5	6	$\{\pi_i(A) : A\}$	$\pi_i(c)$	A	2	$\pi_i(A)$	-	-		a	a	a
7	8	9	10	$\{pi_i(R) : A\}$	$\pi_i(c)$	R	2	$\pi_i(R)$	R	-	A	a	a	a
11	12	13	14	$\{\pi_i(x) : A\}$	$\pi_i(c)$	3	2	$\pi_i(x)$	-	x	x	x-	xM	x-M
15				$\{d(*)\} : A\}$	Gc	3	2	$d(*)$		x	a			
16	17			$\{A + \pi_i(1) : A\}$	$\pi_i(h)$	U	2	$A + \pi_i(1)$	-	-	a	a		
18	19	20	21	$\{A + \pi_i(R) : A\}$	$\pi_i(h)$	R	2	$A + \pi_i(R)$	R	-	a	a	a	a
22	23	24	25	$\{A + \pi_i(x) : A\}$	$\pi_i(h)$	3	2	$A + \pi_i(x)$	-	x	h	-h	Mh	-Mh
26				$\{A + d(*) : A\}$	Gh	3	2	$A + d(*)$	-	x	a			
27				$\{r\}$	r	–	2	$r(A)$		-	R			
28				$\{l\}$	l	–	2	$l(A)$		-	L			
29				$\{xR : A\}$	X	3	2			x	X			
30				$\{A : R\}$	R	A	2	A	A	-	a			
31				$\{x : R\}$	R	3	2	A	-	A	xR			
32				$\{A/x : R\}$	$\div$	3	2	A	A	x	$\div$			
33				$\{A : x\}$	S	3	2	A	-	A	S			
34				$\{A : d(*)\}$	Sd	3	2	A	-		Sp			

Continued on next page

Table 2 – continued from previous page																
Nr. for i =				**Symbol**	**Program**			**effects**			**GvN for** $i =$			
0	**1**	**2**	**3**		o	d	e	A	R	X	**0**	**1**	**2**	**3**
35				$\{A : e(*)\}$	Se	3	2	A	-		a			
36				$\{K\}$	Kc	2	–	-	-	-	C			
37				$\{A < 0\}$	Kh	3	2	-	-	-	Cc			
38				stop	0	–	–	-	-	-	a			

Curry's theoretical and logical thinking about programming not only resulted in an extended and more complete list of basic programs, it also resulted in two automatable methods for reducing the class of all receptive basic programs to only five and four basic receptive programs respectively, i.e., methods that allow to program certain basic receptive programs in terms of others. As Curry remarked [12, p.29]:

> the set of basic receptive programs is redundant in the sense that some of them have the same effect as composite programs made up from the others.

The two methods of reduction rely on the principle (2) (p. 20), a principle that also played a key role in Curry's method for automating the analysis of a program into its basic programs (See 2.3.2 and Appendix). And what's even more, his reductions had an impact on the use of memory. The first method does not use auxiliary memory, the second does.[11] We will only discuss the first method here, reducing all basic receptive programs 1–14 and 16–25 from Table 2 to only five basic receptive programs. We insist on introducing it here because it provides an intuition of how to turn the basic idea of Curry's whole thinking, to break up a given expression into smaller expressions up until its most 'rudimentary' components, into an algorithm. In a way this can be seen as Curry's transposition to the computer age of what logicians were doing in the 1920s, trying to express mathematical discourse with a minimum of logical operators.

In what follows, $\phi(t)$ is a function independent from A and ξ is either 1 or a locatum in the memory or the register R. The fact that two programs X and Y have the same output with the same effect is written as $X \cong Y$.[12]
Now using (2) we can rewrite (3) as:

$$\{\phi(\xi) : A\} \cong \{0 : A\} \to \{A + \phi(\xi) : A\}$$

[11] This suggests a trade-off between the minimum number of basic programs needed and the amount of auxiliary memory.

[12] Note that this is the standard notation from recursive function theory to indicate that two functions f and g either are both defined and have the same values or are either both undefined. This notation was introduced by Stephen C. Kleene [24] and was most probably known to Curry.

According to the criterion $(*)$, this can only be true if $\phi(t) = \pi_i(t)$, $i \leq 3$ (else, $\{0 : A\}$ changes the locata on which $\phi(\xi)$ depends, i.e., the accumulator. We also have:

$$\{A - \xi : A\} \cong \{-A : A\} \to \{A + \xi : A\} \to \{-A : A\}$$

This reduces cases $i = 5, 7$ to $i = 4, 6$.
Using these two formulae for reduction, 25 basic programs can now be synthesised from the following 5 'primitive' ones.

$$\begin{array}{ll}
\{0 : A\} & \\
\{-A : A\} & \\
\{A + \pi_i(t) : A\} & i = 0, 2 \\
\{A + \pi_i(R) : A\} & i = 0, 2 \\
\{A + 1 : A\} &
\end{array}$$

2.3.2 An arithmetic compiler

The task Curry set out to complete next, was the analysis and synthesis of arithmetic programs, viz. how to analyse an expression such as $(x+1)(y+1)(z+1)$ into its basic programs and synthesise it into a program of compositions of basic programs. The basic formula $\{\phi(\xi) : \lambda\} = \{\xi : \mu\} \to \{\phi(\mu) : \lambda\}$ once more provided the basis, but now as the starting point of an inductive scheme. This scheme resulted in a "more or less complete theory for the construction of an arbitrary [arithmetic] program" [12], where an arithmetic program, as defined by Curry, is a program that does not involve discriminations or mixed arithmetic orders (i.e., orders $d(*)$). Furthermore, as Curry emphasised, the details of the scheme resulted in a [13, p. 101]:

> systematic method of defining a program $\zeta : \lambda$ the writing out of the program, although somewhat involved , is none the less completely automatic

In modern words this amounts to a complete description of an arithmetic compiler, some years before people like Böhm, Rutishauser in 1952 [25, 218–227] or Laning and Zierler in 1954 [25, 236–240] came up with a description of a compiling routine. Because of its historical value, the detailed description of this paper compiler is included in the Appendix.

Two important features of the 'arithmetic compiler' should be highlighted. First, the focus on economising memory space in the 'compiling' process, second, the observation that mathematical equivalence does not imply program equivalence.

For his 'compiler' Curry distinguished between *elementary arithmetic programs* and *general arithmetic programs.* For each type he developed a separate 'compiling' method. The difference between the two lies in the fact that programs of the first type can be compiled without using auxiliary memory locations. This takes full advantage of memory savings made possible by relying on (the redundancy of) the set of basic programs (cf. Appendix for more details). In this context Curry recommended to either hard-wire all basic programs or to hard-wire a limited set of them together with the automatic method of reduction for basic programs (as described in sec. 2.3.1, p. 24) [12, p. 38–39]:

> [T]he possibility of making such [arithmetic] programs without using auxiliary memory is a great advantage to the programmer. Therefore, it is recommended that, if it is not practical to design the machine so as to allow these additional orders, then a position in the memory should be permanently set aside for making the reductions contemplated [before]

Curry also emphasised that his method does not *uniquely* determine a definition of a given arithmetic expression in terms of basic programs, since a given algebraic form for some arithmetic expression can often be worked out in different ways. Indeed, algebraic equivalence of arithmetic expressions like for instance:

$$(x+1)(y+1) = xy + x + y + 1$$

does not necessarily imply the expressions will be synthesised as identical programs. Indeed, the left-hand side compiles to:

$$\{\zeta : \lambda\} = \{x : A\} \rightarrow \{x+1 : A\} \rightarrow \{A : w\} \rightarrow \{y : A\} \rightarrow \{A+1 : A\} \rightarrow \{A : R\} \rightarrow \{Rw : A\} \rightarrow \{A : \lambda\}$$

the right-hand side to:

$$\{\zeta : \lambda\} = \{x : R\} \rightarrow \{yR : A\} \rightarrow \{A{+}x : A\} \rightarrow \{A{+}y : A\} \rightarrow \{A{+}1 : A\} \rightarrow \{A : \lambda\}$$

As a consequence, depending on the particular methods one is using, a shorter or a longer program might result, or a program that uses more or less auxiliary memory locations. This immediately links back to Curry's persistence in developing not only a practical theory of programming, taking into account the characteristics of the machine itself, but also a true calculus of programs (see Sec. 2.2.2, p. 19).

Lastly, a technical detail should be mentioned. While discussing the necessity to specify the actual auxiliary locata needed when combining arithmetic programs with others, Curry considered the possibility of assigning a term *simultaneously* to different locata, a kind of 'parallel assignment'. This is written as:

$$\{\zeta : \lambda_1, \lambda_2, \ldots, \lambda_n\}$$

Indeed, this is a very useful operation in the context of parallel programming and most probably stems directly from Curry's ENIAC experience.

2.3.3 Discrimination and secondary programs

The analysis and synthesis of discrimination and secondary programs (the latter ones involve mixed arithmetic orders) was not treated with full generality in Curry's report. Contrary to the arithmetic program, no "compiler" was described. Yet, a partial treatment was laid out that mainly focused on those compositions needed to complete the specific inverse interpolation program.

The basic discrimination program is represented by $\{A < 0\}$, if written in full length it is the program consisting of the order Kh (order 37 in Table. 2) and two outputs O_1 and O_2, where O_1 and O_2 are the two possible outcomes or branches of the conditional tested by the value in the accumulator. O_1 if $A < 0$, O_2 if $A \geq 0$. Normally, the output O_1 is consecutive to the Kh order and O_2 is another location in the program to which the discrimination can jump. To generalise this basic discrimination, Curry began to set out compositional definitions of four principal kinds of discriminations

$$\begin{array}{ll}
\{\xi < 0\} = & \{\xi : A\} \rightarrow \{A < 0\} \\
\{\xi > 0\} = & \{-\xi < 0\} \\
\{\xi < \eta\} = & \{\xi - \eta < 0\} \\
\{\xi > \eta 0\} = & \{\eta - \xi < 0\}
\end{array}$$

Then Curry proceeded to show how to analyse and synthesise simple propositional functions. Consider the logical combinations $\Phi_1 \wedge \Phi_2$ and $\Phi_1 \vee \Phi_2$ (logical AND and OR) and the logical negation $\{\sim \Phi\}$. The corresponding discrimination programs may be defined as [12, p. 49]:

$$\begin{array}{l}
\{\Phi_1 \wedge \Phi_2\} = \{\Phi_1\} \rightarrow (\{\Phi_2\} \rightarrow O_1 \ \& \ O_2) \ \& \ O_2 \\
\{\Phi_1 \vee \Phi_2\} = \{\Phi_1\} \rightarrow O_1 \ \& \ (\{\Phi_2\} \rightarrow O_1 \ \& \ O_2) \\
\{\sim \Phi\} = \{\Phi\} \rightarrow (\{Kc\} \rightarrow O_2) \ \& \ O_1
\end{array}$$

Here, $\{A\} \rightarrow \{B\} \ \& \ O_2$ means if A true then do B else go to output O_2. With these three equations it is now possible to synthesise a large class of logical propositions whose truth values could be used in a discrimination.

The remainder on discrimination programs mainly focussed on "practical" programs, needed for the synthesis of the inverse interpolation. This included iteration control and tests for "betweenness", or "bracket tests" that check whether a given number falls within a certain interval. An example of one side of a bracket test, $\{x < y\}$, is given in Table 3.

As far as the iteration controls are considered, it is said that "any collection of programs will have standard procedures for controlling iterations." [12, p. 59] Iteration control programs involve three programs: A working program Y to be iterated and with an output that will increase a quantity i, a preliminary program X and a terminal program Z to be initiated after the last iteration. Now, there are two possibilities, an iteration control can receive a signal from X, make a discrimination and then proceed to Y, or, the iteration control receives its signal from Y and then makes a discrimination. The increase of the quantity i is always after Y. These two cases are called initial and final iteration control,

		$\{x : A\}$	$\{A - y : A\}$	$\{A < 0\}$
1		+c 3	-h 3	Kh 3
2		0	0	0
3		x	x	0

	$\{x < y\}$	$\{x : A\}$	$\{A - y : A\}$	$\{A < 0\}$
1	+c 6	1		
2	–h 7	(2)	1	
3	Kh 5		(2)	1
4	0			2
5	0			3
6	x	3		
7	y		3	

Table 3: The example is $\{x < y\} = \{x - y < 0\}$ which is the result of the composition $\{x : A\} \rightarrow \{A - y : A\} \rightarrow \{A < 0\}$. In the first four lines, the instructions for the basic programs are given. The remaining lines list the program formed by the multiple composition. The 'compiling' tabular procedure to construct a program resulting from multiple composition is used. (Cfr. [12, p. 66])

or, to put it in more modern terms, a 'for-loop' and a 'while-loop'. An iteration with initial control can thus be synthesised as follows: $\{A : i\} \rightarrow \{A - m : A\} \rightarrow \{A < 0\} \rightarrow (\{i : A\} \rightarrow \{A + 1 : A\} \rightarrow \{Kc\})$ & O_z. The Kc order jumps back to the beginning ($\{A : i\}$) and m is the number of iterations.

Secondary programs do not fall under the theory developed in the 1949-report [11], but some instances of them are often needed. With an eye on the synthesis of the inverse interpolation program, Curry developed three types of secondary programs: remote control programs, tabulation/read-out programs and preparatory programs. Caution is demanded, since "the locata being substituted are likely to occur in such a variety of locations in the interiors of programs that [...] this would make our notation break down." [12, p. 78] However, the combination Curry presented "can be handled by the present technique, and [...] at least in the cases considered here, they will do what they are supposed to do." A remote control program might be best understood as a kind of exception handling. It was already useful back on the ENIAC, e.g. when an "irregular calculation" would normally "lead to an error signal" but "is not pathological".[12, p. 74] Instead of stopping the program in such a case, a signal can trigger a so called 'remote control' that will modify the normal calculation program. In general, one will need two nested simple remote control programs, that can be combined with other subprograms. A simple remote control is the following:

1	$Gc = \{d(*) : A\}$	5
2	$Sd = \{A : d(*)\}$	4
3	O_1	
4	O_2	
5	O_3	

Here, O_1 is the normal output, O_2 and O_3 are secondary outputs. Given two such controls Y_1 and Y_2, with Y_2 nested in Y_1, Y_1 will 'program' the normal situation (give the normal order of commands), but if it comes upon Y_2 in this regular situation, Y_2 will 'reprogram' the normal calculation to provide for the irregular situation (change order of commands).

Another classic situation where a secondary program is required, occurs when using tables. In order to go to a next value, or jump to another value in the table, it is necessary to 'calculate' with the address numbers. However, this sort of 'pointer-arithmetic' is only allowed within a certain range and finds its modern parallel in automatic memory protection schemes. To provide for this, Curry defined:

1	$Gh = \{A + d(*) : A\}$	5
2	$Sd = \{A : d(*)\}$	4
3	O_1	
4	O_2	
5	-	

The dash at 5 indicates that it is irrelevant what its contents are and, amusingly, is called a dummy named F_{ooo}. In the case when only quantities are involved, Curry introduced a new notation: with α location number, $L(\alpha)$ gives the word at location α. This exactly corresponds to the modern notion of 'pointers.'

Finally, in respect to the preparatory programs, Curry referred to Goldstine and von Neumann's routine as "a device for picking out a certain segment of the memory and increasing all the datum numbers of the orders in that segment by a fixed amount." [12, p. 90] Curry, in fact, translated the GvN-routine into his system of symbols, but instead of using this routine to synthesise the seven parts of his inverse interpolation, he chose another solution [12, p. 95]:

> [A]lthough [GvN's preparatory routine] has some automatic features to recommend it, it seems more complicated than is really necessary. A substitution program could be constructed along the lines of Sections B and C [tabulation routines] which would be simpler.

2.4 Role of Notation and System of Symbolisation

The transition in Curry's thinking from his encounter with the ENIAC towards a theory of program composition is reflected by successive stages that would 'transmute' this experience into "a system of symbolization" [11, p. 1]. Symbolisation acts as a mediator between the case study of inverse interpolation on the ENIAC in 1946 and the later, fully developed program composition technique of

1950. In his laboratory memorandum of 1949 Curry was pretty aware that ‘the fact that the illustrations are postponed to this later paper [of 1950] may make the present theory seem somewhat abstruse”, but nonetheless felt confident that ‘by means of a system of symbolization” his ‘attack’ of the given problem would be far more systematic and deliver, “in fact, a notation for program construction which is more compact than the “flow charts” of Goldstine and van Neumann.” [11, p. 7]

In lack of a physical machine that would actually implement Curry’s programming technique, notation is the very medium that can transform the ‘logic of stages’ drawn on paper (cf. Fig. 1) into a virtual procedure of symbol substitutions and thus beyond any visual control flow still drawn on paper by GvN. Indeed: “The present theory develops in fact a notation for program construction” [11, p.7]. Consequentially, already prior to a final mechanisation of his ‘abstruse’ technique, the bare practicability of such a notation raises the question for a suitable code.

This connects to Curry’s acute sensitivity to notation he cultivated within the field of logics. As gets obvious from a 1937 paper [9], Curry had worked on the perfection of Peano’s notation on the use of dots instead of brackets. In particular, he developed a notation that allowed for a deep nesting of terms, while keeping the structure of iterative formulae most transparent. Curry’s key concept to achieve a transparent notation was ‘significance’ such that in principle any nested formula is an entity in itself. When some ten years later Curry developed his notation for a theory of compositions, the ‘logic of stages’ suggested by the hardware architecture of ENIAC would match up with stages of symbolisation in logics. Since each component of a program requires a certain ‘significance’, that justifies to be considered a proper program term at all, the “conditions of significance” [9, p. 28] are reflected in the notation of composition.

The elementary notations used by Curry are $X \rightarrow Y$ for composition, $\{A : B\}$ for assignment, and $X \rightarrow Y \ \& \ Z$ for a conditional, meaning that X is either followed by Y or Z depending on the output of X. For a multiple substitution and complex composition, Curry finally came up with the following notation, $X \rightarrow (Y \rightarrow (Z \rightarrow O_1 \ \& \ < X > \ \& \ < Z >))$ for a program that is the composition of X with Y with Z that can lead to three outputs, to Output 1, back to X or back to Z ($< X >$ shows that X has already occurred).

For more complex compositions, while holding up “to keep the notation straight”, Curry gave a historical account of contemplatable forms to notate logical expressions – quite unusual to encounter in a technical military report [11, 45–49]. To avoid that a “large number of parentheses” may ruin the perspicuousness of the formula, Curry reviewed Peanese dot notation, Polish prefix operator notation (where “parentheses can be inserted in only one way”), a Frege-like notation (“perspicuous”, but “it must be remembered, Frege’s notation died with him”), and finally flow charts (“a quasi-graphical representation of the flow of the control”). In practice, Curry switched between notations, mainly using his original notation with &’s and brackets, sometimes, for very complex programs, the Peano-Curry notation, and for clearly indicating the discriminations, a variant of Frege's notation. For instance:

$$X \to (Y \to (Z \to O_1 \ \& \ < X > \ \& \ < Z >) \ \& \ < X >) \ \& \ < O_1 >$$

is written in Peano-Curry notation (with the $*$ showing previous occurrence) as follows:

$$X \to: Y \to .Z \to O_1 \ \& \ X * \ \& \ Z * . \ \& \ X * : \ \& \ O_1 *$$

In the Frege-variant this same program looks like:

$$\begin{tikzcd}
X \arrow[r] \arrow[dr, to path={|- (\tikztotarget)}] & Y \arrow[r] \arrow[dr, to path={|- (\tikztotarget)}] & Z \arrow[r] \arrow[dr, to path={|- (\tikztotarget)}] \arrow[ddr, to path={|- (\tikztotarget)}] & O_1 \\
 & O_1 * & X * & X * \\
 & & & Z *
\end{tikzcd}$$

3 Discussion

At first sight, and what might look as contradicting our 'integrative' approach stated in the introduction, it appears striking that Curry started his work on programming while having no actual machine at his disposal. However this only happened after a concrete experience with the ENIAC. This incidence suggests that lack of an actual machine helped him to make abstractions from the machine but more to point it seems that Curry's formalist procedures guided him to generalise the problems of programming by breaking them into his logic of stages reflecting the modular design of the machine and recompose these stages as a comprehensive theory informed by the architecture of yet another machine.

This observation might help to explain part of the differences between von Neumann's and Curry's approach to programming after their outset at the ENIAC. Indeed, contrary to Curry, von Neumann was confronted with computers on a daily basis. He was involved with the construction of the IAS machine and he still paid frequent visits to ENIAC. As is clear from Sec. 2.2 and 2.3, the differences between GvN and Curry are legion: GvN's [19] approach boils down to a set of heuristic tools (such as flowcharts) which were illustrated by a demo-set of example routines, whereas Curry takes off from a singular 'prototype'-problem to develop a complete theory of program compositions.

Originally, Curry feared that the third part of the GvN reports [19, Part III] *Combining routines* might overlap with his own work on composition of programs [11, p. 6]. Yet by 1950, Curry had seen this report and promptly concluded that his approach differed significantly from GvN's [12, pp. 3–4]. One might even say Curry was rather disappointed by the content of this final part of GvN, for he remarks that they only give "a preparatory program for carrying out on the main machine a rather complicated kind of program composition."

Curry continued his critique in a telling manner that underlined a still valid distinction between a proper programmer and a quick hacker [12, p. 4]:

> But one comment seems to be in order in regard to [GvN's] 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.

To this Curry added 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 comment 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 optimisation. Exactly these features are absent in GvN and seem to have been systematically underestimated by them.

Another affirmation is Curry's list of basic programs which extends those given by GvN. This 'extension' was actually a systematisation of GvN's ad-hoc list arranged by Curry's logical thinking and rendered concrete by his sensitivity to the machine's limited memory.

Another important point made by Curry concerns the fact that he well knew that his notation may appear "less formidable", but would at least be adequate for automatic programming, whereas the 'prodigious' flow chart notation of GvN is not. Therefore, "it will be found that another form of program composition will be necessary" [12, p. 3] and, indeed, "it is quite possible that the technique of program composition can completely replace the elaborate methods of Goldstine and von Neumann"[13, p.100]. Consequentially, at least in his case, "flow charts will be used [...] primarily as an expository device"[11, p.7], since:

> several persons have noticed that the technique in reference (g)[GvN 1947] involves a lot of more fuss then is really necessary [but] the procedures suggested by the theory of program composition [...] appear to lead to a less formidable technique, even when the problems are planned in detail from scratch. [11, p.50]

Although Curry's approach allows for a larger distance between the programmer and the machine, for it provides a less machine-dependent "system of symbolization", this does not mean that Curry's theory itself 'takes distance' from the machine. In effect and to bring out this point one might formulate the alleged oxymoron that because the applied logician makes abstractions from the machine he is more machine-aware than GvN. In order to emphasise the significance of logics for any field of human ventures, Curry noted:

> 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. [C]onsiderations affecting the design of the machine are likely to arise, so that is advantageous that such studies [as Curry's] be prosecuted *before the designs are completely frozen* [m.i.]. Efficiency in the management of an eventual computing enterprise can no doubt be furthered by such a study as this, but if it is to have that effect it must be pursued while the plans are still in the formative stage, and it must be carried beyond the stage of preliminary analysis to the point where it can be tried on practical problems. [12, p. 5]

Certainly, even if being all but certain "that the actual formation of the composite program can be carried out by a suitable machine" [11, p.7] this does not conjure one into existence. As far as practicability is concerned, it seems as if Curry's foray towards automatic programming shared the same fate as McCarthy's Lisp compiler, namely in waiting for an appropriate machine with enough memory. Curry was pretty aware that if ever his replacement and synthesis "process could be carried out with a suitable machine; considerable memory might be involved, but not extensive calculation" [11, p. 43–44]. This contingency, of course, is not without a certain irony, because contrary to Lisp, Curry's proposed substitution process of symbols was particularly concerned to minimise the waste of memory resources—when they are finally compiled to be run.

Seen historically, it appears that advanced and far reaching concepts in the field of programming antedated the progressiveness of machine development. Indeed, it is by making abstraction from the machine that it becomes possible to think about concrete improvements of the machine. This at least, is significant for Curry's work on program composition and his concept of an economising calculus. However, such prosperous options in the springtime of computer languages were shock-frozen by a mono-culture of the so-called von Neumann architecture as Backus' alert Turing Award Lecture of 1977 raised the issue [2, p. 614]:

> For twenty years programming languages have been steadily progressing toward their present condition of obesity; as a result, the study and invention of programming languages has lost much of its excitement.

27 years after Curry's work on programming, when Backus gave a historical reassessment of his own field and searched for a new functional "algebra of programs", he rediscovered one of Curry's dearest insights that if once "given a certain memory capacity, the principle bottleneck for efficient performance is the preparation of problems for the machine" [12, p. 97]. Still, and tellingly enough, the addressee of Backus' historical speech has not changed denouncing the "*von Neumann bottleneck*"[2, p. 614], by now a winged-word, for being the culprit

of the current misery. Backus, in fact, did refer to Curry's well-known work on combinators [15] but subsumed him under the same category of "applicative models" together with Church's lambda calculus [6] and pure Lisp [30]." [2, p. 615] Obviously unaware of Curry's eager for practical simplicity and the logician's explicit concernment with a still unsealed hardware-design, he concluded that "applicative computing systems [...] have not provided a foundation for computer design"[2, p. 616] and, as a consequence, assessed Curry's approach alongside with lambda calculus, or Lisp:

> Moreover, most applicative systems employ the substitution operation of the lambda calculus as their basic operation. This operation is one of virtually unlimited power, but its complete and efficient realization presents great difficulties [...] for example, pure Lisp is often buried in large extensions with many von Neumann features. The resulting complex systems offer little guidance to the machine designer. [2, p. 616]

Yet, after our detailed tracing of Curry's "attack" of the matter and proof of his 'fundamental philosophy' to be directly motivated by his ENIAC experience, one feels kind of obliged to modify this picture and to point out Curry's warning at the end of his last report on programming that "features of machine design which will cause an improvement in programming technique should be very seriously considered."[12, p. 97]

In this respect, it seems appropriate to mention at least a few of these 'Curry-features' and their relation to hardware modifications undertaken to widen the von Neumann bottleneck. Outstanding and first of all, there is Curry's in-depth classification of basic programs with regard to the demand of auxiliary memory (cf. sec. 2.3.1). Though for Haskell today laziness seems to prevail over eagerness for simplicity, the avoidance of side-effects, meaning programs that would "not disturb any locations in the memory", as Haskell Brooks Curry used to say, surely counts as one of the most important features of the functional programming paradigm and is a manifest design-goal,[13] when Haskell states [12, p.38–39]:

> Now the possibility of making such [arithmetic] programs without using auxiliary memory is a great advantage to the programmer. Therefore, it is recommended that, if it is not practical to design the machine so as to allow these additional orders, then a position in the memory should be permanently set aside for making the reductions contemplated.

A compliance with Curry's emphasised request for an extended set of orders and reserved memory locations right inside the ALU would certainly have helped to

[13] Astonishingly, the Haskell community does not seem to provide any explicit motivation for their own name-giving and, with regard to their logo, even stumble into the same felicity as Backus by equalising Church's lambda calculus and Haskell's theory of program composition: "The language is named for Haskell Brooks Curry, whose work in mathematical logic serves as a foundation for functional languages. Haskell is based on the lambda calculus, hence the lambda we use as a logo." Cf. http://www.haskell.org/haskellwiki/Introduction

circumvent the chronic stack-overflow issues associated with the paradigm of functional programming or might have even accelerated the inventions of backside caches (like L2/L3 memory) attached straight to the CPU. Furthermore, the rather early effort to provide for index registers can be seen in relation to Curry's table condition and as a means to avoid the "freak-condition" of self-manipulating orders featured so prominently in the von Neumann style of programming when indirectly paging through memory locata. Also the introduction of modern memory management units designed to prevent out-of-context access of data (or stored orders) in RAM can be regarded as devices to fulfil some of Curry's logical restrictions, now imprinted directly in hardware. Not in the least Curry's idea of assigning a term simultaneously to different locata (p. 26), might have allowed for a parallelisation of processes much earlier. Telling thereof is the interesting fact that today's on-chip parallelism schemes depend—much like Curry's theoretical synthesis—on a strict discrimination of instructions and data in order to drive their pipelines. In this respect, and most exciting perhaps, Curry's practical solution of the inverse interpolation problem back on the ENIAC can be regarded as 'pipelining' different processes concurrently through the accumulators of the machine (cf. p. 5).

Taking up our historiographical stand, accounts in the history of computing often tend to overemphasise only one aspect of computing (engineering, logics, programming, etc) culminating in the neglect of others, and thus forgetting about the actual computer as a meeting point rather than a divide. If one looks at ENIAC as a bifurcation point in the history of sciences, where things begin and develop fast and in different ways, one must recognise that an encounter with the ENIAC helped to beget the classic framework of a von Neumann machine and the associated von Neumann style of programming, but also inspired a different approach found in Curry's contributions. By following Curry's steps of program composition, starting from the concrete meeting between an advanced logician and the archaic "behemoth" ENIAC, it is clear that a confrontation between the physical machine and abstract symbolic logic can result in a theory of programming that functions as an interface between both sides. This interface is not a visual display of program schemes like the GvN flowchart but is mounted on a system of symbolisation that acquires "significance" as the details of the composition are sketched and put together. Since this meets "a somewhat similar situation which has arisen in combinatoric logic" [11, p. 27] , as Curry referred to his backgrounds, one may say that the German notion of "Umwandlung"—imported to allow for logical reductions, or for symbol sequences to "be manipulated according to the rules of the theory of combinators [11, p. 27] (cf. sec. 2.4) —has been 'curried' to its essential meaning, namely for 'transmuting' into an algorithm of symbol substitutions that we nowadays call a compiler.

As a result, 'impact' of reciprocal permeation of architectural hardware design and logical software schemes sets in before the automation of programming becomes viable. Hence, re-reading Backus' text [2], filtered back and through Curry's work, it nearly seems that Curry might have liberated us from the von Neumann style, well before it was established. Constructive suggestions were explicitly made: "say by removing the tube A-43 on the drawing PX 8-304, [since] the stepping of the stepper can then be controlled by the counters". [16, p. 57] How much of 'Curry', or how much of an engagement of symbolic logic with electronic circuits would have been necessary to significantly change the historical course of things, and thus to what extent 'currying' could have been transmuted into 'engineering', remains an open question.

This matter cannot, however, be looked into further. [12, p. 96]

Appendix: An arithmetic compiler

In what follows we give the details of Curry's arithmetic compiler.

An *arithmetic term* ζ is defined by induction as follows. Let L be a class of locata and let $t_1, ..., t_n$ be indeterminate terms. Then ζ is an arithmetic term if, either, ζ is an *initial term* viz.:

$$\begin{array}{ll} \zeta = 0 & \\ \zeta = 1 & \\ \zeta = t_i & i = 1, 2, \ldots, n \\ \zeta = \lambda & \text{where } \lambda \text{ is in } L \end{array}$$

or, with ξ and η arithmetic terms:

$$\begin{array}{ll} \zeta = \pi_i(\xi) & i = 1, 2, 3 \\ \zeta = \xi + \eta & \\ \zeta = \xi\eta & \\ \zeta = \xi/\eta & \end{array}$$

An arithmetic function $\phi(t_1, \ldots, t_n)$ is defined by:

$$\phi(t_1, \ldots, t_n) = \zeta$$

with ζ an arithmetic term.

Similarly, Curry defined the class of *elementary terms* by induction on a class L of locata and indeterminates $t_1, ..., t_n$. A term ζ is elementary if, ζ is an initial term, or, given an elementary term ξ and a location x in the memory from the locata L:

$$\begin{array}{ll} \zeta = \pi_i(\xi) & i = 1, 2, 3 \\ \zeta = \xi + \pi_i(x) & i = 0, 1, 2, 3 \\ \zeta = \xi + \pi_i(1) & i = 0, 1 \\ \zeta = \xi x & \\ \zeta = \xi/x & \end{array}$$

An elementary function based on L is a function $\phi(t_1, \ldots, t_n)$ defined by:

$$\phi(t_1, \ldots, t_n) = \zeta$$

with ζ an elementary term. The order of a term is defined as the number of times the inductive process needs to be applied. E.g., if ξ is of order n then $\pi_i(\xi)$ will be of order $n + 1$.

What exactly is the difference between arithmetic and elementary terms? The definition of elementary terms involves the basic operations of addition, multiplication and division between an elementary term and a memory location (or, in case of addition, unity). For arithmetic terms these operations are between two arithmetic terms. As will be explained, this has an impact on the need for auxiliary memory locations when applying the principle (2), $\{\phi(\xi) : \lambda\} = \{\xi : \mu\} \rightarrow \{\phi(\mu) : \lambda\}$, to define any arithmetic term (elementary or not) as a composition of basic programs. Remember that the right-handside ($\{\xi : \mu\} \rightarrow \{\phi(\mu) : \lambda\}$) can only function as a proper definition of the left-handside ($\{\phi(\xi) : \lambda\}$, if $\{\xi : \mu\}$ does not change the locata on which $\{\phi(\xi)$ depends (cf. criterion ($\diamond$)). For elementary terms, this is guaranteed by the inductive definition Curry provided. For the arithmetic terms, auxiliary memory is sometimes needed. Intuitively speaking, this is the case when one is confronted with the situation when it is required to use a given location (for example, the accumulator A) whereas its content is needed at some later time. To understand this problematics better, think of the classical problem of switching values in variables. If $x = a$ and $y = b$, to arrive at $x = b$ and $y = a$ you need an extra memory location.

An arithmetic program is the 'assignment' of an arithmetic terms ζ to some locatum λ, viz.

$$\{\zeta : \lambda\}$$

The method for defining any arithmetic program as a composition of basic programs relies on the associativity properties of $\rightarrow$ discussed in Sec. 2.2.2 as well as on the following four special cases of principle (2):

$$\begin{array}{ll}
(\alpha_1) & \{\zeta : \lambda\} = \{\zeta : A\} \rightarrow \{A : \lambda\} \\
(\alpha_2) & \{\zeta : \lambda\} = \{\zeta : R\} \rightarrow \{R : \lambda\} \\
(\beta_1) & \{\phi(\xi) : A\} = \{\xi : A\} \rightarrow \{\phi(A) : A\} \\
(\beta_2) & \{\phi(\xi) : R\} = \{\xi : R\} \rightarrow \{\phi(R) : R\}
\end{array}$$

The condition ($\diamond$) on memory is "vacuous" in the case of α_1 and α_2, since no functions are involved. In the case of β_1 and β_2, one has to be careful that $\{\xi : A\}$ and $\{\xi : R\}$ do not disturb any memory locata, viz. ϕ should not involve A or R as a parameter.[14] Using the instances $\alpha_1 - \beta_2$ of (2) it is now possible to proceed by induction to define any elementary function with ζ an elementary term *without disturbing any locations in the memory*, except, possibly, λ itself. However, this is only possible if Curry's additional basic programs from Sec. 2.3.1 are hard-wired into the machine. Else, even the simple program $\{A : R\}$ would disturb the memory locata and would thus require auxiliary memory locations.

[14]This is the case if:

$$\phi(t) = \begin{cases} \pi_i(t) & i + 1, 2, 3 \\ t + \pi_i(x) & i = 0, 1, 2, 3 \\ t + \pi_i(1) & i = 0, 1 \\ tx & \\ t/x & \end{cases}$$

Hence, when ζ is an elementary term.

Elementary Arithmetic Programs The inductive definition of any arithmetic program $\{\zeta : \lambda\}$ differentiates between several cases of ζ (being an elementary term):

a. If ζ is a basic term:

   1. The program P is basic if either (1) $\lambda = A$, (2) $\zeta = A$, (3) $\zeta = x, \lambda = R$.
   2. Else, reduce by α_1 to case where $\zeta = A, \lambda = A$.

b. $\zeta = \pi_i(\xi)$, $i = 1, 2, 3$:

   1. The program is basic if $\lambda = A, \xi = A, R, x$, or 1.
   2. If $\lambda = A$ and P is not basic, reduce to $\xi = A$ by (β_1).
   3. If $\lambda \neq A$ reduce to λ by (α_1)

c. $\zeta = \xi + \pi_i(x)$ or $\zeta = \xi + \pi_i(1)$, $(i = 0, 1, 2, 3)$

   1. The program is basic if $\lambda = \xi = A$.
   2. If $\lambda = A$, $\xi \neq A$, reduce to $\xi = A$ by (β_1)
   3. If $\lambda \neq A$, reduce to $\lambda = A$ by (α_1)

d. $\zeta = x\xi$

   1. The program is basic if $\lambda = A$, $\xi = R$
   2. If $\lambda = A, \xi \neq R$, reduce to $\xi = R$ by (β_2)
   3. If $\lambda \neq A$, reduce to $\lambda = A$ by (α_1)

e. $\zeta = \xi/x$

   1. The program is basic if $\lambda = R, \xi = A$
   2. If $\lambda = R, \xi \neq A$ reduce to $\xi = A$ by (β_1)
   3. If $\lambda \neq R$ reduce to $\lambda = R$ by (α_2)

Clearly, the principles α_1–β_2 are the main principles for breaking up an arithmetic program $\{\zeta : \lambda\}$ into smaller programs, by reducing the elementary term ζ of order n into elementary terms of a lower order, ultimately resulting of the analysis of $\{\zeta : \lambda\}$ into basic programs. Note that for each of the cases (a)-(d), the first step is always to check whether or not $\lambda = A$. If this is not the case, then α_1 is used to set $\lambda = A$. Similarly for case (e), division, if $\lambda \neq R$ then α_2 is used to set $\lambda = R$.[15]

Curry remarked that for any arithmetic program $\{\zeta : \lambda\}$, with ζ elementary, then, if "all missing parentheses in the algebraic notation for ζ are supplied, so that it is clear in what order the operations are to be performed, then it is

[15]Note that the fact that one needs α_2 instead of α_1 for case (e) is explained by the fact that the basic program for division from Table 2 is $\{A/x : R\}$ and thus an assignment to the register.

uniquely determined which of the cases a-e applies." [12, p. 40] If one adds to this method a specification of the order in which operations $x + y$ and xy need to be performed ("say, the first one occurring in such cases"), then his method is a [13, p. 101]:

> systematic method of defining a program $\zeta : \lambda$ the writing out of the program, although somewhat involved , is none the less completely automatic

In order to illustrate his method, Curry applied it to the following example of an elementary term:

$$\zeta = y_0 + x_1(y_1 + x_2(y_2 + x_3y_3))$$

Define:

$$\begin{aligned}\eta_2 &= y_2 + x_3y_3\\ \eta_1 &= y_1 + x_2\eta_2\end{aligned}$$

then:

$$\zeta = y_0 + x_1\eta_1$$

Clearly, ζ falls under case (c) of the inductive definition, hence by (c.3) we get:

$$\{\zeta : \lambda\} = \{\zeta : A\} \to \{A : \lambda\}$$

By c.2, we get:

$$\{\zeta : A\} = \{x_1\eta_1 : A\} \to \{A + y_0 : A\}$$

By d.2 we get:

$$\{x_1\eta_1 : A\} = \{\eta_1 : R\} \to \{x_1R : A\}$$

Combining the last three equations we get the following definition for $\{\zeta : \lambda\}$:

$$\{\zeta : \lambda\} = \{\eta_1 : R\} \to \{x_1R : A\} \to \{A + y_0 : A\} \to \{A : \lambda\}$$

Similarly with R for λ:

$$\begin{aligned}\{\eta_1 : R\} &= \{\eta_2 : R\} \to \{x_2R : A\} \to \{A + y_1 : A\} \to \{A : R\}\\ \{\eta_2 : R\} &= \{y_3 : R\} \to \{x_3R : A\} \to \{A + y_2 : A\} \to \{A : R\}\end{aligned}$$

Thus:

$$\{y_0 + x_1(y_1 + x_2(y_2 + x_3y_3)) : \lambda\}$$

is defined as the following composition of basic programs:

$$\begin{aligned}&\{y_3 : R\} \to \{x_3R : A\} \to \{A + y_2 : A\} \to \{A : R\} \to \{x_2R : A\}\\ &\to \{A + y_1 : A\} \to \{A : R\} \to \{x_1R : A\} \to \{A + y_0 : A\} \to \{A : \lambda\}\end{aligned}$$

By using the theory of composition explained in Sec. 2.2.2, it is possible to transform this composition into an actual program executable by the machine, including among other things, the actual datum location numbers for each of the basic programs and the like. Because this is possible (and 'automatic' in its literal sense) "it would be pedantic to go through the details" [12, p. 51].

General Arithmetic Programs As explained in the previous section, the fundamental difference between elementary arithmetic programs and general arithmetic programs is that elementary programs, after analysis into basic programs, do not need auxiliary memory positions for their synthesis. General arithmetic programs that are assignments of an non-elementary arithmetic term ζ to a locatum λ do need auxiliary memory. Thus, the method developed for elementary programs has to be modified for non-elementary arithmetic programs. Hence, Curry introduced the notion of a *degree m of an arithmetic program.* The degree m of an arithmetic program is the number m of auxiliary memory positions needed for the program to work properly. The term ζ assigned to λ is then called a term of degree m.

In order to define general arithmetic programs as compositions of basic programs, Curry again proceeded by induction. If ζ in $\{\zeta : \lambda\}$ is elementary, then ζ has degree $m = 0$. For terms ζ not elementary, Curry provided techniques to reduce a program of degree $m \neq 0$ to a composition of programs of degree 0, i.e. elementary programs, which in their turn can be reduced to a composition of basic programs. Given a non-elementary arithmetic program, the first step, if necessary, is to set $\lambda = A$ (using α_1) for those arithmetic terms ζ of the form $\pi_i(\xi)$, $\xi + \eta$ or $\xi\eta$ (note that this is the same for elementary terms).[16] It is assumed that the arithmetic term $\zeta = \phi(\xi)$ and that, initially, $\phi(t)$ is a function independent from A and R. We already have that $\{\zeta : \lambda\}$ is reduced to $\{\zeta : A\}$ by α_1.

There are two cases to be considered: either $\xi \neq A$ or $\xi = A$. If $\xi \neq A$ then we can use β_1, thus:

$$\{\phi(\xi) : A\} = \{\xi : A\} \rightarrow \{\phi(A) : A\} \tag{4}$$

Assume now that the two components on the right have degree m and n respectively. Curry pointed out that then the m auxiliary memory locations of the first component can be chosen such that $\phi(t)$ is independent of them and the n memory locations of the second component can be chosen such that as many of them as possible are among the first m. Then it easily follows that the analysis into components by β_1 results in two components neither of which has a higher degree than the degree of the program analysed.

The second case to be considered, which occurs in the second component of (4), is the case where $\xi = A$ in $\phi(\xi)$. If the program is not elementary then:

$$\phi(\xi) = \psi(\xi, \eta)$$

In this case, we need to clear the accumulator to perform the calculation and thus need to use an auxiliary memory location w to temporarily store the contents of A. To this end, Curry introduced the following equation:

$$\{\phi(A) : A\} = \{A : w\} \rightarrow \{\phi(w) : A\} \tag{5}$$

[16]Curry did not consider arithmetic non-elementary terms of the form ξ/η, but the method for defining arithmetic terms that involve division can be easily constructed on the basis of the method for the other cases. Thus, if $\zeta = \xi/\eta$ then the first step is to set $\lambda = R$. In what follows we will only consider the arithmetic terms ζ that do not involve a division.

It easily follows from this equation that if $\{\phi(w) : A\}$ is a program of degree m then $\{\phi(A) : A\}$.
The reductions (4) and (5) give a process that allows to express an arithmetic program $\{\zeta : \lambda\}$ in terms of simpler programs, similar to $\{\zeta : \lambda\}$ but of a lower degree. By lowering the 'complexity' of the structure of the program, one ultimately arrives at a composition of basic programs. Curry was aware that the method does not uniquely determine a definition in terms of basic programs for a given arithmetic expression, since a given algebraic form for some arithmetic expression can often be worked out in different ways. As a consequence, the number and order auxiliary locata may be different.

To illustrate this point, let us consider the following example of an non-elementary arithmetic program:[17]

$$\zeta = (x+1)(y+1) \tag{6}$$

Let:

$$\xi_1 = x+1 \quad \eta_1 = y+1$$

Then:

$$\zeta = \xi_1\eta_1$$

Then, as the first step, if $\lambda \neq A$ we use α_1 to get $\lambda = A$:

$$\{\zeta : \lambda\} = \{\zeta : A\} \to \{A : \lambda\}$$

Then, applying (4) we get:

$$\{\zeta : A\} = \{\xi_1 : A\} \to \{A\eta_1 : A\}$$

Clearly, $\{\xi_1 : A\}$ is an elementary program so we easily have:

$$\{\xi_1 : A\} = \{x : A\} \to \{x+1 : A\}$$

The program $\{A\eta_1 : A\}$ however is not elementary so we need (5):

$$\{A\eta_1 : A\} = \{A : w\} \to \{w\eta_1 : A\}$$

Now, $\{w\eta_1 : A\}$ is elementary. By applying Curry's definition for elementary programs we ultimately get:

$$\{w\eta_1 : A\} = \{y : A\} \to \{A+1 : A\} \to \{A : R\} \to \{Rw : A\}$$

Combining all these compositions we get:

$$\{\zeta : \lambda\} = \{x : A\} \to \{x+1 : A\} \to \{A : w\} \to \{y : A\} \to \{A+1 : A\} \to \{A : R\}$$
$$\to \{Rw : A\} \to \{A : \lambda\}$$

However, this is not the shortest program of basic programs for (6). Indeed, we could also have used the fact that:

$$(x+1)(y+1) = xy + x + y + 1$$

[17] Curry considered the slightly more complicated example $(x+1)(y+1)(z+1)$.

Thus, the following program is also a composition of basic programs for (6):

$$\{\zeta : \lambda\} = \{x : R\} \to \{yR : A\} \to \{A{+}x : A\} \to \{A{+}y : A\} \to \{A{+}1 : A\} \to \{A : \lambda\}$$

Thus, in the first case, ζ is a non-elementary term, in the second, ζ is in fact elementary; and depending on the way $(x+1)(y+1)$ is worked out, two different programs result, the one shorter than the other. This observation leads one to considerations concerning the uniqueness of program for a given term. As Curry pointed out, a term should be [12, p.46]:

> thought of as the result of a process of construction from the locata. Different processes of construction should be thought of as different terms, even though the results are algebraically equal.

In other words, the terms ζ in the two examples are not the same kind of terms and thus we have two different programs. From this point of view, it follows from the definition of an arithmetic term that ζ in the second example can actually not be regarded as an arithmetic term at all, since its method of construction does not follow the method for constructing an arithmetic term. Although this differentiation does not comply with the intuition of a human mathematician in the process of calculation, it makes sense for a machine that automatically 'compiles' its routines of computation. Again Curry preferred a method that, although it would not be the one used by a human mathematician, it is very suitable to be used by a machine (cf. p. 4).

References

[1] William Aspray. *John von Neumann and the origins of modern computing.* Cambridge, MA/London, MIT Press, 1990.

[2] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21 (8): 613–641, 1978.

[3] Maarten Bullynck and Liesbeth De Mol. Setting-up early computer programs: D. H. Lehmer's ENIAC computation. *Archive for Mathematical Logic*, 49 (2): 123–146, 2010.

[4] Arthur W. Burks, Herman Heine Goldstine, John Von Neumann Preliminary Discussion of the Logical Design of an Electronic Computer Instrument Vol. I, Part I of *Report on the mathematical and logical aspects of an electronic computing instrument*, 1946.

[5] Paul E. Ceruzzi. A history of modern computing. Cambridge, MA, MIT Press, 2002 (2nd edition).

[6] Alonzo Church The calculi of lambda-conversion. Princeton, Princeton University Press, 1941.

[7] Dick Clippinger A logical coding system applied to the ENIAC. BRL 673, Ballistic Research Laboratories, Aberdeen Proving Ground, 1948.

[8] Haskell B. Curry Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, 52 (3 & 4): 509–536 & 789–834, 1930.

[9] Haskell B. Curry On the Use of Dots as Brackets in Logical Expressions *Journal of Symbolic Logic*, 2 (1): 26–28, 1937.

[10] Haskell B. Curry. The combinatory foundations of mathematical logic. *The Journal of Symbolic Logic*, 7(2): 49–64, 1942.

[11] Haskell B. Curry. On the composition of programs for automatic computing. Technical Report 9805, Naval Ordnance Laboratory, 1949.

[12] Haskell B. Curry. A program composition technique as applied to inverse interpolation. Technical Report 10337, Naval Ordnance Laboratory, 1950.

[13] 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: Gauthier-Villars, 1954.

[14] Haskell B. Curry. *Foundations of mathematical logic.* New York: McGraw Hill, 1963.

[15] Haskell B. Curry. and Robert Feys. Combinatory Logic, Vol. I Amsterdam, North-Holland Pub. co., 1958.

[16] 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.

[17] W.B. Fritz. The Women of ENIAC. *IEEE Annals of the History of Computing*, 18 (3): 13–28, 1996.

[18] Herman H. Goldstine and John von Neumann. On the Principles of Large Scale Computing Machines. *John von Neumann, Collected Works*, 6 Vols., Oxford (1961–1963), Vol. V, p. 1–32, Lecture manuscript, 1946.

[19] Herman H. Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument Volume 2 of *Report on the mathematical and logical aspects of an electronic computing instrument*, part I,II and III. Report prepared for U. S. Army Ord. Dept. under Contract W-36-034-ORD-7481, 1947-48.

[20] Herman H. Goldstine. *The Computer from. Pascal to von Neumann.* Princeton: Princeton University Press, 1972.

[21] David Grier The ENIAC, the Verb "to program" and the Emergence of Digital Computers. *IEEE Annals of the History of Computing*, 18 (1): 51–55, 1996.

[22] Hartree, D.R. The ENIAC, an Electronic Computing Machine *Nature*, 158, 500–506, 1946.

[23] Iuri I. Ianov. On the equivalence and transformation of program schemes. *Communications of the ACM*, 1(10): 8–12, 1958.

[24] Stephen C. Kleene. Recursive predicates and quantifiers *Transactions of the American Mathematical Society*, 53(1): 41–73, 1943.

[25] 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: Academia Press, 1980.

[26] Derrick H. Lehmer A history of the sieve process In J. Howlett, N. Metropolis and G.-C. Rota, editors, *A History of Computing in the Twentieth Century*, pages 445–456, New York: Academia Press, 1980.

[27] Michael S. Mahoney. The histories of computing(s). *Interdisciplinary science reviews*, 30 (2): 119–135, 2005.

[28] Michael S. Mahoney. What makes the history of software hard and why it matters. *IEEE Annals for the history of computing*, 30 (3): 8–18, 2008.

[29] A.A. Markov. *Theory of algorithms.* Translated from the Russian edition (1954). The National Science Foundation, Washington, D.C. and The Department of Commerce by The Israel Program for Scientific Translations, Jerusalem, 1962.

[30] John McCarthy Recursive functions of symbolic expressions and their computation by machine Part I. *Communications of the ACM*, 3 (4), 184-195, 1960.

[31] 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.

[32] Liesbeth De Mol and Maarten Bullynck and Martin Carlé. Haskell before Haskell: Curry's Contribution to Programming (1946-1950). In F. Ferreira, B. Löwe, E. Mayordomo, L. Mendes Gomes, editors, *Programs, Proofs, Processes, CiE 2010*, Volume 6158 of *LNCS*, pages 108–117. Springer, 2010.

[33] Hans Neukom. The Second Life of ENIAC. *IEEE Annals of the History of Computing*, 28 (2): 4–16, 2006.

[34] George W. Patterson. Review of “The logic of program composition by H.B. Curry”. *The Journal of Symbolic Logic*, 22(1): 102–103, 1957.

[35] A. Rose. Lightning Strikes Mathematics: ENIAC *Popular Science*, 148 (April): 83–86; 1946.

[36] Henry S. Tropp. Franz Alt, interview, September 12, 1972. *Computer Oral History Collection, 1969–1973*

[37] Henry S. Tropp. Jean Bartik and Frances Holberton, interview, April 27, 1973. *Computer Oral History Collection, 1969–1973*

[38] Konrad Zuse. Some Remarks on the History of Computing in Germany. In J.Howlett, N. Metropolis, and G.-C. Rota, editors, *A History of Computing in the Twentieth Century*, pages 611–627, New York: Academia Press, 1980.