AISB/IACAP World Congress 2012

Birmingham, UK, 2-6 July 2012

Symposium on the History and Philosophy of Programming

Liesbeth De Mol and Giuseppe Primiero (Editors)

Part of

Published by
The Society for the Study of
Artificial Intelligence and
Simulation of Behaviour

http://www.aisb.org.uk

ISBN 978-1-908187-17-8

Foreword from the Congress Chairs

For the Turing year 2012, AISB (The Society for the Study of Artificial Intelligence and Simulation of Behaviour) and IACAP (The International Association for Computing and Philosophy) merged their annual symposia/conferences to form the AISB/IACAP World Congress. The congress took place 2–6 July 2012 at the University of Birmingham, UK.

The Congress was inspired by a desire to honour Alan Turing, and by the broad and deep significance of Turing's work to AI, the philosophical ramifications of computing, and philosophy and computing more generally. The Congress was one of the events forming the Alan Turing Year.

The Congress consisted mainly of a number of collocated Symposia on specific research areas, together with six invited Plenary Talks. All papers other than the Plenaries were given within Symposia. This format is perfect for encouraging new dialogue and collaboration both within and between research areas.

This volume forms the proceedings of one of the component symposia. We are most grateful to the organizers of the Symposium for their hard work in creating it, attracting papers, doing the necessary reviewing, defining an exciting programme for the symposium, and compiling this volume. We also thank them for their flexibility and patience concerning the complex matter of fitting all the symposia and other events into the Congress week.

John Barnden (Computer Science, University of Birmingham)
Programme Co-Chair and AISB Vice-Chair
Anthony Beavers (University of Evansville, Indiana, USA)
Programme Co-Chair and IACAP President
Manfred Kerber (Computer Science, University of Birmingham)
Local Arrangements Chair

Foreword from the Symposium Chairs

Liesbeth De Mol[1] and **Giuseppe Primiero**[2]

Given the significance of computing for modern society, the relevance of its history and philosophy can hardly be overestimated. Both the history and philosophy of computing only started to develop as real disciplines in the '80s and '90s of the previous century, with the foundation of journals (e.g. the IEEE Annals on the History of Computing, Minds and Machines and the like) and associations (SIGCIS, IACAP, . . .), and the organization of conferences and workshops on a regular basis. A historical awareness of the evolution of computing not only helps clarifying the complex structure of the computing sciences, but it also provides an insight in what computing was, is and maybe could be in the future. Philosophy, on the other hand, helps to tackle some of the fundamental problems of computing: the semantic, ontological and functional nature of hardware and software; the relation of programs to proofs and, in another direction, of programs to users; the significance of notions as those of implementation and simulation, and many more. The aim of this conference is to zoom into one fundamental aspect of computing, that is the foundational and the historical problems and developments related to the science of programming.

Alan Turing himself was driven by the fundamental question of 'what are the possible processes which can be carried out in computing a number'. His answer is well-known, and today we understand a program as a rather complex instance of what became known as the Turing Machine. What is less well-known, is that Turing also wrote one of the first programming manuals ever for the Ferranti Mark I, where one feels the symbolic machine hiding on the back of the Manchester hardware. This was only the beginning of a large research area that today involves logicians, programmers and engineers in the design, understanding and realization of programming languages.

That a logico-mathematical-physical object called 'program' is so controversial, even though its very nature is mostly hidden away, is rooted in the range of problems, processes and objects that can be solved, simulated, approximated and generated by way of its execution. Given its widespread impact on our lives, it becomes a responsibility of the philosopher and of the historian to study the science of programming. The historical and philosophical reflection on the science of programming is the main topic at the core of this workshop. Our programme includes contributions in

1. the history of computational systems, hardware and software
2. the foundational issues and paradigms of programming (semantics and syntax, distributed, secure, cloud, functional, object-oriented, etc.).

Our wish is to bring forth to the scientific community a deep understanding and critical view of the problems related to the scientific 'paradigm' represented by the science of programming. The kind of questions analyzed and relevant to our task are:

- What was and is the significance of hardware developments for the development of software (and vice versa)?
- In how far can the analogue and special-purpose machines built before the 40s be understood as programs and what does this mean for our conception of 'program' today?
- How important has been the hands-off vs. the hands-on approach for the development of programming?
- What is the influence of models of computability like Church's lambda-calculus on the development of programming languages?
- Which case studies from the history of programming can tell us today something about future directions?
- Is programming a science or a technology?
- In how far does it make sense to speak about programming paradigms in the sense of Kuhn?
- What are the novel and most interesting approaches to the design of programs?
- How do we understand programs as syntactical-semantical objects?
- What is the nature of the relation between algorithms and programs? What is a program?
- Which problems are the most pressing ones and why are they relevant to more than just programmers?
- How can epistemology profit from the understanding of programs' behavior and structure?
- What legal and socio-economical issues are involved in the creation, patenting or free-distribution of programs?

The invited speakers for this symposium are Gerard Alberts and Julian Rohrhuber. *Gerard Alberts* (University of Amsterdam) is a well-known historian of computing. He is the series editor of the Springer book series on the history of computing and one of the editorial members of the leading journal on the history of computing, viz. IEEE Annals for the history of computing. He is also the project leader of the SOFT-EU project. In his talk, he is going to tackle the rather philosophical question: What does software mean?, by studying how it developed historically. *Julian Rohrhuber* (Robert Schumann School of Music and Media in Düsseldorf) is a co-developer of the open source computer language SuperCollider, a language for real time audio synthesis and algorithmic composition and an experienced live coder. Apart from this more 'practical' work, he has made various contributions to philosophy of science in general and the philosophy of programming in particular. He will present some of his philosophical reflections on programming. As an additional Special Event, Julian can also be seen in action during a live performance on Wednesday July 4th titled "*When was the last time you spent a*

[1] Postdoctoral fellow of the Fund for Scientific Research – Flanders, CLMPS, Ghent University, email: elizabeth.demol@ugent.be
[2] Postdoctoral fellow of the Fund for Scientific Research – Flanders, CLMPS, Ghent University, email: giuseppe.primiero@ugent.be

pleasant evening in a comfortable chair, reading a good program" (Jon Bentley). *Live coding to celebrate the Turing Centennial*, by the Birmingham Ensemble for Electroacoustic Research and Julian Rohrhuber playing improvised algorithmic network music.

The other contributors to this Symposium and their talks are:

- Wolfgang Brand, *Two Approaches to One Task: A Historical Case Study of the Implementation and Deployment of two Software Packages for the Design of Light-Weight Structures in Architecture and Civil Engineering*
- Selmer Bringsjord and Jinrong Li, *On the cognitive science of computer programming in service of two historic challenges*
- Timothy Colburn and Gary Shute, *The Role of Types for Programmers*
- Edgar G. Daylight, *A Compilation of Dutch Computing Styles, 1950s–1960s*
- Vladimir V. Kitov, Valery V. Shilov, Sergey A. Silantiev, *Anatoly Kitov and ALGEM algorithmic language*
- Shintaro Miyazaki, *Algorhytmic listening 1949-1962. Auditory practices of early mainframe computing*
- Pierre Mounier-Kuhn, *Logic and computing in France: A late convergence*
- Allan Olley, *Is plugged programming an Oxymoron?*
- Uri Pincas, *On the Nature of the Relation between Algorithms and Programs*
- Nikolay v. Shilov, *Parallel Programming as a Programming Paradigm*

Our programme will be followed by a double session on *Philosophy of Computer Science meets AI and Law*, organized by Rainhard Bengez (TU München) and Raymond Turner (University of Essex).

The Symposium on History and Philosophy of Programming is intended as a follow-up to the First International Conference on History and Philosophy of Computing (`www.computing-conference.ugent.be`), a IACAP sponsored event. The Conference, which took place in November 2011 at Ghent University, represented a first approach to build a community of philosophers and historians working in the area of the computational sciences. The present smaller Symposium will be a bridge to the Second edition of the History and Philosophy of Computing Conference, to be held in October 2013 in Paris. We hope everyone interested in the historical and systematic study of computational sciences and their intersections with other sciences and its applications will get involved in what promises to be a crucial and exciting research area.

Table of Contents

Developing an historical notion of software

Gerard Alberts[1]

Simple as it seems to agree that software is more than programming, trying develop a specific notion of software is rather complicated. Today we take an historical approach. What does software mean when we try to pinpoint it historically? In this talk I will sketch three steps towards an historical notion of software, followed by a note on the clarifying effect of this historical exercise.

One may look for early occurrences of the word software and find these in the era 1955-1960. Admittedly, there is a natural opposition to the expression hardware, but this will not lead us much further than a negative demarcation of software being anything to do with computers which is not hardware. Historians of computing have noted that software was given a separate name in the late 1950s when people were hired to do programming, training or maintenance and other services around the computer (Martin Campbell-Kelly 2003). Tom Haigh (2002) adds to this the striking phenomenon that a similar job of programming might be called coding when done in-house and software when performed by an external consultant or contractor. Not only was software soft, it was also ware. By consequence, to pinpoint the notion of software, we request the assistence of economic historians. Upon closer inspection, and this is third step, software had a specific content. The experts hired for the service of getting the machine to work, would typically be less involved with one dedicated set of operations like the bookkeeping systems or calculations in a specific field, but rather with programs to run the machine smoothly. While in 1955 the debates among computer experts were on automatic coding systems, in the following years such systems evolved into compilers, languages, and operatings systems (Michael Mahoney 2002; Nathan Ensmenger 2010; Gerard Alberts 2010). Such were the kind of things called software: programs that help other programs run smoothly, translate them, take care of memory management, of input-output. Software was an indication for the specific kind of programs generating programs. Historical precision thus yields that software, was soft and ware and comprised not just any programming but programs generating programs.

This historical notion considerably adds to technical notions of software. Even if after a decade, around 1970, for all practical purposes software and programming were taken to be synonyms, it helps to remind ourselves of the divergent origins. Moreover, it raises the awareness that the fundamental idea of programs being manipulated while running took a decade to take foot as a practice. We are tempt to think that Von Neumann, Goldstine and Burks in 1946 and Wilkes, Wheeler and Gill in 1951 were aware of this fundamental insight in programming computing machinery. If they were, it took another decade for this insight to turn into a practice of programming programs generating programs: software. Two practical consequences may be drawn from this historical insight.

1. as the early notion of automatic coding system reveals, software meant that not only calculating, sorting and dataprocessing were left to the computer, but that also programming was transferred to the machine.
2. one of these systems of autocoding came to be called language, programming language, introduced around 1955. It pays to realize how deeply intrenched this language metaphor is in the discipline of computer science.

[1] University of Amsterdam, The Netherlands, email: G.Alberts@uva.nl

Algorithmic Complementarity. Some thoughts on experimental programming and the history of live coding

Julian Rohrhuber[1]

Abstract. Today one can say that programming has not only osmotically infused scientific and artistic research alike, but also that those new contexts elucidate what it may mean to be an algorithm. This talk will focus on the 'impatient practices' of experimental programming, which can never wait till the end, and for which it is essential that the modification of the program in some way integrates with its unfolding in time. A contemporary example is *live coding*, which performs programming (usually of sound and visuals) as a form of improvisation.

Early in the history of computer languages, there was already a need felt for reprogramming processes at runtime. Nevertheless, this idea was of limited influence, maybe because, with increasing computational power, the fascination with interactive *programs* eclipsed the desire for interactive *programming*. This may not be an accidental omission, its reasons may also lie in a rather fundamental difficulty, on which we will focus here.

In itself, the situation is almost obvious: not every part of the program-as-description has an equivalent in the program-as-process. Despite each computational process having a dynamic nature, an integration of programming into the program itself must in principle remain incomplete. As a result, a programmer is forced to oscillate between mutually exclusive perspectives. Arguably, this oscillation reveals a specific internal contradiction within algorithms, a necessary obstacle to semantic transparency. By calling this obstacle *algorithmic complementarity*, I intend to open it up for a discussion in a broader conceptual context, linking it with corresponding ideas from philosophy and physics.

Here a few words about this terminology. Complementarity has been an influential idea in conceptualising the relation between the object of investigation, as opposed to the epistemic apparatus and the history of practice. Originating in the psychology of William James, where it referred to a subjective split of mutually exclusive observations, Niels Bohr used it to denote the existence of incommensurable observables of a quantum system (position vs. momentum, time vs. energy). Independent of the particular answer Bohr gave, complementarity raises the question of whether such a coexistence is induced by the experimental system or already present in the subject matter observed. Accordingly, in the case of programs, we may ask whether this obstacle is essential to their nature or whether it is a mere artefact of a specific formalisation. Algorithms, arguably situated between technical method and mathematical object, make an interesting candidate for a reconsideration of this discourse.

The existence of an obstacle to semantic transparency within algorithms and their respective programs need not mean a relative impoverishment of computation. Conversely, prediction is the wager and vital tension in every experimental system, as well as in interactive programming. After the conceptual discussion, I will try to exemplify this claim by introducing a few examples in the recent history of *live coding*. Again and again surfacing in form of symptoms such as an impossibility of immediacy, I hope this practice will be conceivable in terms of having algorithmic complementarity as one of its driving forces.

[1] Institute for Music and Media, Düsseldorf, email: julian.rohrhuber@musikundmedien.net

Two Approaches to one Task: A Historical Case Study of the Implementation and Deployment of two Software Packages for the Design of Light-Weight Structures in Architecture and Civil Engineering

Wolfgang Brand[1]

Abstract. The advent of powerful computers during the 1960s enabled architects and civil engineers for the first time to design and construct light-weight structures never dreamed of before. This historical case study describes the implementation and deployment of two software packages for the design of light-weight structures at the University of Stuttgart, Germany in the context of the software engineering and hardware technology around 1970. Both software packages were used to design the tent-shaped membrane roof of the Munich Olympic Stadium for the 1972 Olympic Games. One software package (ASKA) was based on the Finite Element Method (FEM), the other package relied on the newly developed Force Density Method (FDM). The ASKA package had been under development since the early 1960s, whereas the development of the second package had just started. This environment led to two different design processes and to a rather limited interaction between the two groups carrying out the work. Both applications proved to be successful in creating light-weight structures and the fate of both software packages is traced until today. The design of the two software packages was influenced by the philosophy of structuring problems in a way appropriate to specific high performance computer architecture's. Today, this paradigm is still at the core of software engineering for supercomputers.

1 INTRODUCTION

Since ancient times, design and planning in architecture and civil engineering has been grounded in building models and experimenting. After the formalisation of mechanics in the 18^{th} and 19^{th} Century, and the advent of numerical algorithms, the early computing machines, with their limited capabilities, allowed for the first time the employment of numerical computations and simulations in architecture and civil engineering during the first half of the 20^{th} Century.

The emergence of high speed electronic computers (supercomputers) during the 1960s enabled architects and civil engineers to study their designs using numerical methods on an even larger scale, tackling problems of dimensions that have never been reachable before. These new computational capabilities allowed for the design and construction of novel structures. Structures that employed new materials and tried to mimic *natural shapes*. However, such shapes are of a highly non-linear nature and can be handled only with numerical methods.

While designing the tent-shaped membrane roof of the stadium for the 1972 Olympic Games in Munich, large scale numerical computations and simulations played a decisive role. Frei Otto (1925–), architect and head of the Institute for Lightweight Structures at the University of Stuttgart, had been working and experimenting for a long time with lightweight tensile and membrane structures, space frames and their structural properties. The architect Günter Behnisch (1922–2010) and the civil engineer Jörg Schlaich (1934–), both professors at the University of Stuttgart, managed to transform his design ideas with their working groups into a piece of landmark membrane architecture of steel and glass (see [19] and [24]).

Because there was only a limited time frame to do the calculations, the architects decided to award the contracts to conduct the actual calculations for the roof to two working groups that used different approaches. To be able to handle those computations, not only powerful and fast computers had to be available and new algorithms had to be developed, but they also had to be implemented in a consistent and reliable way. As always, there are many ways leading to Rome. This proverb holds in the design and implementation of large software packages, too.

One group was headed by John H. Argyris (1913–2004), who coinvented the Finite Element Method (FEM), and was able to use his approach on the design of the membrane roof. Another group was led by Klaus Linkwitz (1927–), whose group was able to deploy methods from geodesy to determine the optimal shape and inner structure of the constructions. Both groups had to implement their solutions by either extending existing software packages, as done by Argyris, who used his ASKA (Automatic System of Kinematic Analysis) package, or by implementing a new package, as done by the Linkwitz group (see [1] and [36]).

1.1 The setting

This case study will focus on the time around 1970 during which these two groups adapted and implemented their software packages in a hands-on manner. The dynamics of the interaction between the two groups is studied and the approaches taken are compared and evaluated. Typical obstacles which had to be overcome are described.

[1] Universität Stuttgart, Historisches Institut, Abteilung für Geschichte der Naturwissenschaften und Technik, Keplerstraße 17, 70174 Stuttgart, Germany, email: wolfgang.brand@studenten.ims.uni-stuttgart.de

One of them was the quality assurance of the results. One of the groups did this by calling in the German Army to do manual quality checks.

It is also shown that basic elements of software engineering, a well structured and systematic approach to software development and maintenance, were invented without much interaction with the emerging discipline of computer science and the further fate of the two software packages till today will be traced. They are still at the core of software packages used today.

These developments unfolded on powerful hardware, such as Control Data's CDC 6600, which provided sufficient computing power to users. It is described how the paradigm of high performance computing evolved and how this paradigm remains largely unchanged until today, being nearly independent of many recent developments in software engineering and programming language design.

There have been only a limited number of contributions regarding the history of computers in architecture and civil engineering which are focusing mainly on Computer Aided Design (CAD). To our knowledge it is for the first time, that numerical software packages are the object of study and the materials presented have never before been used in a historical case study. Parts of this study are based on oral interviews with those who participated in these developments and university documents never used before.

2 A SHORT NOTE ON THE HISTORY OF SOFTWARE ENGINEERING

Today, Software Engineering is a well established term in computer science and the software industry, which dates back to the late 1960s. During this decade the so-called *software crises* forced programmers and academics to reconsider the procedures of software production. During the early days of computer programming, the code was short, hand-crafted and could be maintained with reasonable effort. The first computers which became available after the Second World War and during the 1950s had rather limited resources. Bytes and words, today's standard format for internal data and instructions, were yet unknown and only introduced by IBM in their System 360 several years later. Every two or three years, machines were replaced and the programmers would face new instruction sets and data sizes. Program code had to be updated and adapted to the new computing environment on a regular basis. The first high level programming languages provided enhanced portability of code and empowered the programmers to write code of a substantial size. Along with the length of the code, the complexity increased. And so did the costs of writing the programs, debugging, and maintaining them over their life-cycle. During the 1960s, the burden of software development in this manner became unbearable, both with respect to production time and economic costs. Software quality was deteriorating and the overall impression among those involved was the urgent need for change. In 1968, at a NATO-sponsored conference, chaired by Friedrich L. Bauer, the challenges, requirements, and obstacles which had to be overcome were discussed in detail. This conference not only coined the term *software engineering*, but also triggered the creation of new development tools and methods which were centred on the design of programming languages implementing new paradigms, such as modularity or object orientation. Other domains of activity were code documentation, verifying and testing (see [8], [9], [14], [43], [47], [49] and [46, p. 32]).

The decade of the 1960s also saw dramatic changes in the way computers were programmed. In the beginning, there was an intimate relationship between the programmer and his machine. Only he would know all the intricacies of the device and the tricks of the trade to get the most out of the intractable hardware. Later on, the computer became a device locked-up in a separate room, taken care of by specially trained operators. Programmers punched their code into paper cards and delivered boxes with hundreds or even thousands of them to the operators. They would feed the punch cards into card readers and from there the data went into the memory of the computer. This batch processing mode, as it was called, led to notable turn-around times. It often took several hours, sometimes even days, until the result of the program execution was returned in the form of a long sheet of printer paper. The early 1960s witnessed also another invention: The time-sharing system. Several programs shared the resources of the hardware, which gave the impression that programmers and users, who sat in front of the first interactive terminals, had a machine for themselves. However, this new way of using computers came at an expense. The complexity and size of the operating systems increased dramatically (see [46, p. 32–34] and for a personal recollection [19]).

One of the few attempts to structure the history of software engineering is based on patterns and characterises the era around 1970 as *programming in-the-small*. Simple specifications of input and output data where sufficient for the degree of complexity commonly deployed in those days. The emphasis of programmers and software developers during the design process was on algorithms and their optimal representation of the problems to be solved. Usually, programs would run once to generate the required results and then terminate. New parameter sets or tasks demanded another execution of the software. Early examples of more elaborate data structures and types can be found, but especially in numerical calculations just matrices of numbers were used and the persistent database for storage was a deck of punch cards (see [19], [41] and [43]).

In the following decade, the specifications of systems grew in complexity and the focus of software designers shifted from single algorithms to system structure and management. Elaborate and disciplined long-time data management using databases superseded ad-hoc solutions. Program systems became so rich in new features that they would execute continually and not terminate after one solution was obtained. Increasing levels of software complexity and interdependency were fuelled by the advances in hardware technology and computer systems architecture. Users with sophisticated numerical applications were always open to enhanced systems and saw them as indispensable tools to achieve their goals. Others regarded these developments with suspicion and feared a deterioration of software quality – covered up by the progress in speed and storage capacity (see [19], [41], [43] and for a more sceptical view [46]).

This environment formed the context in which two software packages for the design of light-weight structures in architecture and civil engineering were implemented and deployed for complex tasks such as the design of the tent-shaped membrane roof of the Olympic Stadium for the 1972 Olympic Games in Munich.

3 THE ADVENT OF THE SUPERCOMPUTER

Over centuries little progress had been made regarding the mechanisation of calculation. Early attempts by Blaise Pascal and others to build a calculation apparatus had no lasting impact. The projects of Charles Babbage to build a predecessor of the modern computer in the first half of the 19^{th} Century failed due to the lack of appropriate technologies. The discovery of the nature of electricity and the technological advances in the first half of the 20^{th} Century, which led to advanced electronic components, such as vacuum tubes, transis-

tors and ultimately integrated circuits (ICs), enabled not only calculating machines, but programmable devices which would implement the ideas of Turing, Church, v. Neumann, and others developed in the 1930s and 40s. Today, only a few people remember, that in the 1920s and 30s, mostly female, operators of electro-mechanical calculators, were named *Computers* (see [8], [14], [21], [47] and [49]).

The end of the Second World War and the rising tensions of the Cold War saw a proliferation of companies keen to build computers. Mostly from the United States (for example IBM), some from the United Kingdom (Ferranti) or Germany (Zuse), and usually with a background in military research, these companies tried to utilise many different architectures and technologies to create programmable computing devices. Not only digital computers, as envisioned by John v. Neumann, Alan Turing, and others, but also analog computers to solve systems of differential equations quite efficiently (see [4], [33], [47] and [49]).

IBM, the largest and most successful of the new computer manufacturers, ruled the world market of commercial mainframe computers in the 1950s. Applications such as inventory management, bookkeeping, and databases required complex instruction sets and sophisticated storage facilities. Starting in 1957, Minneapolis-based Control Data Corporation (CDC), conquered the market with a radically different philosophy. Their head designer Seymour Roger Cray, the *Father of Supercomputers*, put all his design efforts into systems for numerical and scientific applications. His intention was to build the fastest computer in the world. By reducing the overhead in the instruction set and streamlining the architecture, he managed to create by 1964 the most powerful computer in the world: The Control Data CDC 6600. This machine offered their users a fifty times increase in speed compared to other computers available at that time. It became an instant commercial success with over 100 installations all over the world. Its two core memories had a capacity of 128 thousand and 500 thousand words. Each word consisted of 60 bits and the machine could execute nearly one million floating-point operations per second (MFLOPS). The core of the system was able to communicate over independent communication channels with plotters, printers, high speed card readers and punchers, magnetic tape stations and one of the first CRT terminals in the market. Remote users were able to connect their infrastructure via telephone and dedicated data lines to the system. In the 1960s, the CDC 6600 was the market leader of high performance computer systems and the ideal tool for large-scale numerical calculations like the ones required in architecture and civil engineering (see [2], [29] and [32]).

4 TWO SOFTWARE PACKAGES FOR THE STRUCTURAL ANALYSIS OF LIGHT-WEIGHT STRUCTURES

In 1966, the Olympic Committee selected Munich to host the XX. Olympics Summer Games in 1972. Although Germany would host the Olympics Summer Games for the second time, after the 1936 Berlin Games, the organising committee had only six years to build a completely new infrastructure from scratch in Munich. The still existing facilities in Berlin could not be used, because Berlin, as a result of the Second World War, was divided by the Iron Curtain and Munich had to take its place.

Germany wanted to take the opportunity to present itself as a country with an open and liberal society. A country which had left the dark ages of the Third Reich behind and formed an integral part of the world community. Frei Otto, one of the most prominent German architects of the 20^{th} Century, had developed the innovative architectural style that inspired the leading architect Günther Behnisch, who had won the architectural competition proposing a transparent, open and innovative tent-shaped membrane roof for the Olympic Stadium. After winning the competition in October 1967, Behnisch's design was regarded as being very appealing, but "... no one from the jury actually believed that the proposed design of the light-weight structure, covering an area of 75.000 m^2, could be realised." (cited as in [39, p. 99]). It was up to Frei Otto, who had been working on naturally shaped structures in architecture for over a decade and was willing to offer a helping hand, to convince Behnisch not to abandon his design concept (see for these paragraphs [34], [35] and [42]).

The German architectural press was well aware of the fact, that it would not be an easy endeavour to actually erect such a construction and remarked:

> The realisation and conviction are important, that not the tent roof structure à la Montréal was awarded, as a fashion of architecture, but it was the overall concept of the work of Behnisch that convinced the jury. (...) The built form is not the primary, but the aim conception for a task, which exactly is not derived from a formal aspect but from the nature of the problem. And for this the technological construction possibilities ought to be found. (cited as in [39, p. 99])

It soon turned out, that Frei Otto's standard approach of handling non-linear design problems by drawing plans and building physical models, were data on the shape of the structures could be collected by measurements, would not be sufficient. The dimensions of the construction and the forces acting on the pre-stressed cable nets forming the membrane roof proved to be too large for a simple model building solution. The models could only be used to provide the raw input data for computational models. The actual calculations to determine the shape of the roof had to be done by computers. However, neither the methods and algorithms for computing and optimising the roof's shape were available nor any software to do the actual calculations had been developed until that day.

After Frei Otto had modelled the cable nets forming the roof, using tulle and metal wires, the data from the fragile models were collected by high resolution photogrammetric measurements. The results obtained from the models (scaled 1:50) were of high quality with errors well below the one millimetre level. However, even such minor errors could lead to deviations in the actual construction in the range of several centimetres. The cable nets, which carried the acrylic-glass tiles that formed the roof, were held under stress to get their shape and had to be constructed with the outmost precision (see for these paragraphs [18], [19], [28] and [39]).

The design and computation of the roof had to be completed by 1970. This was the only thing known for sure when the first construction works started in the late 1960s. The pressure under which the two groups, contracted by the Olympia Baugesellschaft and headed by John H. Argyris and Klaus Linkwitz, had to fulfill their tasks was tremendous. They were not only in charge of designing separate segments of the Olympic Stadium's roof, but were also required to produce lists of materials, cable segments and the construction plans to enable a construction process without interruptions. The overall construction costs of the Olympic buildings would soon explode and went from an estimated 17 million German Marks to 190 million German Marks. However, such an important project was supported by all relevant social and political groups in Germany and the majority of the population was willing to carry this additional burden. (see [19], [39] and [42]).

The two teams, who were commissioned to do the actual calcula-

tions for the design, were both experienced in solving complex problems and were equipped with one of the most advanced computers of their time. The Control Data CDC 6600 was at the centre of John H. Argyris' Regionales Rechenzentrum (Regional Computing Centre) at the University of Stuttgart. Argyris had recognised the potential of high speed computers quite early in this career. In 1957, while still with the Imerial College in London, he had already asked for a powerful *Ferranti Pegasus* computer system. Although he usually knew how to get his way, he had to wait until 1959, when he became director of the Institute of Statics and Dynamics of Aerospace Structures (ISD) at the then Technische Hochschule Stuttgart, to get access to sufficient computing resources to pursue this ambitious plans of a computer-based science (see [15, p. 331–332], [17], [23], [13] and [36]).

Both groups had to use the same hardware to do the calculations. However, both the methods and the software they applied were completely different. Argyris, who had co-invented the Finite Element Method during the late 1940s and early 1950s, adapted his method employing his ASKA software package, whereas Linkwitz and his team started with a surface and curve fitting model and created the Force Density Method as part of their project (see [19]).[2]

4.1 The ASKA software package

John H. Argyris was the co-inventor of the Finite Element Method (FEM) and being a civil engineer by training, he knew about the importance of numerical methods right from the beginning. One of the main application areas of the Finite Element Method is structural engineering (see [25]), others are electrical engineering, aerodynamics, nuclear reactors, and many more. Structural analysis studies the deformation of arbitrarily shaped objects under stress. Generally, no analytical solutions to these problems can be found and only numerical approximations are possible. The Finite Element Method partitions complex objects into an assembly of much simpler components. The discrete elements usually take the form of a triangle or a tetrahedron. The boundary lines of these elements meet in nodes and form a mesh or grid covering the whole object. By solving a system of equations describing the nodes and elements, the behaviour of the whole system can be studied numerically using computer programs. There are several ways to describe and model the behaviour of such an assembly. Argyris' contribution to the development of the Finite Element Method was the creation of the matrix displacement method which relies on systems of linear equations. Linear equations can be solved efficiently on a digital computer using matrix operations. In this case, there is a perfect match between the method and the machine to execute it (see [22]).

The development of the ASKA (Automatic System of Kinematic Analysis) software package, implementing the matrix displacement method using sparse matrices, started in the early 1960s at the institute of John H. Argyris at the University of Stuttgart. By the end of the decade several versions of the program had been developed. The latest edition was implemented using FORTRAN IV, a high-level programming language, making the code machine-independent. The system was designed right from the start to grow from humble beginnings step by step in an open manner to a software package of substantial size. To be able to cope with the complexity and the expanding application fields, the system had a modular structure and a library of over 30 different element types, which could be assembled to form a fitting finite element model. New element types with other properties could be added at any time without the need to restructure the whole system. This enabled investigations beyond linear static problems, including plasticity, large displacement effects, and instability phenomena. By 1969, ASKA had reached the level of a framework which allowed the integration of modules to study dynamical processes. The Dynamical Analyser (DYNAN) included different algorithms for free vibration analysis and transient response. Arbitrary time-dependent loads could be applied to damped and undamped structures.

The intention of the programmers behind ASKA was to provide an elegant and descriptive modelling language for the users to relieve them of any tedious and unnecessary technical details. The processing steps were structured into simple and transparent statements and allowed for a problem-oriented approach. As a first step, the physical model (idealisation) of the structure had to be selected. Complex structures like buildings or the fuselage of aircrafts would be divided into substructures. At the second stage all necessary input data, such as shape, element properties (e.g. size, kind of material, etc.), coordinates of the nodes in the mesh, and any external loads had to be provided to the software. Optionally, ASKA could check the input data for consistency. It was designed to validate intermediate results and to work with missing or inconsistent data. This provided a certain kind of user-friendliness, because the time to identify errors in the model was reduced considerably.

The actual computation was done by a sequence of ASKA statements (each representing a specific algorithm, most of them based on linear algebra) which were applied to each substructure and finally to the main structure. The process could be terminated at any time and formed a kind of pipeline structure. To facilitate the interaction with the users, a graphical representation of the results using a CRT display could be generated (see for these paragraphs [6]).

The design philosophy behind ASKA was quite advanced for the 1960s and characterised by the strict separation of functionalities into different modules. Right from the beginning, input and output data were well specified to minimise errors and to improve the usability of the software. The emphasis was on sophisticated numerical algorithms. The rich repository of typed basic elements to be used for model building and the employment of sparse matrices to reduce storage space were another advanced feature of the software package. The focus on modules that execute step-wise and then can be terminated is another typical feature of software produced around 1970. However, one has to keep in mind, that these design decisions had already been made in the early 1960s. Therefore, the ASKA software package had been about five years or one generation ahead of the general developments in software engineering and computer science (see [6] and [43].

4.2 The evolution of ASKA

The ASKA software package was used in numerous large and complex projects throughout the world, but remained deeply rooted in the academic environment (see [40] and [48]). There were only a limited number of contacts with commercial software vendors or manufacturers. Obviously, Argyris was afraid of losing control over the further development of his brainchild. In the early 1980s, the situation started to change and by 1984 a group of former employees of his institute formed a company called *INTES* based in Stuttgart.

[2] A more detailed account of the design of the membrane roof of the Munich Olympic Stadium using supercomputers is presented in [12]. This study reports in more detail about the intellectual background of the two competing working groups headed by John H. Argyris and Klaus Linkwiktz and how their training and experience influenced the way they solved the problem of shape-finding of cable nets.

INTES was intended to be the vehicle to commercialise and sell the software in the market (see [16] and [31]). This proved to be a very time-consuming and length process with a lot of frictions and obstacles. The institute of Argyris had co-operated with a company called *IKOSS* in Stuttgart which claimed some rights in the ASKA software, as did some employees of his institute. It took some time to find a sustainable way to form *INTES* (see [10]).

Today, ASKA, which sells under the brand name PERMAS, has found its place in the market of professional Finite Element Method software. After more than 25 years, the current version (PERMAS V13) consists of more than 3 Million lines of code and is employed in such diverse fields as, statics, dynamics, acoustics, heat transfer, optimisation, electromagnetic fields, and many more (see [16]).

4.3 The software package for the Force Density Method

The group of Klaus Linkwitz had to start more or less from scratch. Two members of the group would play a decisive role in the development of methods and software for the design of light-weight structures. The mathematician Hans-Jörg Schek would develop the Force Density Method (FDM) and Lothar Gründig would adapt methods known from geodesy to structure analysis and write the first programs to do the calculations. Compared to the Argyris group, they had much less resources, much less experience using and writing large software packages, and no software at hand which could be easily modified or extended to solve their problems.

When Lothar Gründig started to code the software, his first task was to find some tiny sample programs and a textbook to get an impression of how the CDC 6600 and its FORTRAN compiler really worked. There was only limited support from the operators and technical details, such as advanced methods of data handling, had to be discovered by studying the manuals or talking to other users.

Both algorithms and software had to be developed in parallel under an enormous time pressure. The core objective was to get results, even when that happened in a quick and dirty manner. There was neither time or manpower to implement sophisticated data structures or to formally specify input, intermediate or output data formats. The database holding the measurement data collected from Frei Otto's physical models of the roof by photogrammetry where stored on punch cards and only temporary data were written onto magnetic tape. Using sparse matrices to do the actual computations was dictated by the limited memory capacity of the CDC 6600 and the decision to plot the results was a requirement of the customers.

The approach of the Linkwitz group was focused on the implantation of an algorithm, not on the creation of a sophisticated software package which could be marketed. The programs were run once, produced their results and then terminated. There was no really interactive element in the software and the emerging standards of software engineering around 1970 were only fulfilled on a rudimentary level. It took many years until the software became so advanced that it could be commercialised using the brand name Easy (see for these paragraphs the recollections of Lothar Gründig [19], [20], [45] and for an introduction to the Force Density Method [44]).

4.4 The evolution of Easy

After the calculations for the Olympic Stadium's roof had been completed, both Hans-Jörg Schek and Lothar Gründig resumed working on their academic careers and later became professors in Zürich and Berlin, respectively. However, Lothar Gründig never lost interest in the software he had created. Compared to the Finite Element Method pursued by Argyris, the algorithms of Schek and Gründig were limited to the design of light-weight structures where architects had to consider physical properties of their planned structures during the design process. However, this focus on light-weight structures only, created a software which gave the architects much more flexibility and freedom to implement their creative ideas (see [19] and [45]).

Finally, in 1989 *technet GmbH* was founded. This company would enhance the existing software and market it on a world-wide basis. Today, Easy consists of five modules which cover all aspects of the design of light-weight structures. Nearly four decades after the design of the Munich Olympic Stadium, Easy was employed in the design of the Allianz Arena in 2006, the new football stadium in Munich (see [45]).

5 TWO GROUPS WITH THE SAME TASK

The two groups working on the design of the Munich Olympic Stadium's roof were assigned separate parts of the structure. Working in parallel would speed up the design process and provide a certain degree of redundancy in case one of the approaches would fail.

Although both teams relied on the CDC 6600 as they basic infrastructure there was not much interaction. The results of the previous sections document how different their background, their approach and their daily work had been. The Argyris group was in the quite convenient position to have a reliable software package (ASKA) at hand. The necessary modifications of the available algorithms and mathematical descriptions to cover the form-finding aspects of the task were completed within six weeks (see [5] and [39]). The Linkwitz group had to built their software more or less from scratch. Because John H. Argyris was also head of the computer centre where the CDC 6600 was based, his group regarded the machine as their property and claimed priority over other users. The Linkwitz group had do more than one night shift to get their fair share of computing time on the machine. This attitude was perceived as arrogant, especially when an assistant of John H. Argyris told members of the Linkwitz group during a presentation: "Das glaubt ja sowieso keiner, dass Sie das können.".[3] Obviously, this remark was proved to be wrong (see for these paragraphs [19]).

The obstacles, the two groups had to overcome, could not have been more different. ASKA, with its detailed specifications of data formats and internal validations, would guarantee results of the required quality level nearly automatically. Only local modifications had to be made. The ad-hoc software of the Linkwitz group had not been specified in a formal way, was rather unstable, and produced many faulty results. Entering data into the system required a lot of manual work. A strict time-line and the limited resources of the programmers involved, sometimes forced Klaus Linkwitz to find unconventional solutions. Instead of investing more money and time into the improvement and debugging of the software, erroneous results would be eliminated manually. If there was no piece of code available to shuffle the matrix elements for an efficient calculation, a member of the team had to do it. However, during the late 1960s Germany was a prospering country and there was virtually no unemployment. Even workers with rather limited skills were sought after. When there were problems with the CDC 6600, he addressed the Bundeswehr (German Army) and asked successfully for a group of soldiers to manually check results, compile papers, and sort punch cards, to catch up with the time-line (see [19], [27] and [39]).

[3] "Anyway, nobody is believing, that you can do this." (translation by the author, quoted from [19])

After the contracts for the construction of the Olympic Stadium's roof had been fulfilled, the Force Density Method developed by the group of Klaus Linkwitz and the calculations of the Argyris group formed the basis of the Sonderforschungsbereich "Weitgespannte Flächentragwerke" (SFB 64) (special research group no. 64 Light-Weight Structures) which continued the research on the design of light-weight structures in architecture. In the following years both groups would co-operate more closely under the umbrella of this new organisational unit (see [19] and for a general overview of the evolution of structural analysis [26]).

6 THE PARADIGM(S) OF SUPERCOMPUTING

Supercomputing or High Performance Computing (HPC) is a world of its own. It is governed by specific system architectures, programming paradigms, and hardware technologies. All propelled by enormous amounts of money. Many regard the supercomputing community a special breed of people living a life of their own.

High performance computing is also an ambitious field of science and engineering. It promises new epistemic power to science at the price of hefty investments, which become within a short time of two or three years nearly obsolete. Belonging to the most powerful computers in the world is a quickly fading property wilting like a blossom. New systems enter the market and occupy the top ranks of supercomputer systems. Constantly pushing processing speed, storage capacity and interconnection bandwidth to the technological and physical limits provides not only an increase in quantity, but also in quality. Combined with powerful visualisation techniques, supercomputers are a means of gaining new insights in science and a way of solving sophisticated engineering problems.

It is easy to talk about supercomputers, but not so easy to define what the essence of a supercomputer is. Obviously, it is a machine which is very powerful in the sense that a gigantic number of operations can be executed in a very short time. One has to define a metrology to measure this power. The most salient measure is the number of operations executed per second. However, it is not clear of what nature these operations are. Are they instructions of the kind *copy one value in a memory cell to another cell* or should one talk about arithmetic operations such as addition and multiplication? As supercomputers are number crunching machines, the universal performance measure became the number of floating-point operations executed per second (FLOPS). Being able to execute a lot of floating-point operations very quickly is one characteristic of a supercomputer, but thousands of personal computers or workstations working together are also able to deliver such a level of performance. There is another aspect to supercomputing which makes supercomputers outstanding. It is their ability to deliver this performance in a sustainable manner on one single application. High performance computing is more than capacity computing, it is capability computing (see for these paragraphs [32] and [41]).

But what is the paradigm of supercompting? Or should one talk about many paradigms? Is there one paradigm which remained unchanged right from the beginning in the mid-1960s?

Depending on the aspects under consideration, there a many paradigms by which the history of supercomputers could be structured. The system architecture of such machines offers a fruitful access to classification. Early systems like the CDC 6600 already offered pipelined processing of data to increase processing speed. Later on, this paradigm became known as vector processing and even found its way into today's microprocessors. From a programmer's and user's point of view, the peculiarities of hardware design where mostly hidden by compilers and operating systems which provided the necessary vectorisation of the code (see [37, p. 94–95]). Another, more recent, paradigm is distributed processing and the emergence of high-speed networks, where large numbers of interconnected processors execute programs in parallel (see [11]). Such paradigms tend to shape the view of software developers and users regarding the structure of their problems. They are lured into adapting their interaction with the systems accordingly. In the world of non-supercomputers, system designers try to conceal the internal structure of computers and create programmer and user interfaces with a non-technical touch and feel. In the field of supercomputing, programmers and users are usually much closer to the system architecture. The only exception might be the visualisation of the results itself, where supercomputers and their specific architectures are hidden behind colourful images and animations.

6.1 The persistent paradigm of supercomputing

The paradigm of users adapting the structure of their problems to the specific architecture of their computers is the one paradigm which did not change since the beginning of supercomputing in the 1960s. In those days many regarded computers as over-sized calculators. Only a few recognised them as heralds of a new time, in which new insight into scientific and engineering problems could be gained by numerical calculations.

John H. Argyris provided one of the few personal reflections in this era and remarked:

> Our intellectual development and, if we may say so, also adventures have shown, however, that one's philosophy of approach to the computer may be more important than particular techniques, some approaches being more fruitful in terms of results than others. [...] But in appreciating the full potential and power of the computer and striving towards its most effective utilization, we have to learn to use it not as a large assembly of desk machines, but as a novel device which forces us to remodel our intellectual processes. In this continuous fight against established ideas we have to be driven by an instinctive understanding not to impose our past analytical models on the computer and to worry if they do not suit it, but to rethink our physical environment in newly invented models which are best suited to the world of computers. ([7, p. 395])

Important concepts of analytical mathematical physics, such as smoothness, continuity and singularities were put aside in favour of more a pragmatic and discrete view on physical problems. And in Argyris' "opinion and practice, the computer, when applied with intelligence and feeling, dictates the whole morphology of a theory and the necessary associated computer software." ([7, p. 395]) In his work, the theories and models became discrete and digital. The world itself was cut into little junks of bits and bytes, ready to be fed into a computer. "In short, we strive to mould the computer and the theory into a unified system for solving problems of any complexity." ([7, p. 396]) High-speed computers were regarded as a tools to solve problems of any degree of complexity and the price one had to pay, was to adapt to the structure and architecture of digital systems.

However, man was not made to deal with long columns of numbers that easily. This problem could be solved by using interactive graphical displays, which allowed the users to visualise results and to study them in an explorative way, more fit to the workings of the human intellect (see for these paragraphs [7]).

Software packages, having been in existence in numerous versions for more than forty years, designed with such principles in mind, do not follow the current trends of new programming languages, virtualisation and cloud computing, that try to abstract from the underlying system architecture. Such an approach would be too costly in terms of performance and would be opposite to the idea of pushing the limits, that is so deeply engraved into the minds of supercomputer designers and users (see [30]).

7 CONCLUSION

This historical case study on the implementation and deployment of two software packages for the design of the light-weight membrane roof of the 1972 Munich Olympic Stadium provided detailed insight into the way software for complex and ambitious projects has been written and used around 1970. It was shown that major principles of software engineering had already appeared independently in the field of structural analysis and engineering in the early 1960s.

Our case study could take advantage of a rare opportunity in history. An event could be examined were two, more or less competing, groups with different philosophies worked in the same environment, at the same place, on the same project, and using the same computer infrastructure. Usually, historical developments have to be studied in differing settings, making it difficult to compare the results. This case study could be conducted in a way usually only common to scientific experiments.

It was also demonstrated, that the paradigm of closely adapting the structure of the problems under study and the software representing them to the system architecture of computers forms an invariant element of the philosophy of supercomputing since more than forty years. This rather weak interaction with the developments in general computing and computer science makes high performance computing a special case worth to be studied in more detail.

ACKNOWLEDGEMENTS

The author would like to thank Klaus Hentschel, *Director of the Section for History of Science and Technology at the Historical Institute of the University of Stuttgart*, for the hospitality of his department.

REFERENCES

[1] Jörg Albertz, Hans-Peter Bähr, Helmut Hornik, and Reinhard Rummel, *Am Puls von Raum und Zeit: 50 Jahre Deutsche Geodätische Kommission*, number 26 in DGK, Reihe E, Deutsche Geodätische Kommission bei der Bayerischen Akademie der Wissenschaften, München, 2002.

[2] J. C. Almond, *Handbuch für Benutzer der CDC 6600-Großrechenanlage*, Regionales Rechenzentrum im ISD, Stuttgart, 1970.

[3] Karyn R. Ames and Alan Brenner, *Frontiers of Supercomputing II: A National Reassessment*, volume 12 of *Los Alamos Series in Basic and Applied Sciences*, University of California Press, 1994.

[4] Werner Ammon, *Schaltungen der Analogrechentechnik*, Oldenburg Verlag, München, 1966.

[5] J.H. Argyris, T. Angelopoulos, and B. Bichat, 'A general method for the shape finding of lightweight tension structures', *Computer Methods in Applied Mechanics and Engineering*, **3**(1), 135–149, (1974).

[6] John H. Argyris, 'ASKA – Automatic System for Kinematic Analysis: A universal system for structural analysis based on the matrix displacement (finite element) method', *Nuclear Engineering and Design*, **10**(4), 441–455, (1969).

[7] John H. Argyris, 'The impact of the digital computer on structural analysis', *Nuclear Engineering and Design*, **10**(4), 395–399, (1969).

[8] William Aspray, *John von Neumann and the Origins of Modern Computing*, The MIT Press, Boston, Massachusetts, 1990.

[9] Friedrich Ludwig Bauer, Peter Naur, Brian Randell, and NATO Science Committee, *Software engineering: report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, Scientific Affairs Division, NATO, 1969.

[10] Peter Beyer, *Letter and report to Matthias Kleinert regarding "Problemfall ASKA" dated 30th October 1984*, volume 108/61B, Universitätsarchiv Stuttgart, Stuttgart, 1984.

[11] David L. Black, *Supercomputer Systems-Software Challenges*, 147–154. Volume 12 of *Los Alamos Series in Basic and Applied Sciences* [3], 1994.

[12] Wolfgang Brand, 'Models, Experiments and Computing: A Historical Case Study of the Design of the Membrane Roof of the Munich Olympic Stadium using the first Supercomputers', *Philosophy & Technology*, submitted, (2013).

[13] Ioannis Doltsinis, 'Obituary for John Argyris', *Communications in Numerical Methods in Engineering*, **20**(9), 665–669, (2004).

[14] Michael Friedewald, *Der Computer als Werkzeug und Medium: Die geistigen und technischen Wurzeln des Personalcomputers*, Aachener Beiträge zur Wissenschafts- und Technikgeschichte des 20. Jahrhunderts, Bd. 3, GNT-Verlag, Diepholz, 2 edn., 2009.

[15] Hannah Gay, *The History of Imperial College London, 1907-2007: Higher Education and Research in Science, Technology, and Medicine*, Imperial College Press, London, 2007.

[16] INTES GmbH. INTES GmbH Stuttgart, http://www.intes.de, Last access: 21th May 2012.

[17] Ingolf Grieger, 'In memoriam: John Argyris', *Stuttgarter unikurier*, **93**, (2004).

[18] Lothar Gründig, *Berechnung vorgespannter Seil- und Hängenetze*, volume 216 of *Deutsche Geodätische Kommission, Reihe C: Dissertationen*, Verlag der Bay. Akademie der Wissenschaften, München, 1976.

[19] Lothar Gründig. Interview by the author on the 7th June 2011 in Berlin, Germany, 2011.

[20] Lothar Gründig, Erik Moncrieff, Peter Singer, and Dieter Ströbel, 'A History of the Principal Developments and Applications of the Force Density Method in Germany 1970–1999', In Papadrakakis et al. [38].

[21] Andrew Hodges, *Alan Turing: Enigma*, Springer Verlag, Berlin, 1994.

[22] Kenneth H. Huebner, Donald L. Dewhirst, Douglas E. Smith, and Ted G. Byrom, *The Finite Element Method for Engineers*, John Wiley & Sons, Inc., New York, 2001.

[23] Thomas J.R. Hughes, J. Tinsley Oden, and Manolis Papadrakakis, 'In memoriam to Professor John H. Argyris: 19 August 1913 – 2 April 2004', *Computer Methods in Applied Mechanics and Engineering*, **193**(36-38), 3763–3766, (2004).

[24] Falk Jaeger, 'Frei Otto. Ingenieurportrait. Architekt, Konstrukteur und Visionär, Förderer der Leichtbauweise', *db Deutsche Bauzeitung*, **139**(6), 72–77, (2005).

[25] Walter Kaiser, 'Tunnelbau mit dem Computer – Die Finite-Elemente-Methode in der Geotechnik', *Ferrum – Nachrichten aus der Eisenbibliothek*, **80**, 77–92, (2008).

[26] Karl-Eugen Kurrer, *The History of the Theory of Structures: From Arch Analysis to Computational Mechanics*, Ernst & Sohn, Darmstadt, 2008.

[27] Klaus Linkwitz, *Letter to brigadier general Friedrich regarding "Aushilfsdienst bei Berechnungsarbeiten für olympische Dächer München" dated 27th April 1970*, volume 14/49B, Universitätsarchiv Stuttgart, Stuttgart, 1970.

[28] Kristina M. Luce, *Revolutions in Parallel: The Rise and Fall of Drawing in Architectural Design*, Ph.D. Thesis, The University of Michigan, Department of Architecture, 2009.

[29] Theo Lutz, *Der Rechnerkatalog: 100 Computer mit ihren Kenndaten*, Telekosmos-Verlag, Stuttgart, 1966.

[30] Christoph Meinel, Christian Willems, Sebastian Roschke, and Maxim Schnjakin, *Virtualisierung und Cloud Computing: Konzepte, Technologiestudie, Marktübersicht*, Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam, Universitätsverlag Potsdam, 2011.

[31] H. P. Mlejnek, *Letter to Jürgen Blum regarding "INTES, ASKA Lizenzvertrag" dated 14th December 1984*, volume 108/61B, Universitätsarchiv Stuttgart, Stuttgart, 1984.

[32] Charles J. Murray, *The Supermen: The Story of Seymour Cray and the Technical Wizards behind the Supercomputer*, John Wiley & Sons, Inc., New York, 1997.

[33] N. N., *Die Rechenanlagen der Universität Stuttgart: Ein Bericht über den gegenwärtigen Stand ihres Einsatzes*, Universität Stuttgart, Stuttgart, 1971.

[34] N. N., *Die Spiele: The official report of the Organizing Committee for the Games of the XXth Olympiad Munich 1972. The Organization.*, volume 1, pro Sport, München, 1972.
[35] N. N., *Olympische Bauten München 1972: Bauabschluss Sommer 1972*, volume 3 of *Architektur-Wettbewerbe/Sonderhefte*, Krämer Verlag, Stuttgart, 1972.
[36] N. N., *Lebenslauf John H. Argyris*, volume 108/31B, Universitätsarchiv Stuttgart, Stuttgart, 1991.
[37] Stan Openshaw and Ian Turton, *High Performance Computing and the Art of Parallel Programming: An Introduction for Geographers, Social Scientists, and Engineers*, Routledge, London, 2000.
[38] M. Papadrakakis, A. Samartin, and E. Onate, eds. *Proceedings of the Fourth International Colloquium on Computation of Shell and Spatial Structures (IASS-IACM), June 4–7, Chania-Crete, Greece, (CD-ROM)*, Athens, Greece, June 2000. Institute of Structural Analysis & Seismic Research, National Technical University of Athens.
[39] Marios C. Phocas, 'John Argyris and his decisive Contribution in the Development of Light-Weight Structures. Form follows Force.', in *Proceedings of the 5th GRACM International Congress on Computational Mechanics (GRACM05), Limassol 29 June–1 July, Cyprus*, eds., Georgios Georgiou, Panos Papanastasiou, and Manolis Papadrakaki, volume 1, pp. 97–103, Nicosia, (2005). Kantzilaris Publications.
[40] N. Rech, *Letter to A. H. Loescher regarding "Planung für das Rechenzentrum Stuttgart" dated 20th March 1968*, volume 107/79A, Universitätsarchiv Stuttgart, Stuttgart, 1968.
[41] Roland Rühle. Interview by the author on the 10th May 2011 in Dachtel, Germany, 2011.
[42] Kay Schiller and Christopher Young, *The 1972 Munich Olympics and the Making of Modern Germany*, Weimer and Now: German Cultural Criticism, University of California Press, 2010.
[43] Mary Shaw, *Position paper for Dagstuhl Workshop on Software Architecture, August 1996: Three Patterns that help explain the development of Software Engineering*, Computer Science Department, Carnegie Mellon University, Pittsburgh, March 1997.
[44] Richard Southern, *The Force Density Method: A Brief Introduction*, Technical Report TR-NCCA-2011-02, The National Centre for Computer Animation, Bournemouth University, Poole, UK, 2011.
[45] technet GmbH. technet GmbH, http://technet-gmbh.com, Last access: 21th May 2012.
[46] Niklaus Wirth, 'A brief history of software engineering', *IEEE Annals of the History of Computing*, **30**(3), 32–39, (2008).
[47] Christian Wurster, *Computer: Eine illustrierte Geschichte*, Taschen Verlag, Köln, 2002.
[48] F. W. Zeitler, *Letter to J. H. Argyris regarding ASKA usage rights for USA and Australia dated 9th April 1970*, volume 107/79B, Universitätsarchiv Stuttgart, Stuttgart, 1970.
[49] Konrad Zuse, *Der Computer: Mein Lebenswerk*, Springer Verlag, Berlin, 3 edn., 1993.

On the Cognitive Science of Computer Programming in Service of Two Historic Challenges

Selmer Bringsjord[1] and **Jinrong Li**[2] and **Naveen Sundar G.**[3] and **Konstantine Arkoudas**[4]

Abstract. We describe two historic and hard programming challenges: the educational challenge of getting humans to effectively program from scratch, and essentially the same challenge applied to computing machines. We review work done in both these domains, and then present our views on what could constitute acceptable solutions for these two related problems. Our view is that for the educational challenge, formal training in logic is not only sufficient but also necessary. The state of the art in solving the second challenge also indicates that any methodology which shies away from building machines that can robustly handle general-purpose formal reasoning has bleak chances for success. A pilot study supporting our stance is presented. We conclude by describing our plans for a second larger experiment that we predict will strengthen our stance.

1 INTRODUCTION

Despite decades of study and development of systems for teaching computer programming to humans (e.g., Logo), and of programs that automatically write programs (in the field of *automatic programming*, hereafter just AP), the process of designing and generating precise instructions "from scratch" for computers remains both exceedingly difficult for most humans (even for those who are exposed to these systems under seemingly ideal conditions), *and* for all computing machines. We have inaugurated a research project designed to change this situation dramatically for the better.

What do we mean by 'from scratch'? One can sensibly distinguish two types of programming. The first we label 'programming$_1$.' Here one receives an explicit algorithm or pseudo-code as input, and produces code that when executed computes the algorithm. For example, given a detailed description of the merge-sort algorithm, one proceeds to code the algorithm in, say, Java. To do this, one basically only needs to learn how to map from pseudo-code to the syntax and basic control structures of Java; very little problem-solving is needed, and one doesn't need to have even an intuition as to whether the resultant code is provably correct. The other, deeper kind of programming is programming$_2$; this is what we allude to by 'from scratch.' Here one receives only an abstract, non-executable description of a function f in for instance the kind of hybrid natural-language/formal-language content traditionally used in a mathematics textbook or article, and then must *create* solutions that include algorithms, and then proceed to code these solutions in some programming language; that is, produce a working computer program that computes the function. When a human truly programs$_2$, she is at least in position to genuinely grasp when and why some computer program is correct.

With this terminology in hand, the chief challenges our research project is intended to meet can be stated.

hypotheses can be stated. We hypothesize but (ii) that those humans who *are* crack programmers$_2$ hold the secrets to writing programs that can program$_2$. Our research project is designed to test (i) and leverage (ii) in order to advance the field of AP. But our project is also intended to meet two challenges that stem from hypotheses (i) and (ii).

2 THE TWO CHALLENGES WE SEEK TO MEET

Our project can be viewed as an attempt to meet two challenges, to wit:

1. The Educational Challenge (**E**): How can we best bring it about that humans (particularly young ones) learn how to effectively program$_2$, where the given input function f is non-trivial and perhaps even quite complicated? This is a severe challenge, because apparently even US college students with high mathematical aptitude (as measured by SAT scores) in technical majors (including computer science itself!) who have been exposed to Logo and other such systems before college are for the most part nonetheless unable to program$_2$.
2. The Technological Challenge (**T**): How can we bring it about that computing machines (which can of course without loss of generality themselves be viewed as programs) program$_2$, where the input functions are, again, non-trivial and perhaps complicated? And how can we build computing machines that not only generate computer programs that compute functions given as input, but also establish that the programs they produce are correct? That is, how can we create *self-verifying* automatic programming programs? The challenge here is like its predecessor quite severe, given the sad state of AP.

At this point we provide further context on **T**, by giving an overview of AP.

3 BRIEF OVERVIEW OF AUTOMATIC PROGRAMMING

Put simply, AP is the field devoted to writing computer programs smart enough to write significant computer programs, from — as

[1] Dept. of Computer Science and Dept. of Cognitive Science, Rensselaer AI & Reasoning Laboratory, Troy NY 12180 USA, Contact Author Email: selmer@rpi.edu, Web site: http://www.rpi.edu/∼brings
[2] Dept. of Computer Science and Dept. of Cognitive Science, Rensselaer AI & Reasoning Laboratory, Troy NY 12180 USA
[3] Dept. of Computer Science and Dept. of Cognitive Science, Rensselaer AI & Reasoning Laboratory, Troy NY 12180 USA
[4] Department of Cognitive Science, Rensselaer Polytechnic Institute (RPI), Troy NY 12180 USA

we've of course said — scratch. By and large, AP has not exactly made impressive strides over the last three decades.[5]

The aims of AP have fluctuated considerably over the decades, as has been pointed out by Rich [14] and others. In the 1950s, the mechanical compilation of Fortran programs into machine code was viewed as "automatic programming." In the 1960s, at the dawn of AI, and in keeping with the rather ambitious dreams of this new and exciting field, a much more lofty goal was set for the field — the black-box version of AP, whereby only a non-executable description of the desired relationship between input and output is provided to the synthesis software, and the latter then emits an executable computer program that realizes the given specification.

There are several degrees of freedom in the above scheme. First, there are several choices concerning the medium in which we express our description; for example:

1. *Natural language*: Ideally, we would be able to simply tell the machine in a natural language such as English what we want the program to do. In practice, this is not feasible currently. One could of course restrict the input language to, say, a controlled subset of English, but such subsets are in fact formally defined.
2. *Formal specification*: The description could be expressed in a high-level declarative formal system, such as first-order logic. It would be assumed that the formal specification is sound and complete with respect to our informal requirements.
3. *Input-output examples*: The "description" could be *illustrated* by way of input/output pairs. This would be necessarily incomplete, as there will always be infinitely many programs that cover any presented finite set of examples.
4. *Hybrid representations*: Here there is a mix of the above, and also a reliance on diagrammatic or visual representations.

Most of the work on AP so far has adopted one of the second and third of these approaches. Typically, inductive techniques synthesize programs from input-output examples, whereas deductive techniques synthesize programs from formal specifications.

The following represent three of the most prominent threads of research in the inductive camp of AP:

1. *Recurrence Detection*. The seminal work in this area was carried out by Summers [18]. It is one of the most psychologically plausible approaches in the field. Summer's ideas have been extended, most notably in the 1980s by Kodratoff [6] and Wysotzki [19], who augmented the basic scheme outlined above. Similar techniques have been used for "programming-by-demonstration" systems such as Tinker [9]. Kitzelmann [5] and others are continuing this line of work, but results so far have been limited.
2. *Genetic Programming*. Genetic programming (GP) [7] was discovered in the 1980s (although the general idea of genetic algorithms goes back to the 1950s). The main idea of GP is the following:

   - Start with a population (say 10^4) of random computer programs. Typically programs are purely functional, often expressed in pure LISP, and represented as ASTs (abstract syntax trees).
   - Assign a fitness value to each program.
   - Create a new population by performing "genetic operations" on selected programs, most notably crossover and mutation.

   This loop is continued until some program in the current population achieves a satisfactory fitness, or until a maximum number of iterations has been made. An important point is that the programs to be operated on are selected with probability proportional to their fitness.
   The most common genetic operations are the following: Usual operations:

   - *Mutation* (on one program only): Randomly alter part of a program's structure.
   - *Crossover* (on two programs): Randomly shuffle two parts of the two programs.
   - *Reproduction* (on one program): Simply carry over a program unchanged into the new population.

   Crossover is the most important of these three operations, and performed most frequently (with the greatest probability).
   GP is generally well-suited for optimization and control problems, and for games. Unfortunately, it's too computationally intensive. Evaluating the fitness of programs entails evaluating the programs themselves, which can be very time-consuming. ADATE [11], for instance, one of the most prominent AP systems based based on genetic programming, takes 6.5 days to evolve a program for list intersection. Like the other approaches, genetic AP techniques have not scaled to realistic programs. In addition, and in contrast to inductive logic programming, genetic programming is very weak on understanding and explanation. Typically the generated programs are horribly convoluted spaghetti code. (Although one can mitigate that to a certain extent via simplification, and by making program structure and succinctness part of the fitness function.) As a result, genetic programming is the least cognitively plausible of all well-known methodologies for AP.
3. *Inductive Logic Programming*. Inductive logic programming (ILP) [10] synthesizes logic (rather than functional) programs. The input to the synthesis process consists of:

   (a) A background theory B.

   (b) A set of positive examples E^+ (almost always atoms).

   (c) A set of negative examples.

   The following requirements are imposed:[6]

   (a) $\forall\, e^- \in E^- \,.\, B \not\models e^-$

   (b) $\neg \forall\, e^+ \in E^+ \,.\, B \models e^+$

   The output is a hypothesis h such that:

   (a) $\forall\, e^+ \in E^+ \,.\, B \wedge h \models e^+$

   (b) $\forall\, e^- \in E^- \,.\, B \wedge h \not\models e^-$

   Of course the conjunction of all the positive examples is a trivial solution, but what we are really after is predictive power — the generated hypothesis should do well on *unseen* data.
   The basic algorithm of ILP is to start with a very specific hypothesis and keep generalizing; or, alternatively, to start with a very general hypothesis and keep specializing. Various combinations are also possible.
   Many successful ILP systems view induction as the inverse of deduction, and form hypotheses by inverting deductive inference rules. A typical inference rule is absorption:

[5] The 'from scratch' phrase is the catch. Great progress has been made in the semi-automatic realm, where code is e.g. generated from libraries of pre-existing code, and from pre-engineered algorithms for code generation from predictable triggers.

[6] We write $\Phi \models p$ to indicate that the set of formulas Φ logically implies the formula p.

$$\frac{A \Rightarrow q \quad A, B \Rightarrow p}{A \Rightarrow q \quad B, q \Rightarrow p} \quad [\textit{Absorption}]$$

The conclusion here logically entails the premises.

While ILP has been successful in data mining, in automatic programming the results have been underwhelming. There have been no remarkable programs generated beyond the usual toy examples (list reversal, etc.). In addition, the generated programs are often quite inefficient. In fact, the method itself is inefficient for recursive programs, since testing examples requires running arbitrary code.

In deductive program synthesis, the input is a formal specification of the desired relationship between the input and output, expressed either in first- or higher-order logic, or else in a very high-level specification language; and the output is a program, typically in a functional language, such as the purely functional subset of Lisp, that is guaranteed to terminate and to satisfy the specified relationship. The guarantees are in the form of formal proofs.

Much of the work in this field has been carried out in the context of constructive logic, whereby the program is extracted from a constructive proof asserting the existence of a suitable output (i.e., an output that meets the specification). Nevertheless, it is not strictly necessary to use constructive logic per se, and indeed some of the most seminal work in this vein took place in the backdrop of classical first-order logic. At any rate, the main idea in either case is the same: Given the formal specification and a background theory that axiomatizes the relevant domain (e.g., a theory of lists or trees), we attempt to construct a proof that the desired function satisfies the given specification.

More precisely, let S denote the given specification:

$$\forall\, x : I, y : O \;.\; S(x, y) \tag{1}$$

where x and y range over the input and output domains, respectively, I and O. In the interest of flexibility, we do not require the specification to be functional. That is, for any given input x, there may be zero, one, or multiple outputs y that bear the desired relationship to x. Often the specification $S(x, y)$ is of the form

$$\mathit{Pre}(x) \Rightarrow \mathit{Post}(x, y), \tag{2}$$

asserting that if the input x satisfies a certain precondition, then the output y is related to x in accordance with some desired postcondition.

The goal is to synthesize a computable definition of a function

$$f : I \rightarrow O$$

for which the following holds:

$$\forall\, x : I \;.\; S(x, f(x)). \tag{3}$$

In particular, when the specification S is of the form (2), the desired condition can be equivalently rewritten as follows:

$$\forall\, x : I \;.\; \mathit{Pre}(x) \Rightarrow \mathit{Post}(x, f(x)). \tag{4}$$

There are several drawbacks to the deductive approach:

- The approach requires the user to submit a formal specification of the relationship between the inputs and desired outputs. But writing such specifications can often be just as challenging as writing a program to compute the desired function.
- The approach depends crucially on the state of the art in theorem proving. Unfortunately, theorem proving is an extremely challenging problem, and while there has been some progress in the field, current capabilities fall well short of what would be required for automated deductive synthesis of realistic programs.
- Typically, the generated programs are purely functional and often quite inefficient. In principle, more efficient versions could then be successively obtained by applying suitable program transformations, but in practice this prospect also runs up against the limited capabilities of theorem-proving systems.

Nevertheless, the deductive approach could prove feasible in the setting of interactive theorem proving, provided that the amount of required human guidance could be kept at a minimum. Moreover, a deductive synthesis module might be a useful (perhaps indispensable) component of a larger synthesis system that combines inductive and deductive techniques. We suspect that such a combination would be reflective of how expert human programmers generate programs in non-trivial cases, but additional empirical work on our part is necessary to test our suspicion, let alone concretize it in a working AP system. At any rate, in order for challenge **T** to be met, some strong contribution from deduction must be made, since the only way, by definition, to prove that some AP-generated code does in fact compute the function it's intended to is to use formal logic — and ultimately to use proof-checking technology (discussed in [1]).

3.1 TWO APPROACHES EDUCATIONAL CHALLENGE E

There have of course been many attempts to meet both challenges. In our planned presentation and demonstration at HAPOP 2012, we discuss two fundamentally different approaches to the challenge **E**:

1. Approach $\mathcal{C}$ (in honor of *constructivism*): This is not unfairly called "learning by doing." This approach holds that humans generate knowledge and meaning from an interaction between their concrete attempts to build, and reaction to the results of these attempts. So, children on this approach supposedly best learn to program$_2$ by diving into examples, without first receiving training about the formal essence of programs and their formal idealizations (e.g., Turing machines), and without being given any background in naïve set theory to aid in achieving a deep understanding of programs. Often approach $\mathcal{C}$ is accompanied by a concerted effort to engage the student with entertaining window dressing (e.g., turtles and other cute animals) that has rather little to do with the formal essence of a program or the function(s) that it computes.
2. Approach $\mathcal{R}$ (in honor of *abstract reasoning*): Learning by deliberative reasoning, applied to progressively more difficult problems, until the learner reaches at least Stage IV in Piaget's [4] continuum for cognitive development. At IV and beyond the learner can reason at the level of full first-order logic, and can generate and assess hypotheses expressed in FOL. $\mathcal{R}$ is based on the idea that the best way to learn to program$_2$ is to gradually come to understand essential theoretical concepts, and on the idea that under the right conditions some humans can indeed reach Stage IV and beyond [15, 2]. $\mathcal{R}$ also suggests that programming languages and environments should themselves be reflective of formal logic (e.g., see [3]).

4 APPROACHES TO CHALLENGE T

It's beyond the scope of this extended abstract to any more than summarize the approaches that have been taken to **T** (as was done in §3). Bringsjord (along with Konstantine Arkoudas) carried out a comprehensive review of the status of AP for the US National Science Foundation. The upshot, which won't come as a surprise to anyone, and which is conveyed by §3 herein, is that essentially no progress has been made on automatic programming, despite the fact that it was one of the original dreams of AI. Note again that since we are interested in automatic programming where the programs produced by programs are known to be correct (in the sense that they provably compute the input functions), evolutionary techniques that generate programs which for all we know may or may not compute the input functions are by definition inadequate.

4.1 OUR TWO CLAIMS

Now, given the foregoing, we make a pair of clams that we hope to articulate and defend at HAPOP 2012:

1. Approach $\mathcal{C}$ has not, and will not, succeed in allowing us to solve **E**. Approach $\mathcal{R}$, on the other hand, will allow us to meet **E**.

   There has been much work done under $\mathcal{C}$. The "grandfather" exemplar of this approach is Logo, created by Bobrow, Feuzeih, and Papert in 1967; Logo has inspired and influenced many other systems: Etoys, Scratch, NetLogo, KTurtle, REBOL, etc. These systems generally provide a graphical environment for children to learn by discovery themselves. Although there is some evidence that learning Logo improves problem-solving and thinking skills [17, 12], the vast majority of researchers outside the Logo/constructivist fold find little to be encouraged by. For example, no greater planning skills have been found for students who had a year of Logo programming, compared with those who did not (e.,g., see [8, 13]).

   In our Experiment 1, The results, for a group of 120 subjects of the afore described type,[7] included that less than 3% came close to solving one of the two programming problems they received, and less than 1% provided *bona fide* solutions to both programming problems.

2. Engineering approaches to meeting the challenge **T** that are analogous to $\mathcal{C}$ (e.g., evolutionary machine-learning approaches) will not allow us to solve $\mathcal{T}$. On the other hand, if we can understand the nature of the deep reasoning and creativity in exceptional individuals who are adept at programming$_2$, we can profitably apply this understanding to the engineering of programs that meet challenge **T**. We concede that the graph shown in Figure 1 is here painfully relevant.

   In a second phase of our pilot study we considered only those students with training in formal logic. Out of a group of 30 subjects who had taken introductory mathematical logic, about 20% solved both questions correctly.

[7] US college students with high mathematical aptitude (as measured by SAT scores) in technical majors (including computer science itself!) who have been exposed to Logo and other such systems before college.

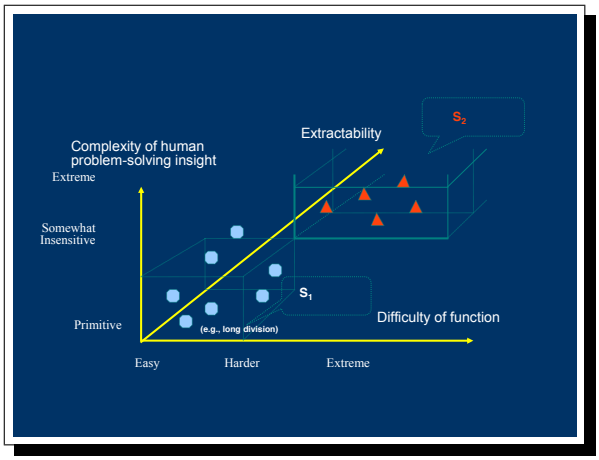

Figure 1. 3D View of Degree of Human Insight, Difficulty of Underlying Function to be Computed by Discovered Program, and Ease of Extracting the Nature of Human Insight. Problems from both "Easy" (S_1) and "Difficult" (S_2) spaces are indicated. In the case of S_1, not much human insight is needed, because the underlying function to be coded is simple; and hence it's relatively easy as well to extract an account of how the human proceeded to success. By contrast, S_2 is difficult on all three dimensions.

5 EXPERIMENT 1 SUMMARY

Subjects were asked to answer a set of questions related to their educational backgrounds with respect to math, logic, and computer programming. Then they attempted to solve two programming problems, using a programming language of their choice, or using pseudo-code. The sections were conducted both online and in the classroom.

5.1 DESCRIPTION OF STIMULI IN EXPERIMENT 1

Both programming problems involved working with a very simple language, $\mathcal{L}$, a fragment of English. The words used in $\mathcal{L}$ are:

$$\{\mathsf{Bill}, \mathsf{Jane}, \mathsf{likes}, \mathsf{chases}, \mathsf{makes}, \mathsf{a}, \mathsf{the}, \mathsf{man}, \mathsf{woman}, \mathsf{cat}, \mathsf{happy}, \mathsf{thin}, \mathsf{quickly}\}$$

Bill, Jane, man, woman, and cat, are nouns; happy and thin are adjectives; likes, chases, and makes are verbs; a and the are determiners; and quickly is an adverb.

The following grammar defines the sentences of $\mathcal{L}$.

S	::=	NOUN VERB NOUN	R1
	|	DET NOUN VERB NOUN	R2
	|	DET ADJ* NOUN VERB DET ADJ* NOUN	R3
	|	DET ADJ* NOUN ADV VERB DET ADJ* NOUN	R4
	|	DET ADJ* NOUN VERB DET ADJ* NOUN ADV	R5

Where NOUN stands for any noun, VERB stands for any verb, DET stands for any determiner, ADV stands for any adverb, ADJ stands for any adjective, ADJ* stands for zero, one, or more adjectives, and S stands for a well-formed sentence of $\mathcal{L}$.

For instance, the sequence $\langle \mathsf{the}, \mathsf{thin}, \mathsf{cat}, \mathsf{makes}, \mathsf{a}, \mathsf{Bill} \rangle$ is a sentence of $\mathcal{L}$, because cat and Bill are nouns, likes is a verb, the and a are determiners, and thin is an adjective, so the sequence has the form DET ADV* NOUN VERB DET ADJ* NOUN, which, by rule

R3, is a sentence of $\mathcal{L}$. (Notice that the first ADJ* is matched with the one adjective thin, while the second ADJ* is matched with the lack of adjectives between a and Bill.)

In each of the two problems subjects were asked to write a program by using a language such as: BASIC, C, C++, Java, Lisp, Pascal of their choice; or pseudo-code. The two problems were quite straightforward:

5.1.1 *Problem 1*

In Problem 1 in Experiment 1, subjects were asked to write a program P that takes as input a (finite) sequence of words used in $\mathcal{L}$ and outputs **yes** if the sequence is a sentence of $\mathcal{L}$, and outputs **no** otherwise. For example, given the sequence ⟨Bill, likes, Jane⟩, P should output **yes** because the sequence is a sentence, according to R1. When given ⟨Bill, Jane, likes⟩, P should output **no**, because this sequence is not a sentence of $\mathcal{L}$.

5.1.2 *Problem 2*

In Problem 2 of Experiment 1, subjects were asked to Write a program P that takes as input a (finite) sequence of words used in $\mathcal{L}$ and outputs **yes** if the sequences is a palindrome sentence of $\mathcal{L}$, and outputs **no** otherwise. A palindrome sentence is a sentence which reads the same in both directions. For example, given the sequence ⟨Bill, likes, Bill⟩, P should output **yes**, since the sequence is a palindrome sentence. When given ⟨the, cat, likes, Jane⟩, P should output **no**, since the sequence, although a sentence, is not a palindrome sentence.

5.2 RESULTS

We give a brief summary of results directly relevant to the claims we made above relative to the two challenges **E** and **T**.

There were two groups of subjects: Group 1: a random group of students who sign up for the test. Total 206 Subjects (183 online, 23 on paper). Among them 85.7% never took a logic class, 35% had programming language classes. A slim 2.3% succeeded.

Group 2: 50 students who were in their second semester of a formal logic lass; 20% (10 out of 50) succeeded in solving of both problems (see Figure 2).

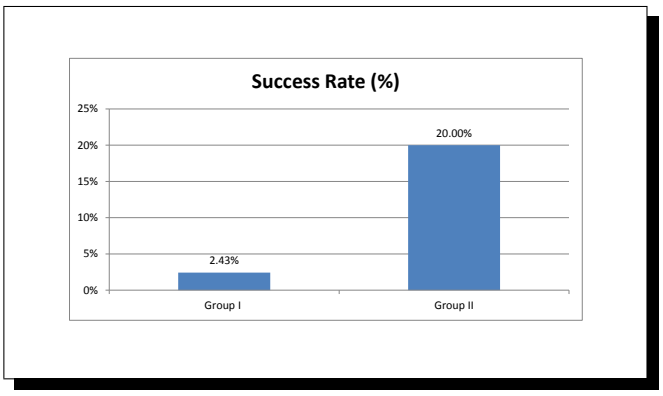

Figure 2. Experiment 1 Result. Group I: Naïve Group; Group II: Logic-Class Group

We concede that Experiment 1 is fundamentally only a pilot study. Our next experiment, Experiment 2, presents subjects with more complicated functions to be computed by the programs these subjects create. In addition, we will be looking systematically at how those select few who succeed managed to do so, so that we can perhaps gain further insight into whether we are right that abstract reasoning ability is key, and whether the "secrets" to success suggest better techniques in the attack on **T**. We give now a brief account of Experiment 2.

6 PLAN FOR A SECOND EXPERIMENT

Since the success rate for the general group in Experiment 1 was stunningly low, we are designing a second experiment which we believe is easier than the first experiment, but still hard enough to understand the impact of formal logic training in simple programming.

In the second experiment, we first provide text familiarizing subjects to three logic gates: $\{and, or, not\}$, Figures 3, 4 and 5, and a programmable logic array.

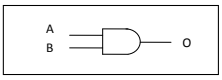

Figure 3. *And* Gate

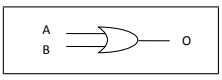

Figure 4. *Or* Gate

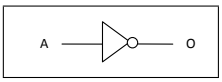

Figure 5. *Not* Gate

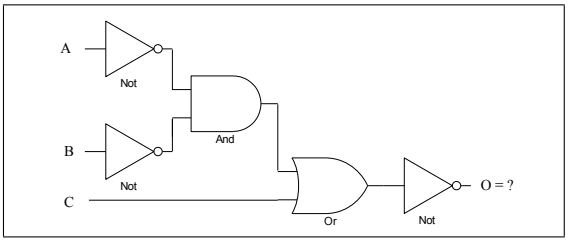

Figure 6. A Simple Logic Circuit

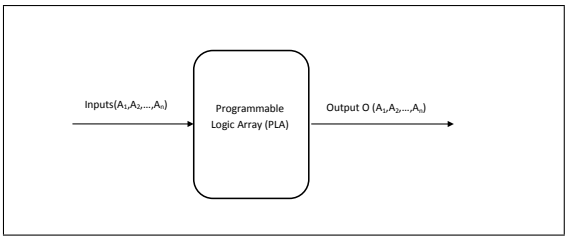

Figure 7. A PLA Block

A programmable logic array (PLA) is a an array of $\{and, or, not\}$ logic gates physically arranged in the disjunctive normal form which

can be customized to compute functions by modifying connections to produce new outputs. The experiment consists of three problems. In the first problem, a logic circuit C is given, for instance Figure 6, and the objective is to write a program m which computes the function f_C that the circuit computes C.

In the second problem, a PLA $P(O_1, \ldots, O_n)$ such as the one in Figure 8 is given, and the goal is to write programs $m_1, \ldots, m_n$ for the all functions $O_1, \ldots, O_n$ output by the PLA.

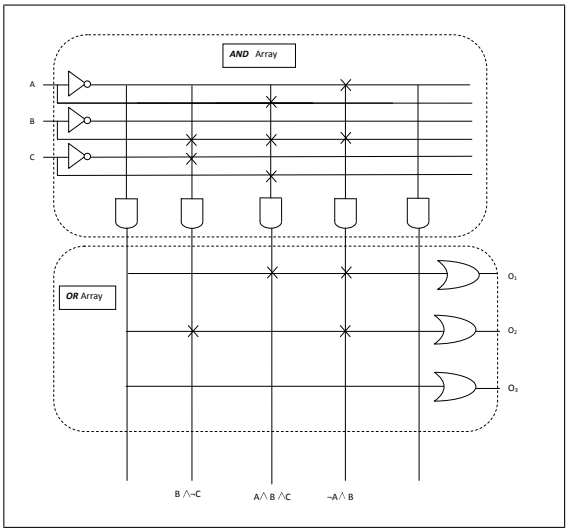

Figure 8. Schematic Diagram of a PLA Block's Internals

In the third problem, input-output descriptions of functions $O_1, \ldots, O_n$ are given in the language of the propositional calculus (see example below) in disjunctive normal form, and the goal is to provide a PLA $P(O_1, \ldots, O_n)$ that can compute the functions.

$$O1 = A \vee (\neg B \wedge C)$$
$$O2 = (A \wedge \neg C) \vee (A \wedge B)$$
$$O3 = (\neg B \wedge \neg C) \vee (A \wedge B)$$
$$O4 = (\neg B \wedge \neg C) \vee A$$

Despite having visual elements to them, we note that these problems fall into programming space S_1, while those in the first experiment fall into programming S_2. [8]

7 CONCLUSION

Of course, we can only offer a temporary conclusion, since our research project is still embryonic. Overall, we conclude, humbly, that we seem to be making some small progress toward meeting challenges **E** and **T**, and vindicating our core, driving claims. We are far from being able to offer students an efficacious learn-to-program environment rooted in a formal approach; and we are far from being able to contribute to AP on the strength of dissecting human programming$_2$ ingenuity. But we press on.

There is in particular much room for experiments that present subjects with *extremely* difficult programming problems, and it's perhaps in analyzing the cognition constitutive of solving such problems that real fruit for the advance of AP will be found. These problems would presumably be those which are such that, as far as anyone can tell, infinitary concepts and constructions are necessary as the human moves toward producing a program. We do have such examples in formal logic. For example, there currently is no finitary way of proving that certain theorems (e.g., Goodstein's Theorem) which are independent of Peano Arithmetic (therefore making PA incomplete) are nonetheless true and provable. It would be very interesting, and perhaps quite revealing, to pose programming problems that require the kind of computational ingenuity required to see that Goodstein's Theorem holds. [9]

REFERENCES

[1] Konstantine Arkoudas and Selmer Bringsjord, 'Computers, justification, and mathematical knowledge', *Minds and Machines*, **17**(2), 185–202, (2007).

[2] S. Bringsjord, E. Bringsjord, and R. Noel, 'In Defense of Logical Minds', in *Proceedings of the 20th Annual Conference of the Cognitive Science Society*, 173–178, Lawrence Erlbaum, Mahwah, NJ, (1998).

[3] S. Bringsjord and J. Li, 'Toward Aligning Computer Programming with Clear Thinking via the REASON Programming Language', in *Current Issues in Computing and Philosophy*, eds., K. Waelbers, B. Briggle, and P Brey, 156–170, IOS Press, Amsterdam, The Netherlands, (2008).

[4] B. Inhelder and J. Piaget, *The Growth of Logical Thinking from Childhood to Adolescence*, Basic Books, New York, NY, 1958.

[5] E. Kitzelmann, U. Schmid, and L. P. Kaelbling, 'Inductive synthesis of functional programs: An explanation based generalization approach', *Journal of Machine Learning Research*, **7**, 429–454, (2006).

[6] Y. Kodratoff, M. Franová, and D. Partridge, 'Why and how program synthesis?', in *AII*, pp. 45–59, (1989).

[7] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, 1992.

[8] D. Midian Kurland, Catherine A. Clement, Ronald Mawby, and Roy D. Pea, *Mapping the Cognitive Demands of Learning to Program*, 103–127, Ablex Publishing Corp., Norwood, NJ, USA, 1987.

[9] H. Lieberman, 'Tinker: a programming by demonstration system for beginning programm ers', 49–64, (1993).

[10] S. Muggleton, 'Inductive logic programming', in *Inductive Logic Programming*, ed., S. Muggleton, 3–27, Academic Press, London, (1992).

[11] J. R. Olsson, 'The art of writing specifications for the ADATE automatic proogramming system', in *Proceedings of the Annual Genetic Programming Conference*, eds., J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, and H. Iband R. Riolo, pp. 278–283, San Francisco, CA, USA, (1998). Morgan Kaufmann.

[12] Seymour Papert, *Mindstorms: Children, Computers, and Powerful Ideas*, IBM Corp., Riverton, NJ, USA, 1980.

[13] Roy D. Pea and D. Midian Kurland, *On the Cognitive Effects of Learning Computer Programming*, 147–177, Ablex Publishing Corp., Norwood, NJ, USA, 1987.

[14] Charles Rich and Richard C. Waters, 'Automatic programming: Myths and prospects', *Computer*, **21**(8), 40–51, (August 1988).

[15] K. Rinella, S. Bringsjord, and Y. Yang, 'Efficacious Logic Instruction: People are not Irremediably Poor Deductive Reasoners', in *Proceedings of the Twenty-Third Annual Conference of the Cognitive Science Society*, eds., J. D. Moore and K. Stenning, 851–856, Lawrence Erlbaum Associates, Mahwah, NJ, (2001).

[16] Peter Smith, *An Introduction to Gödel's Theorems*, Cambridge University Press, Cambridge, UK, 2007.

[17] Cynthia J. Solomon and Seymour Papert, 'A Case Study of a Young Child Doing Turtle Graphics in LOGO', in *Proceedings of the June 7-10, 1976, national computer conference and exposition*, AFIPS '76, pp. 1049–1056, New York, NY, USA, (1976). ACM.

[8] A semi-formal argument for this classification goes as follows: grammar descriptions of the sort given in the first experiment, e.g. context free grammars, can represent a wider class of functions than what is possible with a PLA. This is because a PLA can be represented by a finite state automaton, and the set of languages recognized by the class of finite state automatons is a subset of the set of languages recognizd by the class of all context free grammars.

[9] Smith [16] provides a nice overview of Goodstein's Theorem in the context of Gödelian incompleteness.

[18] P. D. Summers, 'A methodology for lisp program construction from examples', *Journal of the ACM*, **24**(1), 161–175, (1977).
[19] F. Wysotzki, 'Program Synthesis by Hierarchical Planning', in *AIMSA*, pp. 3–11, (1986).

The Role of Types for Programmers

Timothy Colburn[1] and **Gary Shute**[2]

Abstract. The concept of type in computer science is intimately bound with the effort to create safe and expressive languages with which to write programs. We consider types from a programmer's point of view, focusing on how they aid a programmer's mental model of computation in a chosen domain. We also discuss how type systems offered by class-based object-oriented programming languages account for much of modern software success by facilitating code reuse.

1 Introduction and Background

Those of us who have been programming or teaching programming for over thirty years have seen the introduction of many programming languages and several programming paradigms. From a programmer's point of view, a language is useful if it is both expressive and helps avoid programming errors. The concept of a programming language *type* is indispensable on both of these counts.

Many programming errors arise from confusion about types. Computer scientists working in the area of *type theory* attempt to create formal language semantics that can be used by program compilers to automate the process of catching type errors before a program is run. (See [10] and [2] for seminal approaches to modern type theory.) Programmers benefit from this work when, for example, code is written that attempts to perform an arithmetic operation on non-numeric data. Without compile-time type checking, such code would likely produce a run-time error that would be difficult to interpret and fix. With such checking, a message can be generated that helps in identifying and quickly fixing the problem.

While type checking to catch errors is essential for a programmer, language types are just as important for the expressiveness they offer programmers in thinking about and coding the computational interactions they wish to bring about. Here again, type theory provides the foundation for languages to provide this expressiveness. As Cardelli and Wegner put it, "The objective of a [formal] language for talking about types is to allow the programmer to name those types that correspond to interesting kinds of behavior" [2, p. 490]. Our purpose in this paper is not to describe the formal languages that provide such behavior (typed lambda calculus with universal quantification, for example). Instead, we take a philosophical look at how such work has resulted in programming environments that allow programmers to think directly about the real-world entities they wish to model in their programs, and to create software that adapts and grows like the real-world environments in which they are situated. Since this paper seeks to contribute to a growing body of work in the philosophy of computer science, we offer some background on our conception of the nature of computer science in general and software in particular.

[1] University of Minnesota, Duluth, email: tcolburn@d.umn.edu
[2] University of Minnesota, Duluth, email: gshute@d.umn.edu

1.1 Computer Science as an Engineering Science

Our approach to the philosophy of computer science begins with the recognition that computer science is a science that creates its own subject matter, in the form of various abstractions of computational processes, data, and machine state. In this it differs from the natural sciences, which take natural substances and processes as their subject matter, in order to explain and predict natural phenomena.

It is plausible to argue that mathematics also creates its own subject matter, and also in the form of abstractions, but mathematics and computer science differ in the purpose served by their abstractions. While mathematics uses abstraction to create *inference structures*, computer science uses abstraction to create *interaction patterns* [4], where participants in such interactions are themselves abstract computational objects created to specify, control, and reason about underlying electronic processes with which modern human beings directly and ubiquitously interact.

Because computer science creates its own subject matter, it can be thought of as an *engineering science*, sharing with science in general the desire to understand the causal factors influencing phenomena of interest, but sharing with other engineering disciplines the goal of applying the causal knowledge gained to the creation of a desired product, most often in the form of software. But computer science stands apart from other engineering disciplines in that its subject matter consists of abstractions — algorithms, specifications, programs, data structures, etc. — which are not subject to the same physical constraints, such as gravity, temperature, the chemical properties of materials, etc., that are brought to bear on the products of other engineering disciplines.

Engineers of physical products and processes must be intimately familiar with the laws of nature and to use them to their advantage. So, for example, it is essential that a bridge engineer understand the effects of gravity on various materials and structural designs. Certainly, there are times when a computer scientist is similarly constrained by laws of nature, as when writing an operating system that must make judicious use of the limited battery lives of today's cell phones. But in the world of abstractions where a programmer normally resides, there are no laws of nature to discover and exploit. Working in such a world imposes different constraints, namely, the constraints that programmers must impose on their computational objects in order to create and control the computational processes that are the products of their work [5].

1.2 The Objects of Computer Science

In consistently referring to the "abstract computational objects" that are the subject of computer science, what do we mean? There are two senses of such a concept. First, it can refer broadly to any of the formal structures that have a role in the overall creation of software. The ontology of computational objects in this sense has been fairly well

established and unchanged since the early days of computer science and includes: algorithms, data structures, efficiency proofs, specifications, programs, correctness proofs, etc.

Second, in the sense in which we are interested here, an "abstract computational object" can refer specifically to the actual objects that take part in the computational processes that are the goal of software development projects to create. In this sense, one may (perhaps overly loosely) regard computational objects as the abstract counterparts of the electronic objects that actually carry out a computational process on a physical machine. The precise nature of this abstract/concrete relationship is open to debate, but it mirrors an essential duality of software, which has both a *medium of description* in the abstract world of algorithms and textual programs, and a *medium of execution* in the concrete world of silicon structures and their physical states [3].

1.3 How Software's Execution and Description Have Changed

Software's medium of execution, or the actual machines on which it runs, has of course changed dramatically over the decades, witnessed by the steady march of "generations" of new processors in tandem with the celebrated "Moore's Law". But the basic ontology underlying software's medium of execution is essentially, and perhaps remarkably, unchanged from the early days.

From the development of the stored-program computer in the middle of the last century until today, a computer's high-level architecture has consisted of one or more central processors, peripheral processors, primary memory, and secondary memory. And although processor and memory devices have undergone dizzying change during that time, such change is due more to miniaturization and parallelization than to fundamental changes in architecture or ontology. Modern multi-core processors, for example, implement the same basic logic gates as their predecessors from generations back.

In contrast, software's medium of description, or the languages devised for programmers to specify computational processes, has changed markedly with respect to the basic ontology of objects available to a programmer. This ontology is tied directly to the concept of a *type* in computer science. Although computer science types were initially intimately related to machine processor architecture, they have evolved dramatically throughout the history of programming, consistently increasing software's semantic expressiveness.

In section 2 we examine traditional philosophical treatments of the concept of a type, both to set the stage conceptually and to show, in sections 3 and 4, how computer science has enriched the concept from a programmer's point of view.

2 The Notion of Types in Philosophy

Discussions of types in philosophy center around two issues: (i) the type/token distinction, first drawn by Peirce, and its significance for various areas of main-stream philosophy, and (ii) the theory of types first advanced by Russell to handle various paradoxes for mathematical logic.

2.1 Types and Tokens

Peirce used the concept of a type in his theory of signs. A sign is "something which stands to somebody for something in some respect or capacity" [9, p. 99]. In particular, words are signs, but they signify in various ways. If a word is a *sinsign* — Peirce's name for a *token* — it exists as a single particular thing and has spatio-temporal qualities involving say, ink on paper or pixels on an electronic display. If a word is a *legisign*,

> It is not a single object, but a general type... Every legisign signifies through an instance of its application... Thus, the word "the" will usually occur from fifteen to twenty-five times on a page. It is in these occurrences one and the same word, the same legisign [type]. Each single instance of it is a replica. The replica is a sinsign [token]. [9, p. 102]

Loosely, the type/token distinction is a general/particular distinction, and as such it has been associated with many issues in philosophy having to do with the ontological status of abstract entities such as universals, properties, and sets. But this distinction has a unique significance for programming with respect to both (i) measuring a program's size and run-time, and (ii) understanding equality in programs.

Measuring a Program's Size and Run-Time

Consider the following procedure, written in the language C, for computing the factorial of a non-negative integer:

```
int factorial(int n) {
   int p = 1;
   while ( n > 0 ) {
      p = p * n;
      n = n - 1;
   }
   return p;
}
```

The `factorial` procedure consists of *statements*, for example "`p = p * n`" (note that "=" indicates a variable assignment, not an equality test). Before a procedure is stored in memory for execution, its statements, written in a high-level language, are translated into *machine instructions*, which are represented in a binary language of zeros and ones. A procedure's size, or amount of space it takes in computer memory, depends on the number of machine instructions a program called a *compiler* creates when it translates the procedure's statements. When counting instructions to determine a procedure's size, we count instructions in the *type* sense. For example, there would be one instruction for the statement "`p = p * n`" in the `factorial` procedure.

A procedure's run-time, however, is determined by how often its instructions are executed. In `factorial` the instruction for "`p = p * n`" is located in the context of a `while` statement, which represents a program "loop". This loop executes n times, where n is given as a procedure parameter. Each time the instruction is executed, a *token* of it is created and moved into the processor's instruction register for decoding and execution. So n tokens of the instruction exist during the procedure's execution.

It is worth noting that philosophical treatments of the type/token distinction usually place it in an abstract/concrete context. Wetzel [13, web reference], for example, asserts that "Tokens are concrete particulars; whether objects or events they have a unique spatio-temporal location". However, it may be argued that the type/token distinction for a machine instruction given above is not an abstract/concrete distinction, since machine instruction tokens are themselves abstractions of electronic state inherent in semiconductor circuitry. It may be further argued that each token of "`p = p * n`",

though unique in time, has, by virtue of its representation in the instruction register, exactly the same spatial location as any other instruction.

Understanding Equality in Programs

One of the most important aspects of computer programming involves testing computational objects for equality. Such tests are required for making decisions of all kinds, including when to terminate loops, how to search data structures, etc. So programmers must learn how to use equality operations, one variety of which is often indicated with the "==" operator. A programmer learns, for example, that if the following variables are declared and initialized in Java,

```
int a = 5;
int b = 5;
```

then the value of the expression "`a == b`" is true, whereas for the following:

```
Integer a = new Integer(5);
Integer b = new Integer(5);
```

the value of "`a == b`" is false.

The reason for this apparent inconsistency is that "`5`" is the value of a "simple" type in Java, while "`new Integer(5)`" is the value of a "class object" type. Two identical tokens for a simple value are references to the same entity, while two identical tokens for class object creation are not. While programmers are taught this distinction in class/member terms, rather than type/token terms, a similar distinction is at work.

2.2 Theory of Types

Philosophical treatments of the concept of type also have roots in the philosophy of mathematics. Although Russell was an ontological realist in some respects, he preferred *logical constructions* to *inferred entities*. Numbers, being entities that are routinely manipulated through arithmetic, were nevertheless simply inferred to exist as abstract entities, so Russell favored the reductionist attempt to define *number* in terms of another concept considered less mysterious, namely *class*. (It is acknowledged that Frege anticipated Russell in this attempt.)

The reduction proceeded by identifying a number with the class of all classes that are similar according to a suitably defined one-to-one relation, and a set of rules was put forth for transforming propositions about numbers into propositions about classes. This shifts the ontological focus from numbers to classes, and as A. J. Ayer put it:

> In the case where the reason for undertaking a reduction is that the type of entity on which it is practiced is felt to be mysterious, it would seem to be essential that the type of entity which is substituted for it should not be mysterious to the same degree. [1, p. 20]

However, classes introduced problems of their own, including, among others, how to handle infinite classes and the empty class, but Russell circumvented them by defining classes *intensionally* in terms of concepts, rather than *extensionally* by enumerating their members. For example, the class of chairs in this room is not described by listing them, but through the locution "the class of all x such that x is a chair in this room". Now the ontological focus is on *propositional functions* such as "x is a chair in this room".

While it is natural to suppose that every propositional function determines a class, this supposition led Russell to a famous paradox. Consider the class C described by "the class of all x such that x is not a member of itself". C seems reasonable and would seem to include, for example, the class of chairs since such a class is not itself a chair. However, if we ask whether C is a member of itself we get a celebrated contradiction. This paradox and others like it led Russell to his theory of types [11].

Russell proposed that a propositional function be restricted to a domain of objects that are "lower" than the function itself in a hierarchy of object types. By suitably defining the hierarchy, it becomes meaningless to ask whether C is a member of itself, and Russell's paradox is blocked.

The type hierarchy is described as follows. At the bottom of the hierarchy are individuals. The next level of the hierarchy has *first-order* propositional functions, that is, functions that take only individuals as arguments. The next level includes *second-order* functions, that is, functions that take only first-order functions as arguments, etc. With his type theory Russell created the foundations of higher-order logic, and spurred alternative type theories by Gödel, Tarski, Church, and others. The theory of types developed by these philosophers and logicians evolved into type theory as currently conceived by computer science. This theory provides strong semantic foundations for the languages used to write programs.

3 Types from a Programmer's Point of View

The types of computational objects taking part in the computational processes created by programmers are determined by the programming languages used. Programmers are interested in types for their expressiveness and how they support good programming practice. Programming language designers are concerned with types for those reasons, but also to base their languages on solid mathematical foundations.

Most programmers come to learn what a statement in a programming language means by learning its syntax and grasping various metaphors — branching, selecting, catching, throwing, and threading, to name just a few — that help to create an abstract mental model of what is occurring physically in a computer when a program runs (Colburn & Shute 2010). Those with an interest in the theory of programming languages, on the other hand, including language designers, mathematicians, and logicians, are concerned with the very meaning of computation in programming languages, or *programming language semantics*.

For theorists, the meaning of a program is given by a mathematical formalism rather than by a mental model of program execution. Such formalisms range from ones based on sets or categories to ones based on abstract machines or lambda calculus. (See [12] on operational vs. denotational semantics). In either case, program semantics for theorists is mathematical and reductivist in nature, while program semantics for work-a-day programmers involves creating mental models of program execution.

For example, what a programmer conceives as a *shopping cart*, a full-fledged program data type, may, on a denotational semantics account, be considered to be a certain meticulously constructed subset of the universe of all possible computational values — not what a programmer has in mind when thinking about and coding computational processes that manipulate shopping carts. But the expressiveness of a programmer's mental model of what he or she is coding

about is directly dependent on the sophistication of the underlying programming language semantics. In particular, if the semantics allows the introduction of new types by the programmer through type constructors, then the expressiveness of the language for the programmer is limitless. As Cardelli and Wegner put it, "These constructors allow an unbounded number of interesting types to be constructed from a finite set of primitive types" [2, p. 490]. The ability to create types that match a programmer's mental model of his or her computational domain is a hallmark of object-oriented programming (OOP).

Long before OOP, however, computer science types had their origin in the fact that different logic circuitry is required for computing arithmetic operations on integer numeric values than is required for real values (numbers that include fractional parts), and also because integer and real value representations have different binary formats. Types that are based on circuitry and binary representation are called *processor types*. Modern application programmers, however, need rarely concern themselves with processor types, focusing instead on *language types*.

Interestingly, the evolution of high-level programming languages has seen an enrichment of language types, while processor types have become relatively impoverished. We next describe how the type support offered by processors compares with the type support offered by the high-level languages that are used to control them.

3.1 Processor Types

All processors have instructions for moving uninterpreted data, that is, bunches of zeroes and ones. When we say a processor has support for a more complex type we are indicating that the processor can perform operations on the data. We place an abstraction on the zeroes and ones and come up with a *conceptual type* for which we would like processor operations that are tailor-made for it. These instructions provide a *representation*, in hardware, of our conceptual type, and the hardware provides an *interpretation* of the zeroes and ones that is in keeping with our conception. Figure 1 shows this relationship.

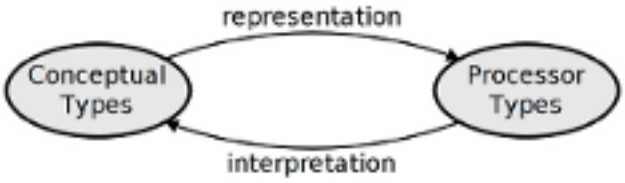

Figure 1. Conceptual Types and Processor Types

An example of a conceptual type is *integer*. Saying that a processor supports an integer type means that for two conceptual integers x and y, there are corresponding bit representations x_b and y_b such that applying the processor's "add" instruction to x_b and y_b results in a bit representation corresponding to $x + y$. Figure 2 depicts the situation where the particular conceptual type in a programmer's mental model is an integer instruction.

Processor types have evolved in the direction of simplicity in the past 35 years. Processors whose instruction sets were designed prior to circa 1990 are typically *complex* instruction set computers (CISC) that aimed to bridge the "semantic gap" between high-level languages and processors. That is, they tried to build type support into

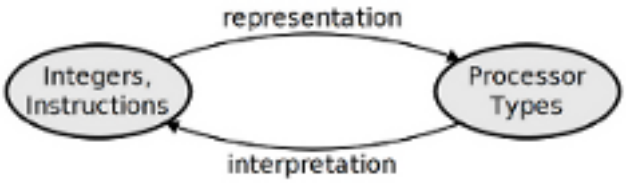

Figure 2. Integer as a Conceptual Type

processors that was closer to the type support offered by the programming languages of the time. A casual observer might find it surprising that processors whose instruction sets were designed after that time are typically *reduced* instruction set computers (RISC), with a reduced number of processor types.

RISC Processor Types

Current RISC processors support types for data, instructions, and simple functions. Data types include bit fields (various small sizes), integers (signed and unsigned, various small sizes), real numbers (single and double precision), and small chunks of uninterpreted data. Types that identify parts of instructions include registers, memory addresses, and displacements (differences between two addresses). The only support provided for functions is a special type of jump instruction that saves the current program counter in a register. This allows a function to return to where it was called from.

CISC Processor Types

Processors whose initial instruction sets were designed before circa 1990 (this includes the Intel X86 and Pentium families of processors) attempted to support a wider range of types. In addition to the types supported by current RISC processors, early processors have supported types for text strings, large chunks of uninterpreted data, large bit fields, arrays, and complex functions. Support for complex functions, for example, is illustrated by complex instructions (e.g. the VAX CALLS instruction) that performed complex manipulations on the run-time stack.

Modern processors generally have limited direct support for modern language types on which programmers have come to rely. They have even less support for three aspects of programming that are typically an integral part of a well-designed high-level language: (i) defining *equality* between computational entities, (ii) providing a *naming context* for these entities, and (iii) detecting *errors* that occur during computation.

Equality. When we conceive of a type, we are implicitly deciding what the tokens are for the type. For represented data, that requires deciding when two representations are the same. Processors, whether RISC or CISC, only have direct support for two kinds of equality test: (i) the same bit representation, and (ii) the same location in memory (equality of addresses).

Naming Context. A processor, by itself, provides little in the way of a naming context. Registers and memory locations are just named by bit patterns. Assemblers improve the situation somewhat by assigning symbols to the registers and letting programmers define symbols for memory locations. However, these names are all in one global name space. To some extent, operating systems add some

flexibility by supporting separate compilation. Individual compilation units can have their own namespace but can declare some of their symbols to be global.

Type Errors. For effective programming we need some indication of errors in our programs. Processors recognize very few types of errors, including trying to execute illegal instructions, giving invalid arguments to instructions (e.g. divide by zero), and referencing invalid memory addresses. It is easy to make type mistakes with processor instructions. An integer "add" instruction can be applied to data that is intended to represent real numbers. Such errors are typically not even detected by a processor. The only error indication is incorrect program results.

3.2 Language Types

Clearly, processor types are not adequate for today's programmers. The move from CISC to RISC has seen a corresponding increase in the reliance of high-level languages and their compilers to provide for built-in language types such as structures, higher-order functions, and objects in the technical sense of OOP.

Such language constructs provide a much better match to a programmer's conceptualization of the world than is provided by processor types. In Figure 3, processor types have been replaced by language types.

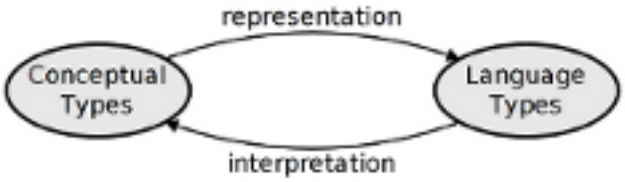

Figure 3. Conceptual Types and Language Types

3.3 Abstract Data Types

High-level language types are still often not enough to satisfy a programmer's desire for expressiveness. As every beginning computer science student learns, even high-level languages must be used to implement types the languages do not directly provide, such as lists, stacks, queues, binary trees, hash tables, etc. Such types are called *abstract data types* (ADTs) because they are supported by neither the processor nor the programming language. Instead, they must be implemented "on top of" the language with their operations (e.g. removing from a list, pushing on a stack) provided as a service by procedures or functions that hide the details of their implementation. ADTs such as those above are ubiquitous in programming and often provided by language libraries. They are so indispensable to programmers that from their point of view ADTs provided by libraries are indistinguishable from types defined by the language.

3.4 Programmer-Defined Types

But even a rich set of ADTs provided by a language library does not match the modern programmer's mental model, which might include shopping carts, characters in a game, objects in a simulation, etc. While language libraries can provide the foundational abstractions on which to build, they cannot anticipate all the types of entities programmers will think about when faced with developing software.

The key to high-level language expressiveness for programmers is therefore a mechanism allowing them to define their own types. When a language allows programmer-defined types, a programmer's conceptual types, namely the types of entities that populate both the programmer's application domain and her mental model of it, are more closely matched by the types made available by the language.

Many languages that allow the creation of programmer-defined types only allow data of those types to be *passive*, meaning they can be asked for their data but they cannot otherwise act on their own behalf. Some languages allow programmers to define their own types, but they further add a messaging mechanism that allows data items to request information and changes of each other. When data items are *active* in this sense, they are technically called "objects." In the object-oriented programming paradigm, data objects can be viewed as participating collaboratively in computation like processing nodes in a network. This gives programmers a medium of description that allows them to bring about computational processes whose objects behave in ways that are closer to how objects behave in the "real" world.

In addition to providing active data types, object-oriented languages provide better support for the three important aspects of programming previously mentioned: defining equality, providing a naming context, and detecting errors.

Equality. Most object-oriented languages provide a default conception of equality based on identity: two objects are the same if they occupy the same memory location. However, equality can be overridden to allow programmers to specify their own criteria for equality. Consider these lines of Java code:

```
Integer a = new Integer(5);
Integer b = new Integer(5);
```

Although the expression "`a == b`" is false by virtue of the fact that `a` and `b` occupy different locations in memory, the expression "`a.equals(b)`" is true because the value represented within those locations is the same.

Object-oriented programmers have the freedom to define "`equals`" in any way they wish. This freedom in effect allows them to define custom type/token distinctions; the criteria for what counts as two separate tokens (objects) of the same type can be made as strong or as weak as is called for by the application.

Naming Context. Some object-oriented languages are *strongly typed*, meaning, in part, that objects of the same type are grouped into a *class*. Such grouping has significant advantages, including compile-time error checking, more efficient executable code, and the avoidance of name conflicts.

For example, consider the following Java statement:

```
System.out.println("Hello World");
```

Here, `System` is the name of a standard library class. This class has an attached variable, `out`, which is the standard output stream for a program. This variable is an object of class `PrintStream`. The `PrintStream` class defines the `println()` operation (also called a "method"), that prints its argument followed by a new line. The names "`out`" and "`println`" are interpreted in the namespace of two different classes, allowing the classes to be defined without a tedious effort to avoid name conflicts.

Type Errors. All objects taking part in computational processes have types. In strongly typed languages, program variables that *refer* to objects must also have types. This offers opportunities for compilers to catch type errors, such as attempting to divide an integer by a text string, before the program is run. Since compiler errors are often easier to fix than run-time errors, this can be an advantage by minimizing a software project's debugging effort.

However, some object-oriented languages, such as Smalltalk, Javascript and Common Lisp Object System, are not strongly typed, permitting their variables to refer to objects of any type. In these languages, type errors are only reported at run-time. The trade-off is that development can often be quicker, since meticulously attending to type detail can be time-consuming. These languages are therefore often used for rapid prototyping and client-side scripting.

The OOP paradigm, which gained hold in the 1980s with languages like Smalltalk and C++, and flourished in the 1990s with the introduction of Java, required a radical change in how programmers think about designing code. Data structures and algorithms are still central in programmers' thinking, but instead of being controlled by functions and procedures, they emerge as objects and their methods. Objects, rather than procedures, are in control, and objects can be made to model anything: physical objects, events, relationships, systems, users, etc. Since programmers can create types for anything they can think of, they can code in essentially the same language in which they think.

4 Types and Code Reuse

The size and complexity of modern software systems preclude "starting from scratch" for each project. Ideally, software would be constructed out of components previously written in other contexts by putting them together to create an entirely new application, much like a new piece of electronics hardware can be built from modules "off the shelf." While this ideal has not been realized, software reusability is a major objective of OOP development. The "object" language construct offered by OOP lets a programmer code at a high level of abstraction. The higher a level of abstraction a programmer codes in, the more reusable that code becomes if the programmer designs those abstractions well. As we show in the rest of this section, this reusability is closely connected to types in OOP.

4.1 Reuse Through Class Inheritance

In OOP languages like C++ and Java, like objects are grouped into a *class*, which serves as the objects' type. Class terminology provides language constructs that gather together coded descriptions of the grouped objects' state and methods. As such, classes are essentially static abstractions used by programmers to describe the *instantiation* and behavior of objects at run-time. In a sense, classes are the *medium of description* for programmers to describe computational interactions, while objects populate the *medium of execution*.

Programs "stamp out" instances of classes while they run, using linguistic operators such as "**new**" followed by the name of the class they wish to instantiate. In this way, the class/object distinction for the programmer is analogous to the type/token distinction for the philosopher.

When a programmer defines a new class, a new type is introduced, and the class definition serves as the *implementation*, i.e. the internal code representation of the state and methods, of every object in the class. That implementation can be reused by any class that is made a *subclass*. Subclassing is built into many OOP languages and is represented graphically by the Unified Modeling Language (UML) [7], as shown in Figure 4. The **Child** subclass defines its own state (attributes) and methods but also *inherits* the state and methods of the **Parent** superclass. When properly used by a programmer, the subclass/superclass relationship is also an *is-a* relationship. That is, it should make sense to say that any instance of **Child** "is" an instance of **Parent**. For example, any rectangle is a polygon, but the converse is not necessarily true.

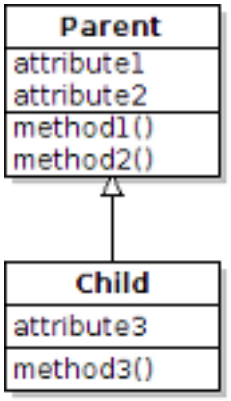

Figure 4. Class Inheritance

The subclass/superclass relationship among classes imposes a corresponding subtype/supertype relationship among the objects that are instances of those classes. OOP programmers are taught to understand class inheritance through supertype and subtype relationships among the objects they create in order to achieve code reuse. Class inheritance, built into the language of C++ and Java, lets a programmer define a new kind of object rapidly in terms of an old one, and get a new implementation almost for free, inheriting most of what he or she needs from existing classes.

This advantage must be balanced with the understanding that when programmers are defining classes for a program they are not just classifying static entities; they are defining classes and types for objects that are *dynamically changing*. Mathematically and logically it makes sense to define a subtype by imposing constraints on attributes of a parent type. For example, a square "is a" rectangle whose width and height are equal. Thus it might make sense to define a class **Square** as a subclass of **Rectangle**. There is an inheritance benefit to doing so, since, for example, **Square** could inherit methods for computing area and circumference from **Rectangle**. However, inheritance will cause type problems if **Rectangle** also has methods for changing the width and height, since these "mutator" methods allow **Square** instances to violate their constraints. Programmers must always base their type systems not just on mathematical or logical conditions; they must also take change into consideration.

Programmers must also learn to distinguish between an object's class and a program variable's type. Variables are used to *refer* to objects. Because **Parent** is a supertype of **Child**, a **Parent** variable can refer to a **Child** object, as shown here:

```
Parent child = new Child();
child.method1(); // No error
```

However, since **Parent** does not define **method3**, the code shown below does *not* work:

```
Parent child = new Child();
child.method3(); // Error: no such method
```

While this problem is avoided by typing the **child** variable as **Child**, this makes the code overly coupled to the **Child** type, as we discuss below.

The concept of a subtype may not have occurred to Peirce when conceiving the type/token distinction, but it is essential for a class-based object-oriented programmer. It may be possible to extend Peirce's language and observe that with respect to the **child** variable in the code above, the object it refers to is a *direct* token of the **Child** type and, by virtue of inheritance, an *indirect* token of the **Parent** type.

The superclass/subclass relationship and the accompanying concept of inheritance allows a programmer to create a type hierarchy that matches his or her mental model of the entities in the application domain. The hierarchy may correspond to perceived "natural kinds" in the world in the form of *domain* classes, or it may be created to present a framework for expessing *foundation* classes necessary for coding the low-level details of input/output and graphical user interfaces. In either case, the hierarchy is programmer-created, and should not be confused with the type hierarchy first proposed by Russell to dissolve his famous paradox (see section 2.2).

4.2 Abstract Classes and Types

When a programmer defines a class, each method is given a unique *signature*, including the method name, the order and types of data the method requires as arguments, and the type of data that can be communicated as a result. While a signature identifies a method, it does not provide an implementation, which is a description of the actual code that is run when the method is invoked.

Many classes are defined by giving each method both a signature and an implementation. Such classes are called *concrete*. However, language designers have found it useful to allow the definition of classes whose implementation is only partially given. In such a class, all methods have signatures, but not all methods have implementations. If an object were an instance of a partially implemented class, to invoke one of its unimplemented methods would cause an error. Therefore, partially implemented classes, though legal, may not be instantiated, and are called *abstract*.

Abstract classes are thus like types that have no tokens. This may have seemed strange to Peirce, who considered the type/token distinction with respect to words. Just to mention the concept of a unicorn, for example, is to use a "unicorn" token. Although an abstract class represents a bonafide type, there can never be an object that instantiates it. The ability to legislate such a restricted notion is in keeping with computer science's status as a discipline that creates its own subject matter.

If just one of a class's methods is not implemented, the class is abstract. However, it may have state, in the form of attributes, and usable implemented methods. Abstract classes are therefore intended to be subclassed by other classes that provide the needed implementation and then themselves become instantiated. This seems to present a puzzle with respect to types and tokens: If *some* of an abstract class's implementation is reused by a subclass, how can it be that an abstract class represents a type with no tokens? The distinction between direct and indirect tokens introduced above is helpful here. Although an abstract class represents a type with no *direct* tokens, it may have *indirect* tokens by way of inheritance.

Because an abstract class can have many subclasses, the set of things that can function as indirect tokens of an abstract type can be wide and diverse. Correspondingly, code that uses variables of abstract class types can refer to objects of wide and diverse classes, resulting in diverse and flexible behavior.

4.3 Reuse Through Object Association

While useful, class inheritance (also called implementation inheritance), whether from concrete or abstract classes, is just a mechanism for code and representation sharing. Because it is supported by the syntax of some popular OOP languages and fairly easy to understand, it is over-emphasized compared with the advantages associated with other approaches to reuse.

Consider the class inheritance hierarchy in Figure 5 that might be employed to represent various recordings. There are two problems

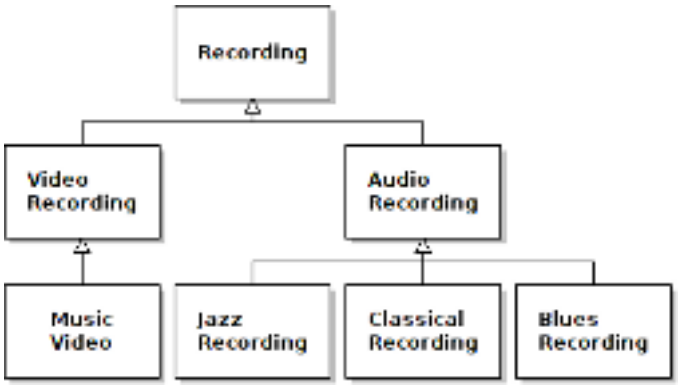

Figure 5. Misuse of Class Inheritance

with this representation. First, the classes are so similar there would be little difference in their implementations. Second, every time a new kind of recording is conceived, a new class must be created.

A better representation sees code reuse not at the class level, but at the object level. If an object representing a recording has another object representing a recording category as part of its state, the matter is simplified. Relationships between objects like this are called *associations* and represented in UML as in Figure 6. Now when a new

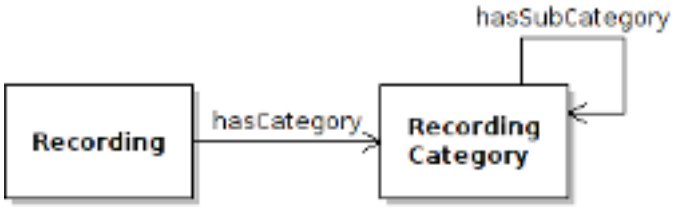

Figure 6. Object Association

kind of recording is conceived, a new **RecordingCategory** object is created with an appropriate description, but no new class needs to be defined.

Associations among objects reflect *has-a* relationships rather than the *is-a* relationships among subclasses and superclasses, but they also offer opportunities for code reuse. If an object *A* has an associated object *B* as part of its state, and if *B* has a method (or methods) that accomplish something that *A* needs to accomplish, then *A* can *delegate* the work to *B*, thereby reusing *B*'s code. To understand

how this delegation is best accomplished in class-based languages, we must understand the concept of an *interface*.

4.4 Reuse Through Interface Inheritance

We have seen how abstract classes can "widen" a variable's type by declaring methods and forcing other classes to implement them. This idea is taken to its extreme with the concept of an *interface* type. An interface is just a named list of method signatures. Loosely speaking, an interface is like a class that has no state and *none* of whose methods has an implementation. If a class is defined to handle all the method signatures listed in an interface, it is said to *implement* the interface.

Figure 7 shows two classes implementing the same interface. Neither **Child1** nor **Child2** inherits code from **Parent**. However,

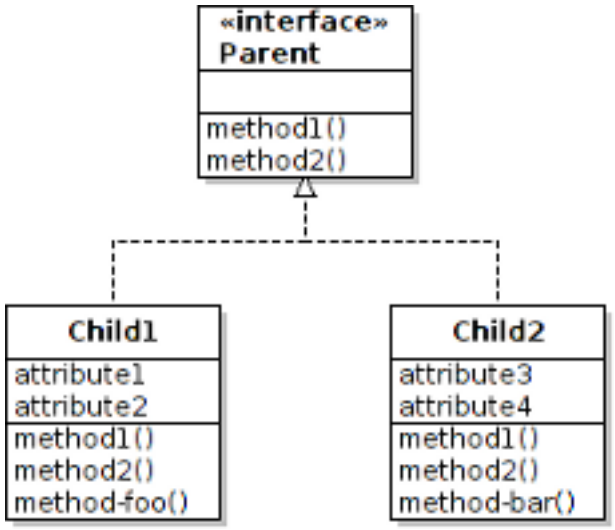

Figure 7. Interface Inheritance

code that uses interface types can more easily reuse code from other classes through *interface inheritance*. Interface inheritance makes it easier to substitute one object for another in high-level code, thereby exploiting reuse through object association. The code below shows interface inheritance at work.

```
Parent y;          // Give y interface type
y = new Child1();  // y refers to some object
y.method1();       // Get some behavior
y = new Child2();  // y refers to new object
y.method1();       // Same code, new behavior
```

Interface types, like abstract class types, can only be applied to variables, not the objects to which variables refer. So, also like abstract class types, interfaces are types without any direct tokens. A class that implements an interface type is a subtype of the interface type, so an interface type may have a wide and diverse set of indirect tokens. Because an interface offers nothing in the way of an implementation, code variables of interface types are even "wider" than those of abstract class types.

Note that if the variable **y** above does not have the interface type **Parent**, flexibility is lost as shown here:

```
Child1 y;          // Give y class type
y = new Child1();  // y refers to some object
y.method1();       // Get some behavior
y = new Child2();  // Error: type mismatch
y.method1();       // No Child2 behavior
```

4.5 Polymorphism

The ability to elicit different behavior from the same code in this way is called *polymorphism*, and it lies at the heart of much of the success of modern object-oriented programming.

The simplest use of polymorphism is to abstractly define a fixed behavior while allowing for different implementations of that behavior. In the Java standard library, for example, this is illustrated by the **Set** interface. The primary methods in this interface specify fixed behavioral expectations for adding elements to a set, removing elements, determining membership, iterating through the members of a set, and determining equality of two sets. Implementations of the **Set** interface differ primarily with regard to performance characteristics depending on the context. If the set is volatile, i.e. frequently modified, then a **TreeSet** implementation is available that uses a balanced binary tree data structure to efficiently manage adds and deletes. If the set is nonvolatile but often interated over, a **CopyOnWriteArraySet** implementation is optimized for fast traversal. If the set has members that act as values for a program's enumerated type variable, an **EnumSet** implementation is optimal

Polymorphism has, however, more interesting uses than to just adjust implementations for efficiency. In such uses there is no fixed behavior defined at the interface level, so that implementations can differ substantially from one implementation to another. In Java this is illustrated by the **Comparator** interface. A display for a table with multiple columns may have a variable of type **Comparator** to use when sorting rows. Each column can have an object that implements **Comparator** in its own way. For example, a column representing prices simply compares numbers, while a column representing items in a shopping cart compares textual descriptions of the items. When the user clicks on a table column header the comparator for that column is assigned to the variable and the table rows are re-sorted. This allows the user to dynamically determine the ordering of the rows.

Comparators are also useful for building complex ordered search structures that support *generic types*. For example, when programmers use the **TreeSet** implementation, they provide a *type parameter* indicating the type of element in the set. Thus **TreeSet**<**String**> describes a set of character strings while **TreeSet**<**Integer**> describes a set of integers, to name just two. Rather than having to separately define **TreeSet**<**String**> and **TreeSet**<**Integer**> classes, programmers developing the **TreeSet** search structure can design one class with an appropriate type parameter. Class instances are then created with a comparator argument that provides the appropriate searching behavior.

4.6 Design Patterns

While class inheritance promotes code reuse at the expense of implementation dependencies between a class and its subclasses, interface inheritance and polymorphism can all but eliminate such dependencies, resulting in two important principles of reusable object-oriented design: (i) *Program to an interface, not to an implementation* [8, p. 18], and (ii) *Favor object association over class inheritance* [8, p. 20].

Interface types are of central importance to class-based object-oriented programmers, because by separating method signatures from their implementation, they *decouple* code that uses interface typed variables from code that implements the methods those variables use. We have argued elsewhere [6] that decoupling pervades both computer science as a discipline and the wider context of computing at large. For OOP, decoupling means eliminating dependencies between objects that result in changes to one object requiring changes in another. So if a programmer who doesn't use interface types wants to use **Child2**'s **method1** in the code at the end of section 4.4, another variable of type **Child2** must be introduced, complicating the code. The problem is that the code is *too tightly coupled* to the **Child1** class.

Object-oriented programmers have developed a design discipline that emphasizes object decoupling by using *design patterns* (Gamma et al. 1995) that focus on interface types. For example, the error-free code in section 4.4, as simple as it may look, uses a design pattern called *Strategy* [8, p. 315] to allow the code's behavior to be easily changed. Like many design patterns, Strategy employs object association and interface typing to accomplish its objective, which is to delegate the execution of a strategy in some context to another object. Such delegation allows the replacement of that object by another that implements the same interface, in order to get different (possibly radically different) behavior. The Strategy design pattern is represented in UML as shown in Figure 8. Note that the UML model shown in

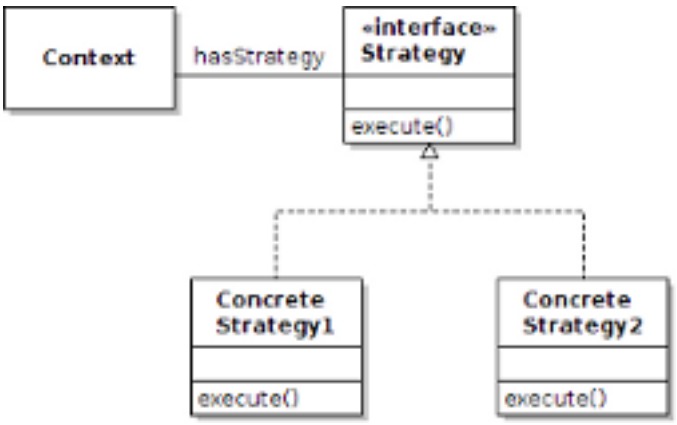

Figure 8. The Strategy Design Pattern

Figure 7 is an instantiation of the general pattern shown in Figure 8.

4.7 Roles As Types

In the "real" world, humans often make classifications not of individual entities, but of *roles* that such entities play in complex interactions. For example, national constitutions and laws define complex protocols governing the interactions between various legislative, executive, and judicial agents. In democratic countries, at least, we expect these roles to be designed polymorphically. That is, within broad limits they do not restrict people acting in these roles to certain types of behavior such as liberal or conservative; they only govern how they interact to define new laws, enforce them, and interpret them.

We may even metaphorically describe the governmental positions as **Strategy** references. When citizens elect new individuals to serve in these positions through the election process, it is analogous to variable **y** changing its reference from an object of class **Child1** to an object of class **Child2** in the code in section 4.4. By electing new officials, citizens can implement different political behavior.

Experienced object-oriented programmers consider roles to be types just as much as they consider individual entities to be types. For example, when designing software for keeping track of information about people in a complex educational organization, a programmer could define a **Person** class with several subclasses such as **Student**, **Faculty**, and **Administrator**. This design treats people as tokens in a hierarchy of types. While this might work in a simple program, it would require significant changes when the software grew to deal with more complex needs, including situations where people change roles, or have two or more roles at the same time. A better design is to create a type system where the tokens are abstractions (roles) instead of concretes (people).

To emphasize the importance of roles in programming, design patterns are often described in terms of *participants*. For example, the *Observer* design pattern [8, p. 293], shown in Figure 9, defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically. Two of the essential participants in Observer are a **Subject**,

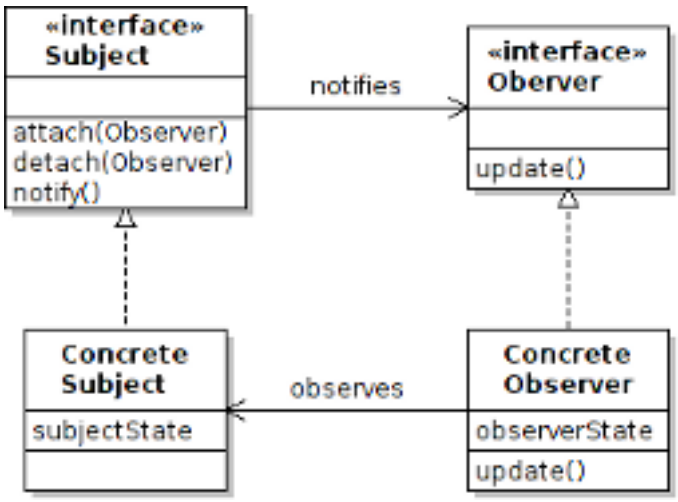

Figure 9. The Observer Design Pattern

which provides a mechanism for registering and unregistering observers, and an **Observer**, which provides a mechanism for responding to change. In complex software these participants are not types of entities. Instead they are types for roles in a certain kind of interaction between objects, and as such they are *abstract* participants in the pattern. Defining interfaces for these participants allows diverse objects with different *external* responsibilities, that is, responsibilities not inherent to the Observer design pattern, to play the roles in the pattern. These objects are *concrete* participants in the pattern. The **Concrete Subject** adds methods that observers use to determine the nature of the observed change, while the **Concrete Observer** uses those methods to respond to the change. When used in this way, the interface types **Subject** and **Observer** are therefore doubly abstract: (i) they type variables, not objects, and (ii) they type roles, not entities fulfilling those roles.

This is not surprising given that computer science creates its own subject matter in the form of abstractions. But the type discipline offered by high-level languages, in particular object-oriented ones, accounts for the linguistic richness that allows programmers to code

with the objects of their thought.

REFERENCES

[1] A.J. Ayer, *Russell and Moore: the analytical heritage*, Harvard, Cambridge, MA, 1971.

[2] L. Cardelli and P. Wegner, 'On understanding types, data abstraction, and polymorphism', *ACM Computing Surveys*, **17:4**, 471–522, (1985).

[3] T. Colburn, 'Software, abstraction, and ontology', *The Monist*, **82:1**, 3–19, (1999).

[4] T. Colburn and G. Shute, 'Abstraction in computer science', *Minds and Machines: Journal for Artificial Intelligence, Philosophy, and Cognitive Science*, **17:2**, 169–184, (2007).

[5] T. Colburn and G. Shute, 'Abstraction, law, and freedom in computer science', *Metaphilosophy*, **41:3**, 345–364, (2010).

[6] T. Colburn and G. Shute, 'Decoupling as a fundamental value of computer science', *Minds and Machines: Journal for Artificial Intelligence, Philosophy, and Cognitive Science*, **21**, 241–259, (2011).

[7] M. Fowler and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[9] C.S. Peirce, 'Logic as semiotic: the theory of signs', in *Philosophical Writings of Peirce*, ed., J. Buchler, 98–119, Dover, New York, (1955).

[10] J. Reynolds, 'Types, abstraction and parametric polymorphism', *Information Processing*, **83**, 513–523, (1983).

[11] B. Russell, 'Mathematical logic as based on the theory of types', in *Logic and knowledge: essays 1901–1950*, ed., R. Marsh, 57–102, Capricorn, New York, (1971).

[12] R. Turner and A. Eden. The philosophy of computer science. The Stanford Encyclopedia of Philosophy. http://plato.stanford.edu/archives/win2011/entries/computer-science/.

[13] L. Wetzel. Types and tokens. The Stanford Encyclopedia of Philosophy. http://plato.stanford.edu/archives/win2011/entries/types-tokens/.

A Compilation of Dutch Computing Styles, 1950s – 1960s[1]

Edgar G. Daylight[2]

Abstract. Post-war researchers in applied mathematics experienced a technical shift from the specific to the general: from the down-to-earth *engineering* activity of instructing a specific finite machine for a particular problem to a *science* of programming an abstract computer for a family of problems. This technical shift was enriched by local styles and habits of research. Understanding the different dispositions towards this shift is a rich topic for the history and philosophy of programming, a topic which we address here by comparing and contrasting the work of the following Dutch historical actors: Van der Poel (Delft), Van Wijngaarden (Amsterdam), Blaauw (Twente), and Dijkstra (Eindhoven).

Post-war researchers in applied mathematics experienced a technical shift from solving *specific* calculational problems to reasoning in terms of *general* principles. For example, during the 1950s and 1960s, special purpose machines in industry were gradually replaced by general-purpose computers (e.g. [5, p.5–7][10]). Similarly, engineering concerns about the *finite* limitations of a machine waned when the metaphor of language took hold of the computing field at large [9, 24]. A high-level programming language allowed the programmer to solve a *general* mathematical problem on a *variety* of computing machines. In short, the history of computing can be viewed as an emerging dichotomy between specialization and generalization. Specialization stands for the down-to-earth *engineering* activity of instructing a specific finite machine for a particular problem. Generalization refers to a *science* of programming an abstract computer for a family of problems.

Just like on the international scene, computing in the Netherlands was enriched by its local styles and habits of research. Dutch actors of international standing were, among others, Van der Poel (Delft), Van Wijngaarden (Amsterdam), Blaauw (Twente), and Dijkstra (Eindhoven). Each actor led his own distinct school of thought but shared the international ambition of creating a safe institutional harbor for their emerging profession: first for applied mathematics, later for computer science.

Dutch researchers in applied mathematics, also called numerical analysts, were *jointly* breaking away from the pure sciences which had dominated Dutch academia for so long [1]. This common goal was reflected in *each* local style. Indeed, **the pureness in traditional mathematics characterized Dutch research in computing**. The institutional struggle paralleled the aforementioned technical shift; in fact, they are two sides of the same coin. All Dutch actors, in search for institutional recognition, stressed and defended their own distinct notion of "generality" (alternatively: "simplicity" or "elegance"). As a result, the diversity in local styles overshadowed their common goal, leading to conflict among each other and even causing some to leave the country.

Among the various Dutch cities, Delft and Amsterdam stood out during the 1940s. The former was a center of excellence in engineering. All the aforementioned researchers with the notable exception of Dijkstra were educated as engineers in Delft. After the second world war, Amsterdam became the Dutch center for "applied mathematics". Van Wijngaarden was the man in charge. All aforementioned researchers passed through Amsterdam during the 1950s.[3]

Besides numerical analysis, Amsterdam was already an internationally renowned center in the well-established discipline of mathematical logic. It is no coincidence that in later years, during the 1960s, Van Wijngaarden and his student De Bakker would pick up some specific technical ideas from logic and influence prominent international researchers in computing.

During the 1950s, knowledge of mathematical logic *in connection with* computing was scarce in both the Netherlands and abroad. An exception was Van der Poel in Delft. He applied Turing's theoretical 1936 notion of universal machine in his practical 1952 design of the `ZERO` computer [27]. Van der Poel was one of the first in the Netherlands to make the transition from the "specific" to the "general". He first built what he later called a "pre-Von-Neumann" machine, the `TESTUDO` (cf. [29]). From the early 1950s onwards he built several "Von-Neumann" machines in the sense that both data and instructions were placed in the same memory. These machines were quite successful, especially in comparison with some of Amsterdam's first computers. The reason lies in Van der Poel's insistence to specialize his programs for the machine and the mathematical problem at hand and *not* to follow Van Wijngaarden's linguistic ideals to the extreme. Indeed, Van der Poel was and remained first and foremost an engineer. He did not abstract away from the finiteness of the machine. On the contrary, he viewed Turing's 1936 paper [26] and logic in general in a finite setting, as he demonstrated in his 1956 dissertation *The Logical Principles of Some Simple Computers* [28].

Van Wijngaarden and Dijkstra were reasoning linguistically during the 1950s in Amsterdam; that is, *from* a language *to* a machine, as opposed to the common approach of reasoning about a programming language in terms of its implementation on a real, finite machine[4]. Dijkstra moreover relied on his education in theoretical physics from Leiden. Just like a table is made up of molecules which in turn are made up of atoms, software too can be viewed in such an hierarchical manner. His analogy with physics[5] may help us understand why

[1] This paper is a concise summary of a Dutch chapter in a forthcoming book entitled *De geest van de computer*. The author also plans to publish English derivatives of this work in the near future.

[2] Freelance post-doc researcher at Eindhoven University of Technology, The Netherlands, email: egdaylight@dijkstrascry.com

[3] See [1][3][10]. Even Van der Poel, who did not officially join the Mathematical Center in Amsterdam, was affiliated with Van Wijngaarden as his PhD student [28].

[4] Van Wijngaarden and Dijkstra's linguistic reasoning and their notion of generality are described in [9][12].

[5] See Dijkstra [15]. Also the comments made by Van der Poel and De Bruijn in interviews confirm this: "Dijkstra was neither a mathematician nor an

he viewed the problem of translating ALGOL as an hierarchical problem. He introduced an intermediate machine-independent level in the Dijkstra-Zonneveld compiler, a level which is a precursor to the now widely used virtual machine [9]. Dijkstra continued with his layered software approach during the 1960s; his THE operating system [17] and his later *Notes on Structured Programming* [18] had such a great impact that he was awarded the Turing award in 1972.

Dijkstra became professor in numerical analysis in Eindhoven in the fall of 1962 [3]. By that time he had already distanced himself from Van Wijngaarden's ideology [11]. He would continue to do so, even openly at conferences as in Vienna 1964 [25, p.21], ending with a climax in 1968 when he wrote a Minority Report (cf. [23]) to express his misgivings about ALGOL68.

Van Wijngaarden stayed in Amsterdam and continued his linguistic line of research. His sense of generality, perceived as unorthodox by many, led him to introduce innovative programming constructs which are still used today. Moreover, Van Wijngaarden started to borrow concepts from logic as early as 1962 [30, p.11]. His student De Bakker would go much further in this direction [13, p.21], reading the literature of Davis [7] and, hence, of Turing, and also the work of Markov, Rabin, and Scott. Unlike Van der Poel, De Bakker applied these ideas from logic in connection with the language metaphor[6].

Blaauw brought a lot of know-how with him when he joined Van Wijngaarden's group in 1952. Having worked with Aiken in Harvard, he was able to show the Amsterdamers Loopstra and Scholten how to build reliable computers that actually work [2]. In 1955, he left the Netherlands again, this time to join IBM in New York. There he experienced the struggling transition from the "specific" to the "general" in the commercial setting of IBM machines.

Blaauw was involved with both the IBM STRETCH computer and the later IBM\360 system. The "general purpose" character of these computers stood in sharp contrast to the many special-purpose stored-program machines[7] that IBM had been building during the 1950s and early 1960s [5, 10]. Instead of building a special machine for data processing and another one for scientific computing, Blaauw and his close colleagues started to see the all-purpose character of their stored-program computers. This realization was reflected in the names of their computers: the word "stretch" alluded to stretching over different application domains, and "360" referred to 360 degrees; that is, to a machine that could handle *all* application domains. It should be emphasized here that, just like the majority of IBM employees, Blaauw was an engineer, not a theoretician who was aware of earlier developments in logic. Translated from his Dutch words:

> That was really a new idea, that you have a modest machine that can execute exactly the same instructions as the big machine; the only difference is that the big machine runs faster and has more memories. So that was a key insight, that a bigger machine need not be designed differently. [10]

Blaauw became famous at IBM during the early 1960s when he introduced his three levels of concern: architecture, implementation, realization (cf. [4]). He showed the commercial world how to systematically build general-purpose computers, in line with *some* of the abstractions that had already been put on paper by Dijkstra in 1953 [16].

engineer, he was a physicist" (paraphrased) [8, 11].

[6] As an interesting aside, Hoare's influential 1969 paper on axiom-based semantics [21] and logic programming [6] both have an untold history that starts with Van Wijngaarden's and subsequently De Bakker's distinct views on how language and machine relate to each other.

[7] A machine can be both a special purpose machine *and* a stored program machine.

In 1965, Blaauw started an academic career at the recently founded Technische Hogeschool Twente. He brought his newly acquired notion of generality with him, generality in the form of IBM's APL language. APL is an abbreviation for "*A* Programming Language" [22]; Blaauw advocated using only *one general* language (APL) to design *both* the software and hardware of a computer. In Twente, Blaauw taught his students how to design the IBM way. Twente's computing style, as advocated by Blaauw and Duijvestijn, was one of *prototyping*: first build a prototype of the system (by programming in APL), experiment with that prototype and only afterward start implementing the real system.

Twente's emphasis on prototyping stood in sharp contrast to what Dijkstra was delivering in Eindhoven during the 1960s. In fact, already during his earlier years in Amsterdam, Dijkstra wanted to prove a program correct by mathematical reasoning alone [12, Section 3.2.2]. Testing, as advocated in Twente, was a last resort to Dijkstra. Moreover, one of the technical reasons why Dijkstra distanced himself from Van Wijngaarden's linguistic constructions was because he could not convince himself that they were mathematically correct [25, p.21]. The main incentive for Dijkstra to design his THE operating system in a layered manner was because it aided him in proving its correctness [17].

During the 1960s, Van Wijngaarden and especially De Bakker linked mathematical logic to programming languages [13][14][31][30]. Dijkstra, by contrast, would only become convinced of logic's role during the early 1970s [20, p.346]. And even then he would remain cautious, not wanting to blindly follow the logical tradition which was starting to dominate much of computing research (cf. [19]). The coming man in Dutch academia was Van Wijngaarden's successor, De Bakker. His appeal for logic was heard in many Dutch cities, including Eindhoven. Dijkstra left the Netherlands in 1984 to continue his research at the University of Texas at Austin.

REFERENCES

[1] G. Alberts, *Jaren van berekening*, Ph.D. dissertation, Universiteit van Amsterdam, 1998.

[2] G. Alberts and H.T. de Beer, 'De AERA. Gedroomde machines en de praktijk van het rekenwerk aan het Mathematisch Centrum te Amsterdam', *Studium*, **2**, 101–127, (2008).

[3] K.R. Apt, 'Edsger Wybe Dijkstra (1930–2002): A Portrait of a Genius', *Formal Aspects of Computing*, **14**, 92–98, (2002).

[4] G.A. Blaauw and Jr. F.P. Brooks, *Computer Architecture: Concepts and Evolution, Part I Chapters 1–8*, Addison-Wesley, 1997.

[5] *Planning a Computer System: Project STRETCH*, ed., W. Buchholz, McGraw Hill, 1962.

[6] A. Colmerauer and P. Roussel, 'The Birth of Prolog', in *HOPL Preprints*, pp. 37–52, (1993).

[7] M. Davis, *Computability and Unsolvability*, McGraw-Hill, 1958.

[8] E.G. Daylight, 'Interview with Van der Poel in February 2010, conducted by Gerard Alberts, David Nofre, Karel Van Oudheusden, and Jelske Schaap', Technical report, (2010).

[9] E.G. Daylight, 'Dijkstra's rallying cry for generalization: the advent of the recursive procedure, late 1950s – early 1960s', *The Computer Journal*, (March 2011). doi: 10.1093/comjnl/bxr002.

[10] E.G. Daylight, 'Interview with Blaauw on 29 November 2011, conducted by Gerard Alberts and Karel Van Oudheusden', Technical report, (2011).

[11] E.G. Daylight, 'Interview with De Bruijn on 11 November 2011, conducted by Gerard Alberts and Karel Van Oudheusden', Technical report, (2011).

[12] E.G. Daylight, *The Dawn of Software Engineering: from Turing to Dijkstra*, Lonely Scholar, 2012. www.lonelyscholar.com, ISBN 9789491386022.

[13] J.W. de Bakker, 'Formal Definition of Algorithmic Languages', Technical report, Mathematische Centrum Amsterdam, (1965). MR 74.

[14] J.W. de Bakker, 'Axiomatics of simple assignment statements', Technical report, Mathematisch Centrum Amsterdam, (1968). MR 94.
[15] E.W. Dijkstra, 'EWD 317: On a methodology of design', Technical report.
[16] E.W. Dijkstra, 'Functionele beschrijving van de ARRA', Technical report, Mathematisch Centrum Amsterdam, (1953). MR 12.
[17] E.W. Dijkstra, 'The structure of the 'THE'-multiprogramming system', *Communications of the ACM*, **5**, 341–346, (1968).
[18] E.W. Dijkstra, 'Notes on structured programming', Technical Report T.H.-Report 70-WSK-03, Technische Hogeschool Eindhoven, (April 1970). Second edition.
[19] E.W. Dijkstra, 'EWD 1227: A somewhat open letter to David Gries', Technical report, University of Texas at Austin, (1995–1996).
[20] E.W. Dijkstra, 'EWD 1308: What led to "Notes On Structured Programming"', Technical report, Nuenen, (June 2001).
[21] C.A.R. Hoare, 'An axiomatic basis for computer programming', *Communications of the ACM*, **12**(10), 576–580, (1969).
[22] K.E. Iverson, *A Programming Language*, John Wiley and Sons, Inc., 1962.
[23] C.H. Lindsey, 'A history of ALGOL 68', *HOPL-II The second ACM SIGPLAN conference on History of programming languages*, **28**(3), (March 1993).
[24] D. Nofre, M. Priestley, and G. Alberts, 'When technology becomes language: New perspectives on the emergence of programming languages, 1950-1960', *Presented at SHOT 2011 and in preparation for publication*, (2011).
[25] T.B. Steel, ed. *IFIP Working Conference on Formal Language Description Languages*, Amsterdam, 1966. North-Holland.
[26] A.M. Turing, 'On computable numbers, with an application to the Entscheidungsproblem', *Proceedings of the London Mathematical Society, 2nd series*, **42**, 230–265, (1936).
[27] W.L. van der Poel, 'A simple electronic digital computer', *Appl. sci. Res.*, **2**, 367–399, (1952).
[28] W.L. van der Poel, *The Logical Principles of Some Simple Computers*, Ph.D. dissertation, Universiteit van Amsterdam, February 1956.
[29] W.L. van der Poel. Een leven met computers. TU Delft, October 1988.
[30] A. van Wijngaarden, 'Switching and Programming', Technical report, Mathematisch Centrum Amsterdam, (1962). Report MR 50.
[31] A. van Wijngaarden, 'Numerical analysis as an independent science', *BIT*, **6**, 66–81, (1966).

Anatoly Kitov and ALGEM algorithmic language

Vladimir V. Kitov[1] and **Valery V. Shilov**[2] and **Sergey A. Silantiev** [3]

Except several publications (see, for example [4]), many achievements of Soviet programmers for the period from 1950 till 1980 practically are still remained unknown abroad. The reasons for this are several secrecy of some works, many open papers were published in Russian language and thus were unavailable for foreign scientists etc. But these achievements were very considerable. It is sufficient only to mention such original developments as REFAL metalanguage for formal language text processing (V. F. Turchin, middle of 1960) or El'-76 high level language (V. M. Pentkovsky, middle of 1970) which was Assembler language as well.

One of the scientists who made significant contribution to the theory and practice of algorithmic languages development was Anatoly Kitov (1920-2005) – outstanding Russian scientist in the field of informatics and computing (Fig. 1) . Another famous Russian scientist, IEEE Computer Society Computer Pioneer academician Alexey Lyapunov called Anatoly Kitov the first knight of Soviet cybernetics. This was not accidentally. Anatoly Kitov was the real pioneer and the words the first and for the first time can be applied to all stages of his scientific career. A. Kitov was the author of the first in the USSR positive article about cybernetics which was not recognized by official Soviet communist ideology. He had published the first Ph. D. thesis on programming, the first Soviet book about computers and programming, the first articles on non-arithmetic utilization of computers. He was the author of the first project of wide-national computer network, the first national textbook on computer science, the first scientific report on management information systems (MIS), etc. He designed the most powerful Soviet computer of that time, established the first scientific computer Centre (the so called Computer Centre nr 1 of the USSR Ministry of Defense), developed the associative programming theory, created the standard industrial management information system (for the Ministry of Radio Engineering Industry) etc. The total amount and innovative quality of his scientific works are really impressive. Unfortunately due to some political reasons his research activity was not officially recognized in Soviet Union [6].

Figure 1. Anatoly Kitov, c1965

In the second half of nineteen fifties Anatoly Kitov for the first time formulated proposals for complex automation of information processing and state administrative management on the base of Integrated computer centre state network (ICCSN). On January 7, 1959 A. Kitov sent in the Central Committee of the Communist Party of the Soviet Union the letter about the necessity of National economy automated management system on the base of ICCSN. It was first in the world proposal on designing of national state automated system for economics management. The leadership of USSR partly adopted Kitov's project but the main idea about the structural reconstruction of National economy management system was rejected [7].

That is why in Autumn1959 Anatoly Kitov sent the second letter in the Central Committee in which he proposed the new innovative project which advanced the modern times on several dozen years. It was the project named Red Book establishing of integrated computer centre state network of dual designation (for economics management and defense control). But once more due to the political reasons this project was rejected and, moreover, its author was excluded from the Communist Party, dismissed from his job at Computer Centre and later discharged from the Soviet Army.

At the beginning of nineteen sixties A. Kitov was the head and scientific supervisor of the group of programmers who were developing large program complex for real time military computer used in air defense system. At this time he was working at Scientific Research Institute nr 5 of the Ministry of Defense. Material which determined the specifications for the future programming language had been got by A. Kitov from his practical experience during the realization of above mentioned project. However he had begun development of ALGEM (ALGorithms for Economy and Mathematics) language some years later, when after his dismissal from the army he worked at Main Computer Centre of the State Radio-Electronic Committee. ALGEM was designated for the programming of the economical, mathematical, logical and control (including the real time control of the technical systems) tasks. In particular the extremely important was the aim of this work to design the language for the programs for processing the large (super large at that time) information arrays of complex but determined structure. In 1965 the first version of ALGEM was finally developed. The expertise of ALGEM was fulfilled in some Soviet scientific institutions and in particular at Computer Centre of Siberia Branch of USSR Academy of Science (facsimile of the letter from the USSR State Committee for Scientific Research Coordination to Siberia Computer Centre with the request of expertise is shown on Fig. 2).

In 1967 A. Kitov published the monograph The programming of informational logical tasks in which for the first time ALGEM lan-

[1] Institute of History of Natural Sciences and Technics Russian Academy of Science, email: vladimir.kitov@mail.ru
[2] MATI – Russian State Technology University, email: shilov@mati.ru
[3] MATI – Russian State Technology University, email: intdep@mati.ru

Figure 2. The letter of USSR State Committee for Scientific Research Coordination.

guage was described [1]. This language was realized in the system of computer programming ALGEM ST-3 (ST-3 – Syntax-directed Translator, the 3rd version) described in 1970 in monograph [5]. Besides ALGEM the system included translator and standard subroutines library. The system ALGEM ST-3 was realized on the base of the second generation computer "Minsk-22" (Fig. 3).

Figure 3. Soviet computer Minsk-22

The development of this computer was ended in 1964. For that time it was middle class computer with the efficiency of 5-6 thousand operations per second, ferrite core main memory with the 8196 cell capacity of 37 bit each and external memory on magnetic tape with the 1.6 million cells. Computer was produced by series from 1965 till 1970. Minsk-22 was one of the most mass computers for that time and altogether 953 machines were manufactured. It was installed at hundreds of computer centres in various Soviet ministries and later in some socialist countries. Appropriately ALGEM ST-3 included in Minsk-22 software was also had several hundred of installations and was widely used during the development of various applied systems which were designated for the processing of hospital charts, application forms, results of experiments etc.

According to the concept of the author ALGEM must have been the procedure-oriented programming language. That is why Algol-60 was selected as a base for the new language. But the practical orientation of ALGEM determined the deep modification of the basic language and introduction of serious alterations: new block nesting mechanism, new variable types and also the special advanced instruments for the work with the values densely packed in computer memory cells.

Program structure is the same as in Algol: declarations followed by operators in order of their execution. Variables are localized in the block: *begin end*. The program can be only the block. Four variable types are determined in ALGEM: *integer, real, Boolean, string*. Values of numerical variables may be numbers, values of logical variables logic values and of string variables strings in contrast to Algol.

In ALGEM there is a possibility to describe the structure of variables by means of declarator shape (in Russian *vid*). It points the quantity and type of symbols contained in variable value:

integer P *shap*e 9 (5) means that integer variable P has the length of five decimal digits;
integer P *shape* 7 (3) P has the length of three octal digits;
integer P *shape* 1 (8) P has the length of eight binary digits;
string date *shape* 99 – L(10) – 9(4) means that string variable *date* has the following structure two decimal digits, space, up to 10 letters, space, four decimal digits (for example 21 September 2012).

In declaration of real variables you may point the sign of number and exponent, location of decimal point and possibility of rounding.

These possibilities were introduces in the language because of necessity for programming the air defense tasks which demanded the highest possible compact value package in memory cells.

Variables in ALGEM language may be simple and composite (composite variables have not *shape* tag). Composite variable contains other variables including composite as well and in fact it is a record type.

Composite arrays are also provided in the language. They are formed from composite variables of similar structure. Declaration of composite value is always ended by symbol *level*. Thus, there is the opportunity to work with arrays of records (i. e. tables). All these means make it possible for ALGEM to solve specific economical and management tasks of any level (they are also attributable to the systems of real time, in particular to air defense systems).

There are also compound operators in the language except blocks. It provides unlimited nesting of blocks and composite operators as well. The operators are the same as in Algol: assignment operator, go to operator, conditional jump, loop operator. Instead of procedure operator it was introduced procedure-code operator (that is why there is no declaration of procedures among list of declarations).

Not consider further details of the differences between two languages we only want to mention that ALGEM mainly was designed for practical purposes of industrial programming while Algol is a classical language for algorithms presenting.

ALGEM also included the best instruments of the functional programming languages Lisp and IPL-V but it was the essential extension of these languages by addition of new list structures and procedures for their processing. For example, Lisp functions provide the processing of two adjacent elements of linear and chain lists. Kitov introduced generalized list structures node lists and nested lists. For the declaration of list variables it was introduced list declarator. Together with the *level* symbol it forms the pair of parentheses. List declarations begins with *list* symbol followed by declaration of list header and then by list element structure. The list element may be the list itself.

One more A. Kitov's monograph "The programming of economical and management tasks" [2] was issued in 1970. Next year this book was translated in German language [3] (covers of Kitov's four monographs about ALGEM language and it implementation are

shown on Fig. 4). New version of the language ALGEM-2 was presented in this book. This language was further development of ALGEM and was also oriented on the solving of economical and management tasks in information retrieval systems.

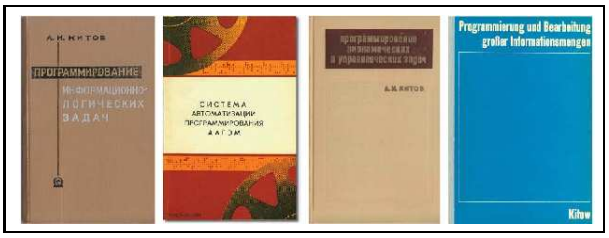

Figure 4. Anatoly Kitov's books on ALGEM languag

ALGEM algorithmic language gained wide popularity and many programs were written on it. For example, information retrieval system "Setka-5" designed for searching the documents upon requests was programmed on this language. This system was realized on Minsk-22 computer and could process document arrays up to 120,000 items stored on 16 magnetic tapes. Search time in the array of 10,000 items (i. e. on one magnetic tape) was 4-5 minutes.

In whole it may be said that ALGEM language was very successful product oriented on industrial programming. It combined the best features of procedure-oriented languages (Algol-60), languages for complex data structures processing (COBOL) and list-processing languages (Lisp, IPL-V). ALGEM realized principles of associative programming developed by A. Kitov. In result ALGEM became one of the most popular programming languages in USSR in 1970-s.

Later A. Kitov developed and introduced one more original programming language NORMIN designed for solving the tasks in medicine sphere but this is the theme of the separate investigation.

Today one can say that programming has not only osmotically infused scientific and artistic research alike, but also that those new contexts elucidate what it may mean to be an algorithm. This talk will focus on the 'impatient practices of experimental programming, which can never wait till the end, and for which it is essential that the modification of the program in some way integrates with its unfolding in time. A contemporary example is *live coding*, which performs programming (usually of sound and visuals) as a form of improvisation.

Early in the history of computer languages, there was already a need felt for reprogramming processes at runtime. Nevertheless, this idea was of limited influence, maybe because, with increasing computational power, the fascination with interactive *programs* eclipsed the desire for interactive *programming*. This may not be an accidental omission, its reasons may also lie in a rather fundamental difficulty, on which we will focus here.

In itself, the situation is almost obvious: not every part of the program-as-description has an equivalent in the program-as-process. Despite each computational process having a dynamic nature, an integration of programming into the program itself must in principle remain incomplete. As a result, a programmer is forced to oscillate between mutually exclusive perspectives. Arguably, this oscillation reveals a specific internal contradiction within algorithms, a necessary obstacle to semantic transparency. By calling this obstacle *algorithmic complementarity*, I intend to open it up for a discussion in a broader conceptual context, linking it with corresponding ideas from philosophy and physics.

Here a few words about this terminology. Complementarity has been an influential idea in conceptualising the relation between the object of investigation, as opposed to the epistemic apparatus and the history of practice. Originating in the psychology of William James, where it referred to a subjective split of mutually exclusive observations, Niels Bohr used it to denote the existence of incommensurable observables of a quantum system (position vs. momentum, time vs. energy). Independent of the particular answer Bohr gave, complementarity raises the question of whether such a coexistence is induced by the experimental system or already present in the subject matter observed. Accordingly, in the case of programs, we may ask whether this obstacle is essential to their nature or whether it is a mere artefact of a specific formalisation. Algorithms, arguably situated between technical method and mathematical object, make an interesting candidate for a reconsideration of this discourse.

The existence of an obstacle to semantic transparency within algorithms and their respective programs need not mean a relative impoverishment of computation. Conversely, prediction is the wager and vital tension in every experimental system, as well as in interactive programming. After the conceptual discussion, I will try to exemplify this claim by introducing a few examples in the recent history of *live coding*. Again and again surfacing in form of symptoms such as an impossibility of immediacy, I hope this practice will be conceivable in terms of having algorithmic complementarity as one of its driving forces.

REFERENCES

[1] Kitov A. I. programmirovanie informacionno-logicheskih zadach. ?oskva: Sovetskoye radio (programming for information-logical problems, in russian), 1967.

[2] Kitov A. I. programmirovanie ekonomicheskih i upravlencheskih zadach (programming for economic and management problems, in russian), 1971.

[3] Kitov A., *I. Programmierung und Bearbeitung Grosser Informationsmengen*, B. G. Teubner Verlagsgesellschaft, 1972.

[4] Shura-Bura M Ershov A., *The Early Development of Programming in the USSR. In A History of Computing in the Twentieth Century*, Academic Press, 1980.

[5] Borodulina N. G. et al., *Sistema avtomatizacii programmirovaniya ALGEM (ALGEM: automation of programming system, in Russian)*, Statistika, 1970.

[6] Shilov V. V. Kitov V. A., *Anatoly Kitov ? Pioneer of Russian Informatics*, History of computing. Learning from the past, Springer, 2010.

[7] Shilov V. V. Kuteinikov A. V., 'Asu dlya sssr: pis?mo a. i. kitova n. s. khrushchevu, 1959 g. (automated management systems for the soviet union: A 1959 letter from a. i. kitov to n. s. khrushchev, in russian)', *Problems of the history of science and technology*, (3), 45–52, (2011).

Algo*rhythmic* Listening 1949-1962 Auditory Practices of Early Mainframe Computing

Miyazaki Shintaro [1]

Abstract. It is still very unknown that besides the first visual interfaces to early computers, such as the Williams-Kilburn Tube operating for the first time in 1948 on the Manchester Small-Scale Experimental Machine (SSEM) or the many type-machine like printing outputs of standard calculating machines, there were as well *auditory interfaces*, which were build in as simple amplifier-loudspeaker set-ups in to the circuits of the early mainframe computers. Examples of such machines were the famous UNIVAC-I, the TX-0 at MIT, the CSIRAC in Australia and the Pilot ACE in England, but as well later machines such as the Pegasus produced by Ferranti Ltd. in Manchester and the PASCAL-Computer of Philips Electronics in Eindhoven, Netherlands.

This paper is one of the first accounts of these auditory practices of early mainframe computing. The term algo*rhythm* as an intermingling of algorithm with rhythm shall be proposed as an alternative conceptual approach to decode the history of computing and programming. Finally this paper will reflect on some epistemological implications of such a historical practice for understanding some aspects of current computational cultures.

1 ALGO*RHYTHMICS* AS METHOD

In an emerging discipline called *software studies* algorithms are defined as abstract structures, which "bear a crucial, if problematic, relationship to material reality [10, 16]." Rhythm on the otherhand can be defined as an elementary movement of matter, which oscillates in-between the discrete and the continuous, hence between the symbolic and the real, between digital and analog. Thus algo*rhythms* are generated by an inter-play, orchestration and synthesis of abstract organisational, calculational respectively algorithmic concepts with rhythmic real-world signals, which have measurable physical properties.

Although rhythm is not a technical term in the jargon of computational sciences, it denotes a semantic area, which forms the opposite of terms such as clock, meter, measure or pulse. A clock seems to be precise, but a rhythm is not. In music theory, rhythms are usually defined in relation to meter, which provides a more symbolic framework within a rhythm is acting. This concept of a symbolic and ideal structure, which provides a frame work for real and fluctuating physical effects, such as sound or vibration is analogous to that of John von Neumann, when he described the characteristics of digital signals. Digital machines work correct

> as long as the operation of each component produces only fluctuations within its preassigned tolerance limits [...] [19, 294]

The rhythm of a piece of music is correct as long as it works within the preassigned symbolic and ideal tolerance limits of human perception. This is the case as well with digital machines and it legitimizes the usage of the term rhythm for describing computational processes.

An exception of the non-use of rhythm as a term in the technical jargon of computing seems to be the british community of early mainframe computing. Frederic C. Williams and Tom Kilburn used the term rhythm extensively in a paper, which presented the Manchester Mark I [20]. This tradition was continued as well in the technical jargon used in the manual of the Ferranti Pegasus Computers [1]. Nevertheless, the usage of the term pattern had for some reason a greater impact and it was used world wide, but when it comes to the task of indicating both the abstract and material – or to be precise signal-based – aspects of computing and programming, then the amalgamation of algorithm with rhythm into algo*rhythm* might be the best description as it will be shown in the following.

2 AMPLIFIER-LOUDSPEAKER SETUPS

In 1990 at the so called UNIVAC-Conference organized by the *Charles Babbage Institute* of the University of Minnesota, Louis D. Wilson, one of the main engineers of the BINAC and the UNIVAC-I remembers how his method of technical listening evolved around the year 1949 as a practical procedure to get some idea of the computation and operational processing happening in the circuits of the mainframe computers:

> When we were testing BINAC out, we were working two shifts and we worked all night long, and we had a radio going. After a while we noticed that you could recognize the pattern of what was happening at the moment by listening to the static on the radio. So I installed a detector in the console and an amplifier and a speaker so that you could deliberately listen to these things [4, 72].

Auscultating and tapping into the circuits of early mainframe computers was very common around the world and the methods of simply attaching an amplifier-loudspeaker system to relevant data channels evolved at different places independently.[2] These set-ups were used for specific listening methods for real time monitoring of computational processes or as sonic indicators for hardware malfunctions.

The UNIVAC-I released in 1951, was one of the first commercially produced mainframe computers. First customers were the *United States Census Bureau*, the *US Air Force* or the *Columbia Broadcasting System* (CBS). The 1958 *Univac-1 Maintenance Manual* indicates:

[1] Berlin, Germany, email: miyazaki.shintaro@gmail.com

[2] It is still possible to do this kind of tapping with current digital gadgets such as laptops, netbooks and iPads by using portable AM-radio players.

> [T]he contents on the highspeed bus are constantly detected and supplied as an audio signal [3, 22-Ch.2].

In the case of the many Pegasus computers produced by Ferranti Ltd. in Manchester from the late 1950s, on the source of the auditory display (see [13, 24]) was variable. It was called a „flying noise probe" as described in [2, Fig. 17.1] and [1, 175]. Christopher P. Burton (*1932), who worked as an engineer of the Ferranti Pegasus Computers and is known for rebuilding the First Manchester Computer (Manchester Small Scale Experimental Machine) in 1998 [6] remembers:

> [O]n Pegasus, the test programs are permanently stored on an isolated part of the magnetic drum, so a few simple manipulations of the control switches could call whatever test program was currently of interest. A working program had it's characteristic sound (by chance, depending where the noise probe was connected) and the sound just changed when a fault was detected [7].

Individual flip-flops in different registers, different data bus nodes or other passages of data traffic could become sources for auditory monitoring. Not only a passive listening of processes of computation was very common, but as well an active exploration of the machine, while listening to its rhythms.

> Very common were contact failures where a plug-in package connector to the 'backplane' had a poor connection. These would be detected by running the 'test program' set to continue running despite failures (not always possible, of course), and then listening to the rhythmic sound while going round the machine tapping the hardware with the fingers or with a tool, waiting for the vibration to cause the fault and thus change the rhythm of the 'tune' [7].

Algo*rhythmic* listening served as an auxiliary method to get information about the inner workings of a mainframe computer. One might assume that it was put into practice as an indirect solution to the calmness of the machines operating on vacuum tubes in comparison to earlier machines, which operated with sounding and clicking relay switches, but this is still to be verified.

3 RECONSTRUCTING A HISTORICAL ALGO*RHYTHM*

Audio recordings of algo*rhythmic* listening practices are rare, but not impossible to find. One such recording was made by engineers of the PASCAL-Computer by Philips in Eindhoven in the early 1960s (see [14] and [15]). By analyzing the audio material and by a close-reading of the scientific explanations written by the PASCAL-engineers the following in-depth analysis would explain how the sounds were created and how they relate to the computation processes and the specific algorithm, which was responsible for the sounds, thus it will specify the reason to call such an listening method an algo*rhythmic* practice.

The area within the electronic circuit of the PASCAL, where some key operations of the algorithm were transformed into audible sounds, noises, rhythms and melodies was the last flip-flop of shift-register S. The electronic signals at this passage got amplified by a simple loudspeaker-amplifier set-up and transformed into mechanical movements of the loudspeaker's membrane, which produced sound.

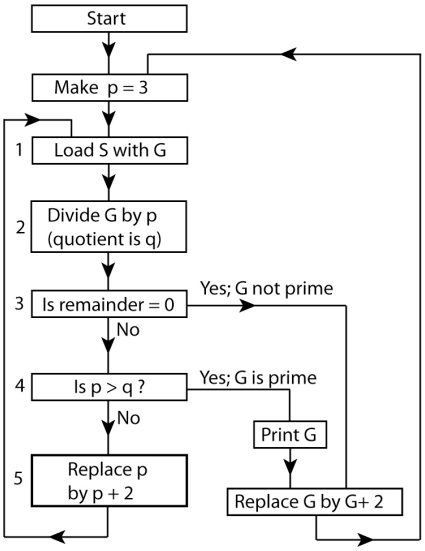

Figure 1. Flow chart of algorithm used in the search for prime number as represented in [15, Fig. 3.].

It is known that calculating whether an integer is a prime numbers or not has always been a benchmark in the history of computing. This was as well the case in the context of the PASCAL-Computer and therefore it had a demo-programm implemented that should show how fast it is able to calculate prime numbers. As you can see on fig. 1 finding out if a number is a prime number or not, is merely a matter of division, comparison, counting, shifting and adding. A concrete example will exemplify the algorithm. The number to find out if it is a prime number shall be 443. As defined by the algorithm the divisor p is 3. The value 443, which is stored in G gets transferred to shift-register S. This is done by step 1. Step 2 is performing the division. The first quotient q would then be 147. 3 times 147 is 441. 443 minus 441 is 2, which is the remainder. In step 3 we check, if the remainder is zero and go to step 4 if no. Here we check if p, which is 3, is bigger than q, which is 147. The answer is no, so we go on to step 5, add the number 2 to p and go back to step 1. P is now 5 and after the division the quotient q is 88. In the following we would loop step 1 to 5 altogether additional 9 times, until the divisor p gets bigger than the quotient q and until we could prove that 443 is a prime number. The series of quotients created is stored in shift-register S. In the present case it would consists of the numbers 88, 63, 49, 40, 34, 29, 26, 23, 21, 19. If you look only at whether the numbers are odd or even you can see that there is some rhythm of change between the two properties, such as E, O, O, E, E, O, E, O, O, O, while E stands for even and O for odd.

Fig. 2 represents the electronic signal and changes of potential, which is measurable at the last flip-flop of shift-register S during one iteration of the above described algorithm for finding out if a number is a prime number or not. A LOW or almost no signal represents a symbolic 0 in shift-register S and at the same time it means that there is an even number stored in it. This is represented by the waveform *a* at the top of the diagram. And as you can see from waveform *b* at the bottom, if you have a 1, then you have a HIGH, which means an odd number. This number is the quotient q and thus an impor-

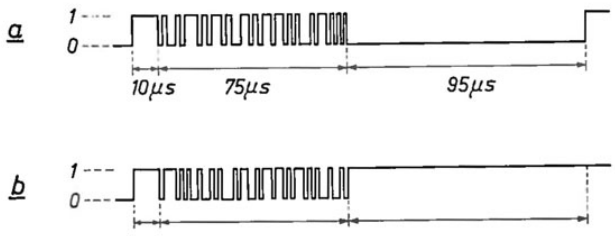

Fig. 4. Voltage variations on the loudspeaker during the thickly drawn cycle in fig. 3.
a) Quotient q even, *b*) q odd.

Figure 2. Electronic signals as represented in [15, Fig. 4.].

tant intermediate result of Step 2 as shown on fig 1. As mentioned earlier, during the computation process the algorithm will produce a variety of different rhythms concerning the pattern of even and odd numbers, thus it will produce different series of waveform *a* and *b* in many variations, while the basic cycle of 180 μs is kept. Therefore a 5.55 kHz tone is audible through-out the whole calculation process. At the same time a sonic process, which could be described as a howling is audible. At the beginning of each calculation the howling is too fast to be audible. It produces some white noise, but then gets continuously slower, until you can hear the up and down of the frequency very clearly. This happens especially with very big numbers, because the bigger the number, the longer the series of quotients and the more regular the rhythm of changes between odd and even numbers [15, 174]. The howling is created by a process called frequency modulation, which can be understand as the coupling of two oscillators. In this manner many different frequencies can be produced. And since the change of rhythms in the alternation of odd and even-quotients happens stepwise, the algorithm produces the specific kind of howling, which is audible on the vinyl recording from 1962 made by the engineers of Philips.

4 POPULARIZATION OF ALGORITHMIC THINKING

Before reflecting the disappearance of the described listening practices, the notion of algorithm itself shall be shortly inquired.

An algorithm is the combination of logic and control as it was defined in the 1970s [12], furthermore, an algorithm formulated in a programming language is not the same as a list of algebraic formulas. Mathematicians of the 20th c. like Alonzo Church (1903–1995) or Stephen C. Kleene (1909–1994) already used the term algorithm in the 1930s and 40s, whereas Alan Turing (1912–54) or Kurt Goedel (1906–1978) used it scarcely. With the dawn of higher level programming languages such as Algol 58, Algol 60 and all the other Algol inspired languages in the early 1960s the notion of algorithmic notation spread through out the academic world of scientific computation. Algol is the abbreviation for ALGOrithmic Language. It was the result of a pan-atlantic collaboration between many different programming pioneers and mathematicians such as Friedrich L. Bauer (*1924), Hermann Bottenbruch, Heinz Rutishauser (1918–1970) and Klaus Samelson (1918–1980), who were members of the European group and John W. Backus (1924–2007), Charles Katz, Alan J. Perlis (1922–1990) and Joseph H. Wegstein from the US. For creating the Algol 60 language later on John McCarthy (1927–2011) and Peter Naur (*1928) joined the team.

In the proposal from the North American side Algol 58 should have been named International Algebraic Language, which distinctly marked the lack of conceptual clarity concerning the differences between the meaning of algorithmic and algebraic in the mindset of the Americans. The Europeans must have protested vehemently against the idea of calling their new language algebraic, since the term algorithm was better established in the emerging community of scientific calculation. Already 1955 there was a Pan-European conference in Darmstadt, Germany, where pioneers such as Herman H. Goldstine (1913–2004), Andrew D. Booth (1918–2009), Howard H. Aiken (1900–1973), Konrad Zuse (1910–1995), Friedrich L. Bauer (*1924) and Edsger W. Dijkstra (1930–2002) presented their work. It was at this conference, where Rutishauser presented for the first time his concept of an algorithmic notation system to a broader audience. Thereby he clearly distinguished it from algebraic notation systems.

> The equal sign is substituted by the so called results-in sign $\Rightarrow$, which demands, that the left values shall be calculated as declared into a new value and shall be named as declared on the right side (in contrast to $a + b = c$, which in sense of the algorithmic notation would merely be a statement) [16, 28].

Donald E. Knuth (*1938) himself a pioneer in algorithmic thinking of a slightly younger generation and as well a software historian wrote:

> [M]athematicians had never used such an operator before; in fact, the systematic use of assignments constitutes a distinct break between computer-science thinking and mathematical thinking [11, 206].

It can be argued that this important shift in the mode of understanding was induced by the machinic reality of computers and their boundedness to time based processes. To convince the Americans by the theoretical importance of this distinction must have been one of the aims of the European fraction. But this change seemed to be quite difficult. John Backus didn't change his vocabulary even after the first meeting of 1958 in Zurich, Switzerland. In a 1959 document called *The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference* [5] he still wrote algebraic and never used algorithmic. Only the fast dissemination of Algol 60 especially in the emerging computer science departments in Europe helped to establish the new way of algorithmic thinking.

5 DISAPPEARANCE OF THE LOUDSPEAKERS

The method of listening to algorithms disappeared around the same time in the early 1960s, when the term algorithm started to get established as a proper term of the then emerging computer sciences. In this sense, nobody got the idea to speak about algo*rhythmic* listening, since it went forgotten very soon and the intermingling of algorithm with audible rhythm was not possible any more.

In fact, the decay of algo*rhythmic* listening as an auditory engineering practice was closely related to the emergence of software in general. The amplifier-loudspeaker setups disappeared because the up to that time mostly human operators were replaced by software based operating systems. John W. Tukey (1915–2000) wrote in 1958:

> Today the 'software' comprising the carefully planned interpretive routines, compilers, and other aspects of automative programming are at least as important to the modern electronic

calculator as its 'hardware' of tubes, transistors, wires, tapes and the like [18, 2].

With the emergence of the *compatible time-sharing system (CTSS)* developed at MIT in the early 1960s and other operating systems, the mainframe computers could listen to themselves. The routine of error detection and process monitoring, which was previously already automatized, but partly still done by ear, got entirely implemented into the functionality of the operating system.

> The supervisor program remains in A-core at all times when CTSS is in operation. Its functions include: [...] [M]onitoring of all input and output from the disk, as well as input and output performed by the background system; and performing the general role of monitor for all foreground jobs [8, 8].

Altough error detection and diagnostics was a feature, which was already in use in earlier machines such as the relay computers in the Bell Laboratories [17, 39], it gained a new momentum with operating systems such as the CTSS. At the end of the decade and the success of higher-level programming languages, nobody was listening to the rhythm, noise and melodies of the data signals, but reading the signs and alphanumerical symbols on their screens. To put it provocatively: computerlinguistics won over computerrhythmics.

> The digital computer is a symbolic machine that computes syntactical language and processes alphanumerical symbols; it treats all data [...] as textual, that is, as chunks of coded symbols. [9, 171].

Programming was always an abstract way of thinking. The better the interfacing between human and machine language was working, the easier the programmers could concentrate on the core matter and forget about the physics, signals and algo*rhythmics* of computing. In fact the black-boxing of the operator was one of the key improvements, which leaded to the dawn of software cultures.

6 CONCLUSIONS

Despite the disappearance of algo*rhythmics* as an engineering practice this paper has shown that it is a legimate, alternative method to decode historical aspects of computing and programming. Similar micro-analytical case studies could be done with the history of phreaking or the history of the term ping and its usage in network debugging.

As it has been briefly shown, algo*rhythmics* as a method of historical and critical inquiry, which takes the notion of a program as a logico-mathematical-physical structure seriously and emphasizes both the signal-based and machinic aspects of programming and the importance of abstract, logical and mathematical concepts, is able to create new de- and reconstructions of the history of programming. Furthermore it could be applied as a method, which is not only a theory, but has practical implications for the area of data asthetics and information aestheticization. For doing that not only visual, but as well sonic aspects must be considered.

ACKNOWLEDGEMENTS

Many thanks to Gerard Alberts, Christopher P. Burton, Peter Samson, Wolfgang Ernst, Michael Takezo Chinen and Martin Howse.

REFERENCES

[1] *Ferranti Pegasus Computer, Vol. 2, Logical Design*, Ferranti Ltd., Hollinwood, 1956.

[2] *Ferranti Pegasus Computer, Vol. 2a, Logical Design (Diagrams)*, Ferranti Ltd., Hollinwood, 1956.

[3] *Univac-1 Maintenance Manual. For Use With Univac I Central Computer*, Remington Rand Univac, New York, 1958.

[4] Univac conference, oh 200. oral history. 17-18 may. Charles Babbage Institute, University of Minnesota, Minneapolis, 1990.

[5] John W. Backus, 'The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference', in *Proceedings of the International Conference on Information Processing*, pp. 125–132. UNESCO, (1959).

[6] Christopher P. Burton, 'Rebuilding the first manchester computer', in *The First Computers – History and Architectures*, eds., Raul Rojas and Ulf Hashagen, 379–386, MIT Press, Cambridge, MA, (2000).

[7] Christopher P. Burton. Interview and email-communication with shintaro miyazaki. Berlin, Jan.-Feb. 2011.

[8] Fernando J. Corbató, M. M. Daggett, and R. C. Daley, *The Compatible Time"=Sharing System. A Programmer's Guide*, MIT Press, Cambridge, MA, 1963.

[9] Florian Cramer, 'Language', in *Software Studies – a Lexicon*, ed., Matthew Fuller, 168–174, MIT Press, Cambridge, MA, (2008).

[10] Andrew Goffey, 'Algorithm', in *Software Studies – a Lexicon*, ed., Matthew Fuller, 15–20, MIT Press, Cambridge, MA, (2008).

[11] Donald E. Knuth and Luis T. Pardo, 'The early development of programming languages', in *A History of Computing in the Twentieth Century. A collection of essays with introductory essay and indexes*, eds., N. Metropolis, J. Howlett, and Gian-Carlo Rota, 197–273, Academic Press, New York, (1980).

[12] Robert Kowalski, 'Algorithm = logic + control', *Communication of the ACM*, **22**(7), 424–436, (July 1979).

[13] Gregory Kramer, 'An introduction to auditory display', in *Auditory Display. Sonification, Audification and Auditory Interfaces*, ed., Gregory Kramer, 1–77, Westview Press, Reading, MA, (1994).

[14] W. Nijenhuis, 'The pascal, a fast digital electronic computer for the philips computing center', *Philips Technical Review*, **23**(1), 1–18, (1961).

[15] W. Nijenhuis, 'Listening in to the pascal', *Philips Technical Review*, **24**(4/5), 164–170, (1962).

[16] Heinz Rutishauser, 'Massnahmen zur vereinfachung des programmierens. (bericht über die 5-jährige programmierarbeit mit der z4 gewonnenen erfahrungen)', in *Elektronische Rechenmaschinen und Informationsverarbeitung (Nachrichtentechnische Fachberichte, hrsg. von J. Wosnik, Bd. 4)*, eds., A. Walther and W. Hoffmann, 26–30, Friedrich Vieweg und Sohn, Braunschweig, (1956).

[17] George R. Stibitz and Evelyn Loveday, 'The relay computers at bell labs (part one and two)', in *Datamation (Part 1, April and Part 2, May)*, pp. 35–44, 45–49, (1967).

[18] John W. Tukey, 'The teaching of concrete mathematics', *The American Mathematical Monthly*, **65**(1), 1–9, (Jan. 1958).

[19] John von Neumann, 'General and logical theory of automata', in *Collected Works, Vol. 5: Design of Computers, Theory of Automata and Numerical Analysis*, ed., A. H. Taub, 288–328, Pergamon Press, Oxford, (1963).

[20] Frederic C. Williams and Tom Kilburn, 'The university of manchester computing machine', in *Proceedings of the Review of Electronic Digital Computers (International Workshop on Managing Requirements Knowledge)*, pp. 57–61, Los Alamitos, CA, (1951). IEEE Computer Society.

Logic and Computing in France: A Late Convergence

Pierre Mounier-Kuhn[1]

How did mathematical logic interact with computing in the pioneer era? Was the "Turing machine", to sum up a common model, a decisive source of inspiration for electronic computers designers? I tend to consider this model as a founding myth of theoretical computer science, an *a posteriori* reconstruction, more than an accurate historical account. Based on archival research and oral history interviews, a detailed investigation on the case of France – a mid-size country where computers appeared years later than in Britain and in the USA – suggests rather a late *encounter* than a *filiation* process.[2]

Of course we must distinguish between several branches of mathematical logic. Boolean algebra was taught and used as soon as the first digital calculators were developed in French companies and laboratories, in 1950. It went differently with the theories of computability and recursive functions which had "revolutionized" mathematical logic in the 1930s, but were ignored in France until the mid-1950s, and did not seem to interact with computing until the early 1960s.

Computing in the 1950s emerged in a few French universities as an ancillary technique of applied mathematics, mainly numerical analysis, to answer the needs of electrical engineering, fluid mechanics and aeronautics [16]. In the science faculties of Grenoble and Toulouse, then of Lille and Nancy, at the CNRS' Institut Blaise Pascal in Paris, small teams of applied mathematicians and electronic engineers struggled to get unreliable vacuum tube calculators to process algorithms written in binary code or assembler: Their concerns were very remote from the high abstractions of mathematical logic.

Besides logic had been eclipsed from the French mathematical scene since Jacques Herbrand's premature death in 1931. Moreover, it was banned from mathematics by the Bourbaki group, for reasons which will be explained in the paper, and rejected toward philosophy [14].

In the "desert of French logic", to quote a contemporary, only a couple, Jean-Louis Destouches and Paulette Février, worked on the logical foundations of physics [9, 1]. Février also published translations of foreign logicians, in a collection of books she directed at a Paris publishing house, and organized a series of international conferences: *Applications scientifiques de la logique mathématique (1952) [7], Les Méthodes formelles en axiomatique (logique mathématique), Le Raisonnement en Mathématiques* (1955), etc.

Only two mentions of the Turing machine appeared in France in the first half of the 1950s. In January 1951, at the CNRS international conference on calculating, a delegate from the British National Physical Laboratory, F. M. Colebrook, introduced his presentation of the ACE computer by mentioning Turing's paper of 1936 – "a most abstract study which was in no regard a preview of modern automatic digital calculators", yet attracted the interest of the NPL director [4]. This mention arose no visible echo in the 600 pages of the conference proceedings, nor in the memory of the participants. More important perhaps, at the end of the same year, the Bourbaki seminar invited German-French-Israeli logician Dov Tamari to speak about "*Machines logiques et problèmes de mots*" [19].[3] Tamari described the Turing machine, remarking that the term was misleading, it was essentially logical schemes representing a simplified "ideal calculating man". It belonged to pure mathematics and offered a new perspective on algorithms. Yet Tamari noticed that Turing's theory might have a "possible application in the field of calculating machines". In short, these two glimpses of the Turing machine were very far from presenting it assertively as the model for modern computers.

The mid-1950s: A revival of mathematical logic

Mathematical logic started a revival in 1955, when Bourbakist Henri Cartan invited the Austrian-American Georg Kreisel to teach in Paris. Simultaneously, three French doctoral students – two mathematicians, Daniel Lacombe and Jean Porte, and a philosopher, Louis Nolin – dared to embrace this marginal matter. Let us introduce two of them to have a glimpse at their trajectories.

Daniel Lacombe (Ecole Normale Supérieure 1945) initially studied number theory and other mathematical themes well established in the French school. In 1955 he started to publish brief texts on recursivity [11], likely under the influence of Kreisel with whom he co-signed two papers. In 1960 he presented a complete overview on "La théorie des fonctions récursives et ses applications" (75 pages), reviewing Gödel's, Church's and Herbrand's theorems, Turing's machine, Kleene's works, etc. The only French author he quoted was Jean Porte, which seems to confirm that there was no other. The introduction stressed that the theory of recursive functions was "*à la base de la majorité des résultats intéressants obtenus en Logique mathématique au cours des trente dernières années*", in other words a paradigm in this branch of mathematics. This considerable article also mentioned briefly that this theory was useful for the formal representation of electronic calculators, which in turn stimulated reflexions on the old, intuitive concept of calculation. Lacombe was not seeking to "sell" this theory to computer specialists, however the fact that he exposed it in the *Bulletin de la Société Mathématique* de France allowed to touch numerical analysts as well as pure mathematicians [12].

Jean Porte studied logic within philosophy, in which he graduated in 1941. Then he took mathematics while participating in the underground resistance in the Toulouse region. In 1949 he joined the French statistics institute (INSEE) where he invented the *catégories socio-professionnelles*, for the 1954 census – an essentially empirical work. Meanwhile Porte began research in mathematical logic and presented a paper on modal logic at a 1955 conference on *Reasoning in Mathematics* [18]. This conference marked a renaissance

[1] CNRS & Université Paris-Sorbonne Associate Researcher, Centre Alexandre Koyré-CRHST, email: mounier@msh-paris.fr.

[2] This research stems out of questions addressed in my book [15].

[3] Dov Tamari (1911-2006), born Bernhard Teitler, had prepared his doctorate in Paris in the 1930s.

of mathematical logic in France, particularly as the French admitted that logic problems could be expressed in algebraic form and that mathematicians were interested. In 1956 Porte proposed "A simplification of Turing's theory" at the first international Cybernetic conference in Namur (Belgium) [8]. This paper reveals that at least one Frenchman had read the major works by Church, Curry, Gödel, Kleene, Post, Robinson, Rosenblum and Turing on computability, lambda-calculus and recursive functions theory. It is also interesting as Porte was adressing a *Cybernétique* audience, which still included specialists of computers (who would soon keep cybernetics at bay as a set of vague speculations). Yet Porte's conclusion mentioned no practical implication, even indirectly, of these theories, which might concern them. On the contrary he suggested to "reach an even higher level of abstraction than Turing's machines". If he talked to cyberneticians, it was from the balcony of the logicians' ivory tower.
In 1958 he received a CNRS researcher position, at the Institut Blaise Pascal in Paris, where another philosopher turned logician, Louis Nolin, had just been appointed to manage the computer pool. Porte and Nolin soon began writing programs for the Elliott 402 and IBM 650 computers of the institute. This was the first recorded interaction of logicians with electronic computers in France. Yet we don't have clues about the relationship they possibly established between their research in logic and their practice as programmers. Even if they did, they remained exceptions for several years. Computer experts were struggling with vacuum tube circuit and magnetic drum problems, or focused on developing numerical analysis, so that computability theories made little sense to them.

The 1960s: A convergence with computer science

Things changed in the early 1960s, when a series of events manifested a convergence between logic and the nascent computer science.
In October 1961, IBM's European education center at Blaricum (Netherlands) hosted a meeting on the *Relationship Between Nonnumerical Programming and the Theory of Formal Systems*.[4] Several French computer scientists and logicians participated, including a co-organizer of the meeting, Paul Braffort. Among the speakers, they heard Noam Chomsky and Marcel-Paul Schtzenberger lecture on "The algebraic theory of context-free languages", and John McCarthy present his vigorous manifesto, "A Basis for a Mathematical Theory of Computation", which proclaimed the foundation of a new *science of computation* based on numerical analysis, recursive function theory and automata theory.
In June 1962, a mathematics conference hold at the science faculty of Clermont-Ferrand included sessions on computing and on logic, the latter being represented by a constellation of international stars – Tarski, Beth, Bernays, Rabin, etc. In his keynote address, Renè de Possel, the head of the Paris computing institute, Institut Blaise Pascal, explained that mathematical logic, hitherto a field of pure speculation, had become useful to mathematics in general and to information processing in particular.[5] De Possel stressed that Von Neumann, "the first promoter of electronic computers", was also a logician; and that, at a humbler level, programmers proved more efficient when they knew some logic – "to my great astonishment", De Possel confessed. With Von Neumann, Turing and others emerged a general theory of machines, which interests computer designers as well as users. It appears in several new application fields. While attempts to make machines reason are still embryonic, actual work on machine translation, automatic documentation, artificial languages and their compilation, reveal problems resorting to mathematical logic and linguistics. "To the point that special courses in logic should be created for this purpose".
At the second IFIP congress (Munich, August 1962), a session was devoted to "Progress in the logical foundations of information processing" – a topic not adressed at the first IFIP congress in Paris (1959). John McCarthy hammered again the gospel he was preaching at Blaricum a year before; and an engineer from Siemens, Heinz Gumin, explained why computer designers needed mathematical logic [10]. Among the French delegation (nearly 10 of the audience), at least a few listeners got the message.
Actually the message was already being spread in the French computing community through its learned society AFCAL. In late 1961 at the AFCAL seminar on symbolic languages, Louis Nolin, who had attended the Blaricum meeting, gave a programmatic lecture. He recommanded to design computer languages according to the axiomatic method established in mathematics – Algol being exemplary of this approach. In order to build an algorithm, it was useful to determine first if the function was effectively computable. For this, computer scientists would be well advised to learn about the solutions elaborated 30 years ago by logicians.[6]
Louis Nolin had become De Possel's assistant and chief programmer at Institut Blaise Pascal, thus he was in a good position to translate words into action. In the autumn of 1962, regular courses on the theories of computability and recursive functions were introduced in the computer science curriculum in Paris at graduate level. A seminar was organized by J.-L. Destouches, assisted by Jean Porte, Daniel Lacombe and a third logician, Roland Fraïssé. The same started teaching "logic for programmers" at graduate level. Meanwhile, Paulette Février published a translation of A. Grzegorczyk's classic treaty on recursive functions, and created within the Institut Blaise Pascal a collection of brochures explicitely titled "Logic for the calculator's use": Reprints of journal articles, seminar and course texts, doctoral dissertations in logic, were thus made available beyond the tiny circle of French logicians.

From 1963, logic was firmly established in the computer science curriculum at the University of Paris' Institut de Programmation. Beside its intellectual interest for programmers that Nolin and others were outlining, the adoption of logic had an institutional motivation: Computing teachers needed to set up course programs with more formal matters than Fortran training or the physical description of machines, and logic responded perfectly to this quest. Three years later, the Ministry of National Education defined a new, nationwide masters diploma, Maîtrise d'informatique, including a certificate of "Algebra, mathematical logic, compiler and system theory" [13].
Other universities followed progressively. Grenoble was practically in phase with Paris, although at a smaller scale, as logic was taught by an astronomer turned linguist, Bernard Vauquois, who headed a laboratory for machine translation and was an early member of the Algol committee. The cross-fertilization between various scientific fields in the mid-1960s in Grenoble is well exemplified by the prehistory of the Prolog language, as told by one of its participants[3]: The syn-

[4] The proceedings were published two years later with an even more explicit title [2]

[5] The conference was divided in four sessions covering Pascal's domains: Logic, Numerical analysis and automatic computing, Probabilities, Differential geometry and mathematical physics. The proceedings were published in Annales de la Faculté des sciences de l'Université de Clermont, Série Mathématiques, 1962, vol.7-8.

[6] The paper [17] gave a few major references: Gödel's definition (1934), its use by Kleene (1952), Martin Davis' treaty on Computability and Insolvability (1958), and Daniel Lacombe's [12].

ergy between two projects – Algol compiling and natural language processing – led young researchers to absorb a wealth of recent international publications on syntax analysis, W-grammars, graph theory, recursive functions and lambda-calculus. This boiling exploration of new avenues geared itself with another soaring movement, artificial intelligence and automatic demonstration, and later led to Prolog and to a novel conception of algorithmics, directly based on mathematical logic [5].

Later came Nancy, Clermont and other faculties where computing science remained firmly rooted in mathematics.

Boosted by this interaction with an expanding new discipline, mathematical logic flourished again in French universities at the end of the decade. Reciprocally, the alliance between logicians and computer practitioners was a decisive factor in the assertion of computing as a new science.

Was the case of France exceptional, or representative of a general process? Evidently the case of France may differ from the case of more advanced countries in the post-war period, and we must attempt a comparison.[7] Still it offers a striking counter-model.

REFERENCES

[1] M. Bitbol, 'Jean-louis Destouches, théories de la prévision et individualité', *Philosophia Scientiae*, **5**(1), 1–30, (2001).

[2] P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, North-Holland, Amsterdam, 1963.

[3] Jacques Cohen, 'A view of the origins and development of Prolog', *Communications of the ACM*, **31**(1), 26–36, (1988).

[4] F. M. Colebrook, 'Le modèle pilote du calculateur automatique électronique arithmétique (ACE) du NPL', in *Les Machines à calculer et la pensée humaine*, Paris, (1953). Éditions du CNRS.

[5] Alain Colmerauer. Du traitement de la langue naturelle à Prolog. Ecole nationale supérieure d'informatique et de mathématiques appliquées, Grenoble, 10 Mars 2011.

[6] Edgar G. Daylight, *The Dawn of Software Engineering. From Turing to Dijkstra*, Lonely Scholar, 2012.

[7] Paulette Destouches-Février (ed.), *Applications scientifiques de la logique mathématique: Actes du 2e colloque international de logique mathématique, Paris, août 1952, Institut Henri Poincaré. Centre d'études de logique symbolique*, Collection de logique mathématique, Gauthier-Villars, Louvain, E. Nauwelaerts, Paris, 1954.

[8] R. Feys, 'Un colloque sur le raisonnement scientifique (Paris - 26 septembre - 1er octobre 1955)', *Revue Philosophique de Louvain*, **53**(40), 636–639, (1955).

[9] M. Guillaume, 'La logique mathématique en France entre les deux guerres mondiales : quelques repères', *Revue d'histoire des sciences*, **62**(1), 177–220, (2009).

[10] Heinz Gumin, 'Digital computers, mathematical logic and principal limitations of computability', in *Information Processing 1962, Proceedings of IFIP Congress 62*, ed., Cicely M. Popplewell, Munich, (1962). North-Holland.

[11] Daniel Lacombe, 'Extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles I', *Comptes Rendus Acad'emie des Sciences Paris*, **240**, 2478–2480, (1955).

[12] Daniel Lacombe, 'La théorie des fonctions récursives et ses applications (Exposé d'information générale)', *Bulletin de la Société Mathématique de France*, **88**, 393–468, (1960).

[13] A. Lentin. Projet de réforme de l'enseignement à l'institut de programmation. Archives de l'Institut Blaise Pascal, s.d. (certainly 1st semester 1966).

[14] A. R. D. Mathias. Bourbaki and the scorning of logic. ms., 2009.

[15] Pierre Mounier-Kuhn, *L'Informatique en France, de la seconde guerre mondiale au Plan Calcul. L'émergence d'une science*, Presses de l'Université Paris-Sorbonne, 2010.

[16] Pierre Mounier-Kuhn, 'Computer science in French universities: Early entrants and latecomers', *Information & Culture: A Journal of History*, **47**(4), (2012).

[17] L. Nolin, 'Quelques réflexions sur les langages de programmation', *Chiffres*, **6**(1), 11–12, (1963).

[18] Jean Porte, *Recherches sur les logiques modales, congrès international du CNRS sur Le Raisonnement en Mathématiques*, CNRS, Paris, 1958.

[19] D. Tamari, *Machines logiques et problèmes de mots. I : les machines de Turing (T.M.)*, volume 2 of *Séminaire Bourbaki*, 1951–1954.

[20] Karel van Oudheusden. The advent of recursion and logic in computer science, 2012. Master's dissertation in logic, supervised by Gerard Alberts, University of Amsterdam. http://www.illc.uva.nl/Research/Reports/MoL-2009-12.text.pdf.

[7] See for instance the in-depth study conducted by Karel Van Oudheusden, Turing's 1936 Paper and the Origin of Computer Programming – as Experienced by E.W. Dijkstra, and [20] and Edgar G. Daylight (an author closely related to the former) [6].

Is Plugged Programmed an Oxymoron?

Allan Olley[1]

Among the key "special purpose" calculating technologies in the 1930s were the punched card machines. These machines were used by business and scientists in performing various calculations both before and after the creation of the first computers. Many computer pioneers were familiar with the technology and used them before or alongside full electronic digital computers. [2] Such machines were the mainstay of IBM's business before and even for some time after the invention of the computer. In this paper I will address in what ways the control and orchestration of machine calculation on these machines is analogous with modern programming. I will also suggest some of the implications for the modern paradigm of programming suggested by the contrast with this earlier system.

The punched card system consisted of cards that encoded numbers and machines that performed functions on the cards. In general each machine performed only one kind of operation such as addition (tabulation), multiplication, sorting, or copying. Results of calculations could simply be displayed on a counter, printed in paper and ink or punched onto other cards. A calculation would often involve passing stacks of cards between two or three machines. These machines included a complex control mechanism that involved the setting of various switches, the plugging of wires to specify where on the cards numbers would be read from and to set-up for functions such as clearing a total, printing sub-totals and so on. These control functions were actuated by indications on the card. The indication was usually either a special punch or the detection of a difference between subsequent cards. [5] Usually, each specific sort of calculation done on these machines would require a specific set-up.

The capacity of these machines is best illustrated with some examples. In 1928 L. J. Comrie carried out a harmonic synthesis to calculate the positions of the Moon for a ten year period for the British Navy's navigational and astronomical tables. The calculation involved adding hundred of trigonometric terms at regular intervals. The key to executing the calculation was to punch the values of the various trigonometric series on sets of cards at appropriate intervals, each function was now embodied as a stack of cards. An element of the position could then be calculated by simply assembling an appropriate set of stacks (functions) and taking the appropriate card from each stack and feeding them into a tabulator that added the value on each card. The drawback was that Comrie's team needed to prepare over half a million cards. Despite the monumental amount of preparation required Comrie felt that the use of the machines had led to a reductions in time, error and cost. He made partial preparation to use the system to calculate the position of the Moon to the year 2000. [3]

Comrie's success illustrates the power of the machine and one of the key elements in performing more technically demanding computations, the need for cards with prepared functions on them. In this case a punched card table added the calculation of trigonometric terms to the repertoire of a machine that had formerly only added numbers. Printed mathematical tables for hand calculation methods had been the source for the values used in the computation.

The importance of the cards also suggests that the cards were not merely an inert storage system. The cards themselves were part of the data processing of the machines. This is not just due to the fact that a stack of cards is a serial storage mechanism. For example the sorter physically separates the cards into ten piles, running the card through the machine was coextensive with the machine's operation. The sorter performed its work at a faster rate than any other punched card machine, as a result, 20 000 cards an hour. More generally the comparison of successive cards was key to the control mechanism of punched card machines. This comparison was achieved by running cards through a set of comparison brushes simultaneously with the previous card running through the main brushes. The card pass was the unit of operation in the punched card machines and constrained the number of operations performed or initiated. Again the passive of the card through different parts of the machine was a key part of the machines operation.[5] The card served not only a storage but also a control function.

Even a well prepared set of cards could not compensate for the need to set-up the machine to calculate according to a different format. Problems such as numerical integration where each step relies on the previous step pose a special problem. The various steps required in one iteration required different calculations and therefore different machine set-ups. If only a single integration was being performed this would mean rewiring the machines after each card pass.

Wallace J. Eckert was a professeur at Columbia University in New York and had worked on using an experimental punched card machine for parts of his numerical integration of asteroid orbits in the early 1930s. He went on to propose to IBM the construction of a device called the Calculation Control Switch that was completed in about 1935. This device used a set of discs notched discs, relays and switches to rewire and toggle various elements of a tabulating machine, multiplier and an associated unit. Essentially it allowed the machines to perform 12 prearranged steps. This allowed the operator to perform the various different additions, multiplications and card punches by feeding the cards through the machine, making some ancillary calculations and sometimes pressing a button on the Control Switch (it proceeded automatically in some cases). [5]

In other cases Eckert took advantage of the speed with which punched card machines could perform the same operations. Faced with performing a large number of common additions and subtractions, he simply sorted the like cases and copied the correct answers. [5]

Another way to make the punched card machines more flexible was to increase the number of operations that could be performed during a single card pass. A standard IBM tabulator of the 1930s could add five numbers to five separate counters in a single pass and also reset the a counter to zero or print output if prompted. The IBM

[1] Independent Scholar, email: allan.olley@utoronto.ca

601 Multiplier could also perform two additions or subtractions on the product it calculated. [5] The German DEHOMAG 11 tabulator took this option even further it could carry out five additions or subtractions and a multiplication in any order on a card pass. [6] While this allowed a more complex sequence to be carried out, the same sort of calculation would be carried out on each card pass.

Also, unless performed in parallel more calculations would slow the rate of card reading. As faster electromechanical and electronic technology was deployed in calculating machines during and after World War II punched card machines were created that carried out more steps during a single card pass. The IBM Pluggable Sequence Relay calculator built in 1944 for the Aberdeen proving grounds ballistic work could perform 48 steps that were equivalent to as many as 24 additions or 4 multiplications per card pass. This calculator processed at a maximum of one hundred cards a minute and could perform various operations in parallel.[4]

This technological lineage would reach its ultimate form in the IBM Card-Programmed Electronic Calculator (CPC), released in 1949, which combined an electronic multiplier and various other machines to create a machine capable of carrying out almost any arithmetic sequence as specified by the stack of cards read into it. The CPC still had plugboard based control, but a single setting of the plugboard would allow for a wide range of calculations. The CPC would be the last major "plugged program" machine and the stored-program machine would dominate machine design afterwards. In one case IBM rejected a plan for a new electronic plugged programm machine in favour of a stored-program alternative, explaining that set-up and operation required far more skill and attention than even though it might also yield superior results in the right hands. [1]

The similarity between plugged punched card machines and programmed computers is that as with a stored-program computer this automation required breaking a problem into a series of steps and arranging for their execution by machine. Both allowed the automatic execution of a plan. However, in punched card machines there was no clear separation between data entry and instructions. The sequence of cards and the machine set-up combined to determine the kind of computation performed. Therefore while the operation of the machine was abstracted into mathematical terms and into engineering instructions for wiring the machine and setting switches to carry out each step, problems were not described in terms of a full set of generic steps in a specific order, like lines of instruction code (a program). Different problems were not given a formal similarity.

The contrast between the control of punched card machines and modern programming techniques suggests how the notion of a program abstracts from the particular physicality of a machine. Those who design basic computer architecture, program machine code must worry about these issues and understand and work with the exact limits of the machine, but most computer users and programmers do not work at this level. Abstraction is a powerful tool for allowing techniques to be shared between people and computers. However, lack of attention to the particulars of a machine makes optimizing operation for a problem difficult. One example is the common use of parallel computations in plugged machines compared with the difficulties associated with parallel computing in the stored program context.

ACKNOWLEDGEMENTS

I would like to thank the referees for their comments which helped improve this paper.

REFERENCES

[1] C. J. Bashe, L. R. Johnson, J. H. Palmer, and E. W. Pugh, *IBM's Early Computers*, MIT University Press, Cambridge, Mass., 1986.

[2] P. Ceruzzi, *A History of Modern Computing*, MIT University Press, Cambridge, Mass., 2nd edn., 2003.

[3] L. J. Comrie, 'The application of the hollerith tabulating machine to brown's table of the moon.', *Monthly Notices of the Royal Astronomical Society*, **92(7)**, 694–707, (1932).

[4] W. J. Eckert, 'The ibm pluggable sequence relay calculator.', *Mathematical Tables and Other Aids to Computation (MTAC)*, **3 (23)**, 149–161, (1948).

[5] W. J. Eckert, *Punched Card Methods in Scientific Computation*, volume V of *the Charles Babbage Institute Reprint Series for the History of Computing*, MIT University Press And Tomash Press, Cambridge, Mass. and Los Angeles/ San Francisco, 1984.

[6] F. W. Kistermann, 'The way to the first automatic sequence-controlled calculator: The 1935 dehomag d 11 tabulator.', *IEEE Annals of the History of Computing*, **17(2)**, 33–49, (1995).

On the Nature of the Relation between Algorithms and Programs

Uri Pincas[1]

Abstract. 'Algorithm' and 'Computer-Program' are two of the very fundamental terms of computer-science and programming. Here we try to clarify the nature of relation between these two terms and shade some light on the issue. By first looking with a one-directional scheme of "theoretical" and "practical" perspective, we can consider some relatively simple relation between algorithms and computer-programs; but a deeper and broader examination of the dynamic interconnections between the two may lead us to view the matter as more complicated.

1 ALGORITHMS

The notion of algorithms lies at the essential core of computer-science. An algorithm, in some general basic (rather loose and primitive) sense, is an instruction for doing something by simple actions according to some systematic manner. In this sense, a casserole-recipe or directions of getting from one town to another are algorithms. However, here we are interested in algorithms in the more computational definite sense, as the term is used and discussed in computer-science contexts.

It is interesting to notice that though the notion is very central and has been (being) discussed and elaborated on for quite many years, there is currently no absolute consensus about the exact definition (and exact meaning) of 'Algorithm', and the attempts to define this term pose rather intricate issues in still ongoing debates (see, for example, [21], [14]).

But let us accept here some "common-sense" characterization of the term, somewhat similar to the one mentioned above, but more accurate and "computer-scientific" (though applies to broader realms than strict computer-science): By 'Algorithm' we refer to a general prescription for performing some definite ("computational") task; i.e. a description of operational steps, each of which is well-defined by itself, which are to lead to some result, if executed according to some unambiguous order (see, for example, [15], chapter 1).

2 COMPUTER-PROGRAMS

A large and significant part of computing is, in a very natural way, computer-programming; that is to say—the designing, writing, improving, running, maintaining and analysing of computer-programs. Computer-programs can be written in thousands of programming-languages, each language differs from others by characteristics of semantics, syntax, purposes and usages. Here we focus on the notion of computer-program in general.

As in the case of 'Algorithm', the term of 'Computer-Program' is very basic and long-discussed, but exact definition and characterization of its meaning are still in dispute (see, for example, [20], [30], [8]; see also [27] for other issues and sources dealing with many aspects of the subject).

Then again, we will refer here to a "common-sense" characterization of the term: by 'Computer-Program' we mean an ordered set of instructions in some computer-language, which are to be run on some computing-environment (computer-system) in order to perform some task (see, for example, [15], chapter 3).

3 FIRST CONSIDERATION OF THE RELATION BETWEEN ALGORITHMS AND COMPUTER-PROGRAMS

When primarily considering 'Algorithm' and 'Computer-Program' as related to each other, two conspicuous features may be noticed (see, for example, [10]):

- One is the more theoretical or abstract nature of the algorithm compared with the more practical or "down-to-earth" nature of the program. As described here above, computer-programs are written in some specific computer-language and are to be (or at least are supposed to be capable of being) run and executed on some concrete computing-environment; while algorithms are more theoretical objects, not tied to such a specific operational environment, and so can be described in a far more general and abstract manner, with much fewer limitations of practical contexts.
- A second feature is the fact that the algorithm precedes the program. It means that when someone is to design, write and run a computer-program in order to perform some task, he or she is very likely to start by designing (or taking an already-designed) algorithm which accomplishes this task on the principal algorithmic-conceptual level, and then construct the program by the lines of that algorithm.

These two features may be summarized by saying that the computer-program is an **implementation** of the algorithm[2].

[1]Department of Mathematics and Computer Science, The Open University of Israel, 1 University Road, POB 808, Raanana 43537, Israel. Email: uri.pincas@openu.ac.il.

[2]Here again we use the term 'implementation' in some of a simplifying sense. It is worth to mention that this term (also referred to as 'realization' by some of the discussants), as well as 'algorithm' and 'computer-program' mentioned above, has been discussed and disputed in many works (see, for example, [19], [24], [3], [28], [26]), and we would not dwell on all the interesting philosophical aspects of the matter here.

4 SOME MORE REFLECTIONS ON ALGORITHMS AND COMPUTER-PROGRAMS

Reconsidering the issue may help us understand it more deeply. Every algorithm (as well as every computer-program) is composed of operations, each of which is considered to be elementary, i.e. assumed to be understood and executed by itself, without having to be divided to more basic operations. Characterizing such operations as elementary depends on the adopted computational model, which is taken as the general frame of computation. Many of the "canonical" computational models are equivalent, in the sense that the set of computable functions of one model is the same as of any other of its equivalents (see, for example, [6]). However, this does not mean that the elementary operations of every model are the same as of any of its equivalent; and so the way of performing some computation, by constructing the required algorithmic object from basic operational building-blocks, in two different computational models can be different (see [4], [17], [23], [31])[3].

Adopting some computational model for designing algorithms is sometimes taken to be carried out "once and in advance" (indeed, methodologically such manner of thinking may be convenient and fertile in some contexts), but we can notice that in practice this is not the case, since having some model taken and accepted as a computational framework is a dynamic multi-stage process. Conventions of a "suitable" computational model stem from some theoretical background and are adopted as a general primary frame of algorithmic praxis; when some practice of designing and constructing algorithms (and their implementations) is exercised and shaped, conventions of "suitable" elementary operations and their organization in algorithmic schemes, as well as principles, methods and criterions for evaluation and improvement of the "quality" of such schemes, are recreated, formed and accepted. Such a phase of formation of an "actual" algorithmic model—based on a theoretical computational model, but not identical with it, as it is characterized and practiced on a different level—is not single, because as more algorithmic work is continued to be done, more changes of the algorithmic model do take place. It is true that most of these changes may be described as minor and not revolutionary, but nevertheless they make some epistemic and operational difference.

In this context, some historical note might be in place. Significant interest in research of the algorithmic problems, solutions and processes as a body of knowledge (what might be considered as the theory of algorithms) arouse with interests in the essence and foundations of mathematics—and so in reflexive mathematical processes—around the end of the 19th century, continuing intensively to the first decades of the 20th century. The mathematical and logical research of that time focused on general properties of mathematical and axiomatic systems and on processes within their frameworks, not on ("quantitative") analysis of detailed operativity of such processes. This second kind of interest arouse and was largely developed later, with the acceptance and establishment of the computational model of Turing as a general characterization of computing, on one hand, and with a lot of practical motivations to improve and make efficient the performance of computational processes, on the other hand (see, for example, [7] and [22], chapter 6).

Having this in mind, we should note that computer-programs, as described here above, are implementations of algorithms, and so are essentially entangled with many aspects of practical business. This makes them a considerable factor of shaping the model-frame of algorithm design. And so we can consider here a relation of "computer-programs effecting algorithms", in the opposite direction, so to speak, of the one mentioned here at part 3, of "algorithms effecting computer-programs".

5 EXAMPLES OF "COMPUTER-PROGRAMS EFFECTING ALGORITHMS"

5.1 FIRST EXAMPLE—ARITHMETICAL COMPUTATIONS AND ELECTRONIC ELEMENTS

One example is the development of algorithms and computer-programs for arithmetical computations. A typical basic usage of electronic computers is, very naturally, arithmetic computation. For performing such a computation, algorithms which compute the required result by "ordinary" arithmetic manner can be designed (and then implemented as computer-programs for actually carrying out the computation).

However, in the early "computer-era" of the forties and fifties of the 20th century, a computer-operation of multiplication was considered as "expensive", compared to the relatively "cheap" operation of addition (due to electronic features of the construction and operation of the computers of those times). As a result of that, actual programs run on computers were preferred to perform fewer multiplications, even in the price of preforming many more additions. This condition led to the designing of different arithmetic algorithms with more additions and fewer multiplications, not only in sporadic cases but as a general paradigm of the computerized arithmetic computation area (see, for example, [12], part III).

Nowadays, as addition and multiplication operations are considered as equally priced (since the electronics of computers has considerably changed), this paradigm is no longer valid—there is no actual advantage by saving multiplications on the expense of extra additions—and algorithms and computer-programs for arithmetic computations are designed by different considerations.

5.2 SECOND EXAMPLE—MEASURING COMPUTATIONAL COMPLEXITY AND COMPUTER-PROGRAMMING PRACTICE

Another example is our evaluation of the efficiency of algorithms and computer-programs. Different (equal and

[3] It is interesting to note the (peculiar?) fact that some of these very canonical equivalent computational models have been emerged and developed at about the same time from different (intersecting and overlapping) directions. For a historic survey and discussion of this issue, see [11].

simultaneously evolved) developments of characterizations of the term 'Computable'—i.e., different computational model—have been the basis of the theory of "hardness of computation", which has later been rooted as the theory of complexity of algorithms. Important mathematical works in the fifties of the 20th century, like [13] and [25], characterized (some sense of) computational complexity by relating to certain mathematical properties (such as the degree of recursiveness); but later on these characterizations were formed and reformed as the more modern senses of computational measures by considering algorithm efficiency. There are different meanings of 'Computational Complexity', and so there are different ways to measure the efficiency of algorithms and complexity-measures of different kinds (see, for example, [22], chapter 6). Two of the more common complexity-measures are time complexity—the number of basic operations which the algorithm executes—and space complexity—the number of memory-units which the algorithm requires[4].

Which of these two measures (and of other complexity-measures as well) is to be considered as the more representative for algorithm efficiency? The answer to this question was motivated by "actual" computing and evaluation of computer-programs in practice. Again, back in the first days of electronic computers, computer-programs were relatively short and simple, and the time taken for most computer-programs to run was less important a factor than the storage they had to use, which was expensive. In such state of affairs, space complexity was a candidate for the dominant measure of computer-program (and algorithm) efficiency. As computing technology evolved and developed, computer-storage became relatively cheap, but the programs became longer and more complicated in a manner which made computing-time a more valuable resource. This being the situation, as computer-programs were attempted to be less time-consuming, and, accordingly, algorithms were designed to have fewer operations, time complexity became the most dominant and common measure for efficiency, and remains so up until nowadays[5] (see [2]).

The effect of the practical "preponderance of time" is evident in following developments in the theory of computational complexity, as criteria of time-complexity have become the characterization of algorithms—and naturally afterwards of computational problems—as "reasonable", i.e., fulfilling the task (for algorithm) or capable of being solved (for problems) in what is to be considered as "realistic" conditions or limitations. Polynomial number of basic instructions, related to as *polynomial time*, has been set as the standard limit for reasonability in this sense, based on typical considerations of the computing practice of running computer-programs (see, for example, [5], [9], [16], [1]). The momentous definitions of computational complexity classes, such as **P**, **NP**, **NPC** and many more, which have become some fundamental business of the theory of computer-science, is a natural continuation of this characterization of reasonability by time-complexity sense (see also [18], [29]).

[4]These two measures are of the category known as *dynamic (or operational) complexity measures*—measures which relate to the behavioural of the algorithm as a computation being performed. Another kind is *static (or definitional) complexity measures*—measures which relate to properties of the algorithmic text as a description of the computation (like, for example, the length of the character-string which describes the algorithm) (see, for example, [2]).

[5]Somewhat interesting, in this context, may be the saying (whose origin is not clear) that time is more of an important (computing) resource than space, since space, contrary to time, is reusable.

6 SUMMARY AND CONCLUSION

Here we considered the relation between algorithms and computer-programs by characterizing the terms on different levels. First we looked at the basic semantic level, considering somewhat simple context of the terms; by that we noticed sort of a one-directional relation between the two, taking each program, by itself, to "come after" and to "put into practice" some algorithm, in the sense mentioned above of being the algorithm's implementation. Later we examined the subject in a broader perspective, taking into consideration some elements in the multi-context dynamics of algorithm and computer-program processes; by that we noticed that the relation between the (dynamic, ever-changing) realms of programs and algorithms is much more intricate and manifold.

We might end this paper by calling for recognition and examination of other and more elements which take part of the complex interrelationship between these two realms, influencing the relation between the terms as well as useful and illuminative for its characterization.

REFERENCES

[1] M. Blum. A Machine-Independent Theory of the Complexity of Recursive Functions. *Journal of the ACM,* 14(2): 322–336 (1967).

[2] A. Borodin. Computational Complexity: Theory and Practice. In: *Currents in the Theory of computing*. A.V. Aho (Ed.). Prentice-Hall, USA (1973).

[3] D. J. Chalmers. Does a Rock Implement Every Finite-State Automaton?. Synthese, 108: 309–333 (1996).

[4] A. Church. An Unsolvable Problem of Elementary Number Theory. The American Journal of Mathematics, 58: 345–363 (1936).

[5] A. Cobham. The intrinsic computational difficulty of functions. In: *Proc. Logic, Methodology, and Philosophy of Science*. Y. Bar-Hilel (Ed.). North Holland, Netherlands (1965).

[6] M. Davis. *The Undecidable*. Princeton University Press, USA, (1965).

[7] M. Davis. *The Universal Computer: The Road from Leibniz to Turing*. W. W. Norton and Company, USA, (2000).

[8] W. Duncan. Making Ontological Sense of Hardware and Software. **`http://www.cse.buffalo.edu/~rapaport/584/S10/duncan09-HWSWOnt.pdf`**. (2009).

[9] J. Edmonds. Paths, Trees, and Flowers. Canadian Journal of Mathematics, 17: 449–467 (1965).

[10] J. H. Fetzer. Program Verification: The Very Idea. *Communications of the ACM,* 31(9): 1048–1063 (1988).

[11] R. O. Gandy. The Confluence of Ideas in 1936. In: *The Universal Turing Machine: A Half-Century Survey*. Second Edition. R. Herken (Ed.). Springer, USA. (1995).

[12] H. H. Goldstine. *The Computer from Pascal to von Neumann*. Princeton University Press, USA, (1993).

[13] A. Grzegorczyk. Some Classes of Recursive Functions. Rozprawy Matematyczne, 4: 1–46 (1953).

[14] Y. Gurevich. What Is an Algorithm?. SOFSEM 2012: 31–42 (2012).

[15] D. Harel (with Y. Feldman). *Algorithmics: The Spirit of Computing*. Third Edition. Addison-Wesley, USA, (2004).

[16] J. Hartmanis and R. Stearns. On the Computational Complexity of Algorithms. Transactions of the American Mathematical Society, 117: 285–306 (1965).
[17] S.C. Kleene. 'General Recursive Functions of Natural Numbers'. Mathematische Annalen, 112: 727–742 (1936).
[18] H.R. Lewis and C.H. Papadimitriou. The Efficiency Of Algorithms. Scientific American, 238(1): 96–109 (1978).
[19] W. G. Lycan. The Continuity of Levels of Nature. In: *Mind and Cognition: A Reader*. W. G. Lycan (Ed.). Basil Blackwell, UK (1987).
[20] J. H. Moor. Three Myths of Computer Science. British Journal for the Philosophy of Science, 29(3): 213–222 (1978).
[21] Y.N. Moschovakis. On founding the theory of algorithms. In: *Truth in mathematics*. H. G. Dales and G. Oliveri (Eds.). Clarendon Press, UK (1998).
[22] U. Pincas. A Computer-Embedded Philosophy of Mathematics, Ph.D. Dissertation, University of Tel-Aviv, Tel-Aviv, Israel. (2008).
[23] E.L. Post. Finite Combinatory Processes—Formulation 1. The Journal of Symbolic Logic, 1: 103–105 (1936).
[24] H. Putnam. Appendix. In: Representation and Reality. MIT Press, USA. (1988).
[25] M.O. Rabin. Degree of Difficulty of Computing a Function and a Partial Ordering of Recursive Sets. Technical Report 2, Hebrew University (1960).
[26] W.J. Rapaport. Implementation Is Semantic Interpretation: Further Thoughts. Journal of Experimental and Theoretical Artificial Intelligence, 17(4): 385–417 (2005).
[27] W.J. Rapaport. What Is a Computer Program? **http://www.cse.buffalo.edu/~rapaport/584/whatisacomprog.html**. (2009).
[28] M. Scheutz. When Physical Systems Realize Functions. Minds and Machines, 9: 161–196 (1999).
[29] L. J. Stockmeyer and A. K. Chandra. Intrinsically Difficult Problems. Scientific American, 240(5): 140–159 (1979).
[30] P. Suber. What Is Software?. Journal of Speculative Philosophy, 2(2): 89–119 (1988).
[31] A.M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, ser. 2, 42: 230–265 (1936).