Attachment A: Research Programme

Systems Verification — The Last Mile

Dr Magnus Myréen, University of Cambridge, UK

1. Purpose and aims

In a recent landmark achievement, the L4.verified project from NICTA, demonstrated that it is possibly to prove strong safety and security properties of a *general-purpose* operating system [SOSP'09]. Like all other systems verification projects, they made a number of simplifying assumptions, e.g. the C compiler is trusted, inlined assembly is assumed to be correct and the boot code is right.

The proposed project will establish new approaches for the construction and verification of reliable systems software, which will allow future systems verification projects to avoid many of the current simplifying assumptions. The plan is to enable this by modelling the actual behaviour of computer hardware and providing proof methods which scale to the point where functional correctness of operating systems can be mathematically proven down to the level of machine code, not just C code. Ultimately, the aim is to have systems where every hardware peripheral has been specified and every single machine instruction is part of the verification.

We aim to push for significant proof automation. For the most part, this automation intends to help with interactive verification. However, at the extreme end, this project will look into how component implementations that are 'correct-by-construction' can be automatically derived from specifications of the desired behaviour and models of the underlying hardware.

In order to make sure that all reasoning strictly follows the rules of a formal logic, we will develop this work within the HOL4 theorem prover — a readily programmable higher-order logic prover; and in order to ensure relevance and impact of this work, I will continue and deepen my collaboration with the high-profile L4.verified project.

2. Motivation

As computer-based systems become embedded in exotic locations — credit cards, phones, radios, washing machines, medical devices, and parts of cars, airplanes and buildings, etc. — we become reliant on them and, therefore, are increasingly in need of rigorous engineering methods that achieve robust computer systems.

At the very base of every software system lies an operating system. This provides the foundation for all other programs. The safety and security features of all software that runs on top of a the operating system are immediately compromised if the operating system is faulty. When operating systems fail, the entire computer becomes vulnerable as process isolation can break down, service can be disrupted, data can be lost, and intruders from outside might be able to hijack control.

As such key pieces of infrastructure, operating systems ought to be developed using proper engineering methods and be rigorously validated. The unfortunate reality is that operating systems and other systems software are developed based on only the intuition of systems programmers. The only validation that is done is testing, i.e. programs are run on sample inputs. Testing may suffice for certain programs, but is very difficult and unreliable to apply to systems software, since these interact very intimately with hardware, e.g. memory management units (MMUs), and certain faults only appear in complex configurations.

An emerging and increasingly relevant approach to gaining higher levels of assurance is the use of mathematical proof. This form of *formal verification* can be used to show that the software has its intended behaviour for all possible inputs, not just a finite set of test inputs. Formal verification and other techniques based on solid logical foundations (boolean algebra, model checking, SAT, BDDs, AIGs etc.) are currently used by all major hardware vendors, since, for them, the cost of a fault in the post silicon phase is so devastatingly high that they invest heavily in finding flaws early. Compared with software, formal verification of hardware is arguably easier to automate.

Formal verification for software has generally been perceived as only being applicable to toy languages and small programs, particularly if the properties to be proven are deep functional specifications. This myth is slowly being dispelled thanks to recent high profile projects that have shown that this is not the case. Most relevant to this proposal is the landmark achievement by the L4.verified project from NICTA [SOSP'09]. They demonstrated that it is possibly to use mathematical proof to show that a C implementation of a *general-purpose* operating system with *decent real-world performance* meets a high-level functional specification, and as a consequence satisfies a number of strong safety and security properties.

As a demonstration of what formal verification can achieve, the work of the L4.verified team is excellent; they successfully opened the eyes of many formal methods skeptics. However, the L4.verified project also did not go far enough. As with many other projects on operating systems verification (see **Survey of the field**), the L4.verified project make a number of simplifying assumptions. Systems verifications to date, with very few exceptions, all fall short of proving the actual binary code correct with respect to well-defined models of real hardware. Nearly all make simplifying assumptions about the underlying hardware, leave gaps in their proofs where hardware interaction occurs, and fail to prove the actual binary code which runs on the hardware.

Nearly all verification so far has targeted the programs as they are expressed in the implementation language, typically C, and thus completely ignore the fact that compilers and linkers can introduce faults. A consequence of sticking to the implementation language, usually C, is that it becomes very hard to give proper semantics to inlined assembly in the presence of the source language semantics and a compiler. No current operating systems verification assigns anything close to a proper semantics to the unavoidable inlined assembly.

Going below the level of C code necessitates actually modelling hardware and dealing with machine code, which is much more detailed than C code, requiring proof tools and modelling to scale much further.

The purpose of this project is to

- provide reusable, detailed and trustworthy specifications of common hardware peripherals, such as MMUs, timers, UARTs and non-volatile RAM,
- establish proof methods that can make reasoning about such hardware specifications manageable and scale well even in the presence of all the detail one encounters when reasoning about machine code, and
- complete a number of increasingly complex case studies that will be used to showcase the research advances made in the two points above.

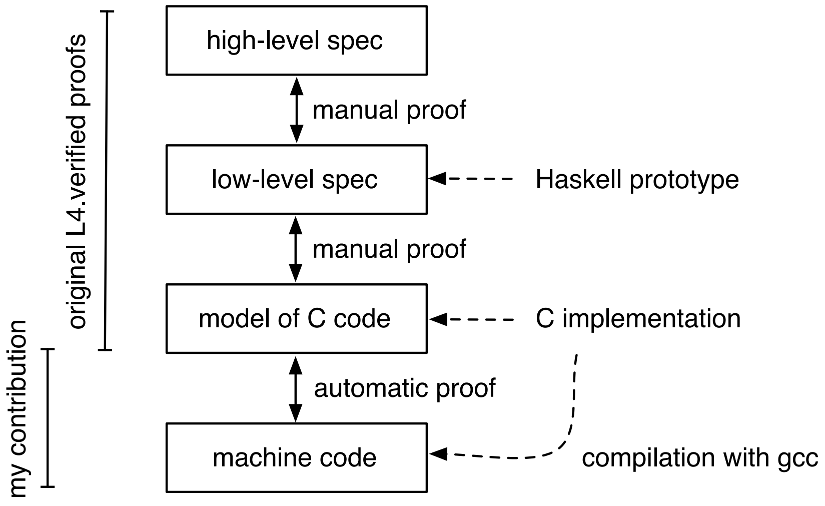
3. Preliminary results

The two most significant simplifying assumptions that project on systems verification make are that (i) the C compiler is correct w.r.t. the semantics used for verification, and (ii) the inlined assembly achieves the low-level state change (e.g. cache flush, MMU reconfiguration) that it is supposed to accomplish. It is very difficult, if not impossible, to give a proper semantics for inlined assembly as part of a C semantics. As a result, a prerequisite for tackling these assumptions is to move reasoning down to the machine code, where proofs target the generated machine code, not the source code.

Much of my previous work, including my on-going collaboration with the L4.verified project, centres around verification of machine-code programs. For this project I intend to use and extend my previously developed verification technology.

My work has so far targeted ARM, Intel x86 and IBM PowerPC processors. In particular, I have a proof-producing decompiler [FMCAD'08, PhD'09, FMCAD'12], which given some machine code for any of the processors mentioned above will automatically, via proof, derive a functional program from the machine code. This functional program is a record of the state change the machine code can perform. The beauty of this tool is that, for each decompilation, it produces a certificate theorem which allows all subsequent reasoning to be performed over the functional program, since the certificate theorem states that the generated functional program is an accurate record of all behaviours of the original machine code w.r.t. a formal semantics of the underlying machine language.

This tool has been successfully used in a number of case studies. The most relevant case study to this project is its use within the L4.verified project. During visits to the L4.verified team at NICTA, I have successfully applied my decompiler to the machine code that the latest version of the GNU C compiler (gcc -O2) produces for their verified operating system. Their original verification proof bottomed out at the level of a C implementation. With the aid of my decompiler, Thomas Sewell (of NICTA) and I managed to extend their entire proof down to the level of concrete ARM machine code and thus remove the assumption that the C compiler correctly compiles their kernel [PLDI'13]. The new structure of the L4.verified proofs is shown in the figure below.



Note that this extension of the L4.verified project's proofs did not discharge their assumption regarding inlined assembly. The inlined assembly has not yet been verified (at any of the levels), since they do not have a specification of a semantics for the hardware peripherals that these inlined assembly sections interact with. The proposed project will produce such semantics and can thus facilitate a further extension which would *complete* the bottom end of the L4.verified proofs.

The decompiler is a good tool for post hoc verification of existing machine code. However, often it is more desirable, if possible, to generate code that is correct by construction. For this purpose, I have also constructed system-building tools that can synthesise correct-by-construction machine code from high-level specifications [CC'09, ICFP'12]. These tools have enabled me to successfully produce functionally correct implementations of sizeable applications, such as implementation of functional programming languages [TPHOLs'09, ITP'11] and just-in-time compilers [POPL'10]. The two approaches, post hoc verification and synthesis, can be combined to provide local optimisations for code fragments whose execution speed is critical, e.g. in fast paths, inside the synthesised code.

4. Project description

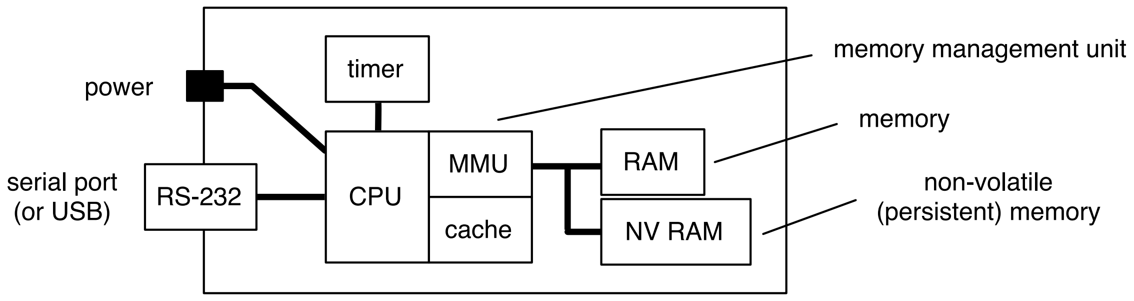
Although my research has made significant contributions to verification of machine code, many challenges remain before full systems verification can be achieved. Being able to prove machine code is not in itself enough to tackle the detailed interaction between hardware and software that operating systems require, e.g. boot code which initializes the state, interrupt handlers which react to concurrent events, and code which processes I/O and deals with memory management via the hardware.

Full systems verification requires giving a semantics to the hardware components (interrupts, memory, MMUs, I/O devices) that such code communicates with. These new models will pose new challenges for proofs. New proof methods and automation will need to be devised that fits these more precise underlying hardware semantics.

The bulk of the effort is thus production of formal specifications for common hardware components and development of related proof methods. The research challenges will be tackled in a step-by-step manner through a series of increasingly complex case studies. The case studies are designed to keep the project on track and also showcase its results.

4.1. Target case study

The final target case study is to run on hardware where all of the modelled hardware components exist. The following diagram sketches such a configuration.



This hardware diagram consists of all the basic components of a simple but complete system: the RS-232 port provides I/O, the CPU performs the computation, the timer can notify the CPU when time has come to perform a potential task switch, the MMU provides memory protection (helps isolate processes), the RAM together with the cache are the non-persistent memory and finally there is persistent memory that survives power off. The power component is included here in order to emphasise that this project will consider the possibility of sudden, potentially malicious, power cuts. Can such power cuts leave the persistent memory in a confused or vulnerable state?

This hardware configuration is an only slightly simplified version of real development boards such as the Raspberry Pi, the PandaBoard or the BeagleBoard. All of these are cheap ARM-based test boards that the author has experience in using.

The target is to model all of these hardware components and then to write and verify a small operating system that sits on top of such a hardware configuration. The example system that is being considered is to implement a cryptographic token, i.e. a device which holds a secret key and provides a cryptographic signature service: given some data to sign, it outputs a *hash* that is computed from both the input data and the secret key. For such systems we want to be able to prove that, if the verified machine code is installed, then the entire device exhibits only behaviours that respect a clean high-level functional specification of its allowed behaviours, even in the presence of erratic power on/off events that aim to confuse the state of the NV RAM.

This case study allows for gradual development: one can start with a nearly stateless version which just echos input over to output; such an echo device can then be extended to echo an encrypted version of the input; and the next version can make use of the persistent store and so forth. The most complex configuration allows user programs to run on the CPU and issue encryption requests to the operating system, which would protect the secret using memory protection provided by the MMU.

4.2. Approach

There are a number of challenges in completing the target case study. Here is my strategy for tackling these challenges:

Managing the details

Without care, the sheer volume of details in any formal proof of realistically modelled hardware is easily overwhelming. To make sure that no detail is dropped or overlooked, all reasoning will be carried out inside an interactive theorem prover [TPHOLs'08]. This ensures that all proofs follow from the primitive inferences of a formal logic. This project will use the HOL4 theorem prover as its higher-order logic provides convenient means of abstraction and its system is easily programmable.

Modelling the hardware

For the hardware modelling, the intention is to start from Fox's formalisation of the ARM instruction set architecture (ISA) and treat that as a model of the ARM CPU. Fox's latest model lives within a domain specific specification language, called L3 [ITP'12], which he is currently developing. By using L3 to model the hardware components, these can essentially be made into parts of the ARM inside L3. The benefits of using L3 are that it has a neat syntax which was designed to be able to look very similar to the data sheets found in hardware manuals; L3 also has a very good back-end for HOL4. It generates corresponding HOL4 definitions from L3 files and at the same time also sets up some basic automation that helps perform fast 'evaluation' of the models by logical inference in the prover.

A further challenge is to keep the hardware models generic enough. For example, each hardware board has its own version of an RS-232 port and each such port is conceptually near identical, but different in the names of pins, detail of protocols etc. To counter this, there will be generic specifications and instantiations of the generic specifications.

Dealing with concurrency

Efficient I/O, among other things, is typically done through interrupts. Interrupts introduce single-processor concurrency. My previous work on machine-code verification does not deal with concurrency. Fortunately, the concurrency found here is very asymmetric in nature. Interrupts jump (non-deterministically) into the system's interrupt handler and the handler *usually* turns off interrupts, deals with the interrupt and then returns while switching interrupt on again.

Such interrupts pose a challenge for code verification. The plan is to address this challenge by first verifying the interrupt handlers (assuming no interrupts happen since these are usually turned off) and then using a trick I call *folding in the verified specification into the hardware model*. The idea is that the user-mode code is to be verified in the presence of the hardware and the interrupt handler. Since the interrupt handler is verified, we know it has a specific behaviour and can treat its actions as just executing its verified specification. In other words, the model which is used to for user-mode code assumes a hardware model consisting of the original hardware model with the specification of the interrupt handler superimposed. From the point of view of the user-mode code, interrupt do not happen, instead the extended hardware has a richer set of actions it can perform.

A worry here is the specification of spurious interrupts. Verification proofs must show that the system is resilient against some kinds of spurious interrupts.

Retaining proof automation

The previously developed proof automation, mentioned in Section 3, needs to carry over to the hardware-enhanced semantics of machine code. Conceptually there are two challenges to overcome here: concurrency from interrupts and non-determinism from the hardware. However, the addition of concurrency can be handled if the approach for folding the verified interrupt handling into the hardware model is followed. The real question is how genuine non-determinism is to be handled. My original approach to decompilation produces deterministic functions describing deterministic machine code. Such functions will not immediately suffice to describe all behaviours of non-deterministic machine code. One possible standard solution is to pass around an 'oracle function'. Another solution is to modify the decompiler to produce relations instead of functions. The best alternative will become clear in the case studies.

Synthesis of special code

Having detailed models of the hardware opens up the possibility of synthesising efficient component implementations from models and behavioural requirements. This will be based on the synthesis tools described in Section 3. Examples that will be considered are synthesis of boot code that initializes the hardware into an appropriate state and simple interrupt handling routines that correctly identify the type of interrupt and act accordingly, e.g. just jump to the appropriate routine or perform some basic update to the global state and then return.

4.3. Time management

This is a four-year project where the work will be organised as follows. The focus of the first year will be modelling the interrupt-driven UART and NV RAM and developing solid proof technology to fit with these models. The second year is devoted to the MMU model and making the methods from the first year scale. The third and fourth years of the project will concentrate on completing the most advanced form of the target case study. Starting from year two onwards there will be a parallel thread of activity experimenting with the use of synthesis of code from specifications.

5. Survey of the field

The desire to have systems verification has been present since Goldstein and von Neumann [Neu'61]. However, the first significant systems verification project was the `CLI short stack' [JAR'89], an ambitious project which constructed a verified CPU, verified compiler, verified systems software — i.e. a full verified stack. All artifacts were built to be verified and I/O was never satisfactorily addressed. Boyer and Yu [JAR'96] showed that it is possible to verify machine code of a real commercial processor w.r.t. a detailed formal semantics. They verified object code for the Motorola MC68020. Proof-carrying code (PCC) by Necula [POPL'97] and typed assembly language (TAL) by Morrisett et al. [POPL'98] renewed interest in verification of low-level code and new advances in dealing with low-level pointer reasoning have also been made [LICS'02].

All of these advances made it clear that the time is ripe for systems verification. A number of project on systems verification emerged, most notably, the aforementioned L4.verified project and the Verisoft and Verisoft XT projects [VSTTE'10]. The Verisoft projects have been a diverse mix of research on verified special-purpose operating systems and a hypervisor, all verified at the level of source code and without I/O. Like TAL and PCC, there have also been systems verifications that focus on safety properties, most notably, Yang and Hawblitzel's type safe operating system [PLDI'10]. New supporting technology such as programming logics for systems like code have been developed, e.g. by Shao's group at the University of Yale [PLDI'08]. Systems verification has yet to be based on detailed models of hardware and I/O devices.

6. Significance

As mentioned above, practically all projects on systems verification to date have made simplifying assumptions about the underlying hardware, leave gaps in their proofs where hardware interaction occurs, and fail to prove the actual binary code which runs on the hardware. My previous work on machine-code verification — which I have already demonstrated to be very effective in systems verification as part of the L4.verified project — promises to provide a very good basis for tackling these shortcomings.

There is a strong desire in current systems verification projects to do better, to have better hardware specifications and to fill the current gaps in their formal arguments. As such, the project proposed here is timely and has the potential to have an immediate impact with its outputs, i.e. the open source models and proof methods, finding use by others very quickly. The work described in this proposal is likely to be published at venues such as PLDI, POPL, ITP, CAV and FMCAD.

The core value of this project is that it is designed to make progress towards a long-standing ambition of the computer science community: to make formal methods applicable directly to real operating systems, running on real hardware as real machine code.

7. International and national collaboration

The NICTA collaboration will be continued and deepened in order to ensure relevance and impact of this work. Many people follow the high-profile verification group at NICTA. If that group picks up and start using the technology developed for this grant, then widespread dissemination is practically guaranteed.

Other international collaborations include a close working relationship with Dr Anthony Fox from the University of Cambridge. He is the author of the extensively validated Cambridge ARM model and the L3 toolchain which will be used as part of this project. Other work at Cambridge may also turn out to be relevant, e.g. that by Prof Mike Gordon on secure devices and Prof Peter Sewell's group on lightweight rigorous methods. Another international collaboration that is expected to have an impact on this project is my work with Dr Jared Davis of Centaur Technologies Inc and University of Texas Austin on implementation of the next generation of theorem provers.

On the national level, I am keen to establishing links with the functional programming group at Chalmers University. Their expertise in design and use of domain-specific languages may turn out to be a very useful in developing both extensions of the L3 specification language and a source language for use for synthesis of systems software. Another group which I intend to keep a close connection with is Prof Mads Dam and his PROSPER (Provably Secure Execution Platforms for Embedded Systems) project, which is developing a verified hypervisor in HOL4 using the Cambridge ARM model.

8. Other grants

I currently hold a Royal Society University Research Fellowship (UK). This research fellowship can unfortunately not be used outside of the UK and will be suspended if this VR grant is funded. I will move to Sweden, if this VR grant is funded.

9. References

PLDI'13: Thomas Sewell, Magnus O. Myreen and Gerwin Klein. Translation validation for a verified OS kernel. To appear in Cormac Flanagan, editor, Programming Language Design and Implementation (PLDI), ACM, 2013.

FMCAD'12: Magnus O. Myreen, Konrad Slind and Michael J. C. Gordon. Decompilation into Logic — Improved. In Gianpiero Cabodi, Satnam Singh, editors, Formal Methods in Computer-Aided Design (FMCAD), IEEE, 2012.

ICFP'12: Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In Peter Thiemann, Robby Bruce Findler, editors, International Conference on Functional Programming (ICFP), ACM, 2012.

ITP'12: Anthony Fox. Directions in ISA specification. In Lennart Beringer, Amy P. Felty, editors, Interactive Theorem Proving (ITP), Springer, 2012.

ITP'11: Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, Freek Wiedijk, editors, Interactive Theorem Proving (ITP), Springer, 2011.

POPL'10: Magnus O. Myreen. Verified just-in-time compiler on x86. In Manuel V. Hermenegildo, Jens Palsberg, editors, Principles of Programming Languages (POPL), ACM, 2010.

PLDI'10: Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In Benjamin G. Zorn and Alexander Aiken, editors, Programming Language Design and Implementation (PLDI), ACM, 2010.

VSTTE'10: Eyad Alkassar and Mark A. Hillebrand and Wolfgang J. Paul and Elena Petrova. Automated Verification of a Small Hypervisor. In Gary T. Leavens, Peter W. O'Hearn and Sriram K. Rajamani, editors, Verified Software: Theories, Tools, Experiments (VSTTE), Springer, 2010.

SOSP'09: Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June,ronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, Symposium on Operating Systems Principles (SOSP), ACM, 2009.

PhD'09: Magnus O. Myreen. Formal verification of machine-code programs. PhD dissertation, University of Cambridge, 2008.

CC'09: Magnus O. Myreen, Konrad Slind and Michael J. C. Gordon. Extensible proof-producing compilation. In Oege de Moor, Michael I. Schwartzbach, editors, Compiler Construction (CC), Springer, 2009.

TPHOLs'09: Magnus O. Myreen and Michael J. C. Gordon. Verified LISP implementations on ARM, x86 and PowerPC. In Stefan Berghofer, Tobias Nipkow, Christian Urban, Makarius Wenzel, editors, Theorem Proving in Higher-Order Logics (TPHOLs), Springer, 2009.

FMCAD'08: Magnus O. Myreen, Konrad Slind and Michael J. C. Gordon. Machine-code verification for multiple architectures – An application of decompilation into logic. In Alessandro Cimatti, Robert B. Jones, editors, Formal Methods in Computer-Aided Design (FMCAD), IEEE, 2008.

TPHOLs'08: Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In Otmane Aat Mohamed, Cesar Munoz and Sofiene Tahar, editors, Theorem Proving in Higher Order Logics (TPHOLs), Springer, 2008.

PLDI'08: Xinyu Feng, Zhong Shao, Yuan Dong and Yu Guo. Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads. In Rajiv Gupta and Saman P. Amarasinghe, editors, Programming Language Design and Implementation (PLDI), ACM, 2008.

LICS'02: John Reynolds. Separation logic: A logic for shared mutable data structures. In Logic in Computer Science (LICS). IEEE Computer Society, 2002.

POPL'98: George C. Necula. From System F to Typed Assembly Language. In J. Gregory Morrisett, David Walker, Karl Crary and Neal Glew, editors, Principles of Programming Languages (POPL), ACM, 1998.

POPL'97: George C. Necula. Proof-Carrying Code. In Peter Lee, Fritz Henglein and Neil D. Jones, editors, Principles of Programming Languages (POPL), ACM, 1997.

JAR'96: Robert S. Boyer and Yuan Yu. Automated Proofs of Object Code for a Widely Used Microprocessor, 43(1), Springer, 1996.

JAR'89: W. R. Bevier, W. A. Hunt, J S. Moore, and W. D. Young. Special Issue on System Verification. Journal of Automated Reasoning 5(4), Springer, 1989.

Neu'61: Herman H. Goldstine and John von Neumann. Planning and coding problems for an electronic computing instrument. In John von Neumann, Collected Works, volume V. Pergamon Press, Oxford, 1961.