UDC 681.3.51

Wolfgang Goerigk, Hans Langmaack

REALISTIC CORRECT SYSTEMS IMPLEMENTATION

The present article and the forthcoming second part on *Trusted Compiler Implementation* address correct construction and functioning of large computer based systems. In view of so many annoying and dangerous system misbehaviors we ask: Can informaticians righteously be accounted for incorrectness of systems, will they be able to justify systems to work correctly as intended? We understand the word justification in the sense: design of computer based systems, formulation of mathematical models of information flows, and construction of controlling software are to be such that the expected system effects, the absence of internal failures, and the robustness towards misuses and malicious external attacks are foreseeable as logical consequences of the models.

Since more than 40 years, theoretical informatics, software engineering and compiler construction have made important contributions to correct specification and also to correct high-level implementation of compilers. But the third step, translation - bootstrapping - of high level compiler programs to host machine code by existing host compilers, is as important. So far there are no realistic recipes to close this correctness gap, although it is known for some years that trust in executable code can dangerously be compromised by Trojan Horses in compiler executables, even if they pass strongest tests.

In the present first article we will give a comprehensive motivation and develop a mathematical theory in order to conscientiously prove the correctness of an initial fully trusted compiler executable. The task will be modularized in three steps. The third step of machine level compiler implementation verification is the topic of the forthcoming second part on *Trusted Compiler Implementation*. It closes the implementation gap, not only for compilers but also for correct software-based systems in general. Thus, the two articles together give a rather confident answer to the question raised in the title.

1. Introduction

Users of computer based systems are often heavily annoyed by errors, failures and system crashes. In our every day experience using programs we observe them to fail, for instance due to lack of memory, programming errors, compiler bugs, or misuses of optimizing compilers under wrong assumptions. Although very annoying, we all live with software errors, but we still hope that application programmers, compiler constructors, operating system designers and hardware engineers have at least been sensible enough to detect and to signal any such error. Undetected errors might have harmful consequences, in particular if they are intentional, perhaps due to *viruses* or *Trojan Horses*.

Often the user is accountable, using systems outside their specified domains without even reading manuals or documentation. However, a large number of system misbehaviors is still due to the system constructors themselves, to professionals, computer scientists, *informaticians*. It is obvious that they should take responsibility as any professional has to. But software constructors and their companies hardly ever give guarantees for their products. And they are not even enforced to because customers purchase software products in full awareness of defects. Nevertheless, informatics scientists and producers of computer based systems are responsible and not allowed to permanently neglect the problem of system misbehaviors in practice.

1.1. Motivation and Outline. Our article addresses correct construction and functioning of large computer based systems. In view of so many annoying and dangerous system misbehaviors we ask (and positively answer) the question: Can informaticians be righteously accounted for system weaknesses, will they be able to justify systems to work correctly as intended, to be dependable, reliable, robust?

Since hardware turns out to be quite reliable, the question comes down to software, i.e. to abstract and mathematically treatable components of systems. The rigid nature of matter educates hardware technologists to be extremely sensitive towards hardware failures; they

are felt as sensations. So system faults are mostly and increasingly caused by software. This observation is crucial and matches the clear delimitation of responsibilities between hardware and system software engineering. Software engineers are permitted to assume hardware to work correctly, and to exploit this assumption for equally sensitive, rigorous low level implementation verification of software. Once software engineers and application programmers can count on (trust in) the correctness of their low level machine implementations and integrated systems, an equal status of sensitivity also for software faults becomes justifiable.

That is to say, at the low end, system software engineering meets hardware engineering, and at the upper end, compiler constructors and system software engineers meet application programmers and software engineers. They want to express their software in problem-oriented languages and rely on machine independent semantics. Compiler constructors cannot be made responsible for application program faults. Actually, a compiler has to be constructed without any knowledge about the intended meaning of application programs. The contract has to be with respect to program semantics. As a return, the application programmer expects correctly implemented machine executables. But both, application and system programmers have to be aware of what kinds of implementation correctnesses make sense and can realistically be guaranteed. We will reflect realistic requirements and introduce the notion of relative correctness and its preservation and variants in view of data and program representation and of errors which can be accepted or others which have to be avoided.

Although an area of research and development since 40 years, realistic language implementation, a central topic in system software engineering, is still a severe gap in trustworthiness. Practical compiler construction proceeds in three steps: (1) Specification of a compiling relation from source language to target machine code, (2) implementation of the compiling relation as a compiler program in an appropriate high level system programming language, and (3) bootstrapping the compiler, i.e. translation of the compiler source program into host machine code using an existing host compiler. Theoretical informatics, software engineering and compiler construction have importantly contributed towards correctness of the first two steps.

1.1.1. Trusted Compiler Implementation. But how to verify the third, the low level step? So far there are no realistic recipes to close this gap, although it is known for many years (at least since Ken Thompson's Turing Award lecture in 1984) that trust in executable code can dangerously be compromised by Trojan Horses in compiler executables, even if they pass strongest tests. Our article will show how to close this low level gap. The Deutsche Forschungsgemeinschaft (DFG) research group *Verifix* has developed the method of rigorous syntactic a-posteriori code inspection in order to remove every source of crucial faults in initial compilers for appropriate high level system programming languages. The method employs multi-pass translation with tightly neighboring intermediate languages and a diagonal bootstrapping technique which effectively is based on the above mentioned correctness assumptions and deliberations. A-posteriori result checking is applicable for the construction of verified compiler generators as well, and it is crucial to the development of strategies to substitute existing system software by proved correct modifications.

There is a current trend for system software to be required open source, enabling source code scrutiny for operating system components, networking software and also for compilers and other tools and utilities. This will definitely unveal a lot of bugs and even malicious code like Trojan Horses or so-called easter eggs. However, we want to stress, that the open source idea crucially depends on trusted compilation. Source level scrutiny does not sufficiently guarantee trustworthiness of executable soft-

ware [Thompson84, Goerigk99b, Goerigk00a, Goerigk00b]. There are sophisticated and intelligent techniques to completely hide Trojan Horses in binary compiler executables, which are not part of the alleged source code, but might cause unexpected, arbitrary, even catastrophic effects if target programs are eventually executed. No source code scrutiny, no source level verification, no compiler validation, virtually no test, not even the strong compiler bootstrap test does help. Note that in this situation it is also very unlikely that any of the known security techniques will help, because not even the application programmer can give a guarantee that her/his delivered application has not been compromised by auxiliary software utilities or compilers used during software construction.

Industrial software production for realistic safety and security critical applications enforces immense checking efforts. The amount of work is apparently unmanageable, and consequently, it is left undone. That is to say: On the one hand, we observe dangerous omissions in industrial practice. On the other hand, we realize how enormous the problems are. Thus, informatics science, in particular viewed at as an engineering science, has to attack them as problems of basic research — supported by research funding institutions like Deutsche Forschungsgemeinschaft (DFG) or German Federal Board of Safety and Security in Information Technology (Bundesamt für Sicherheit in der Informationstechnik, BSI), institutions which we find in all states with high science and technology standards. We are not allowed to leave industry people alone with their responsibility for necessary efforts which are seemingly unsurmountable at present. It is necessary to clearly identify the problems and to work towards methods for their rigorous solution which work out in practice. Mathematics as the classical structure science helps. Again and again, mathematicians invent ways of gaining and communicating insights, which are rigorous and convincing even though or maybe even because they are not completely formal. We shall see that in the central area of correct compiler construction and implementation our techniques of insight can cope with realistic software tasks, and of course then can serve as a model outside the area of compilers. Actually, it is the too often neglected low level implementation verification of so-called *initial* compilers which involves a certain amount of manual checking and proving [Goerigk/Langmaack01b]. For principle reasons we cannot leave all checking work to programmed computers if we do not want to run into circuli vitiosi. Trust in any executed program would furtheron dangerously and hopelessly depend on auxiliary software of uncertain pedigree [HSE01].

We do not want to blame or condemn anybody personally. The problem is enormous, very awkward, and in certain central areas it seems nearly unreasonable to ask for problem solutions in depth. However, because of potential disastrous consequences, informaticians must attack the problem and seek for solutions to give rigorous guarantees. And we shall see that in the crucial area of correct compiler construction, informatics science can achieve quite a lot. There is a remarkable interplay of informatics as a foundational structure science and as an engineering science.

The realistic recipe of bootstrapping and syntactical a-posteriori code inspection, which *Verifix* has developed in order to close the low level implementation gap rigorously, and which has been applied to a realistic compiler executable for a useful systems programming language, is the topic of the second article on *Trusted Compiler Implementation* [Goerigk/Langmaack01b].

1.2. Notions Mentioned in the Title. Let us briefly explain the notions which we mentioned in the title of this essay: *Informatics* (*Computer Science* in the Anglo-American literature) is the scientific discipline of design, construction and networking and of application and programming of computers (often called processors). Although not the most modern one, this definition is well stressing

the two important areas of work: hardware and software. *Computer based systems* (CBS) are engineering systems with embedded computers, programs, sensors, actuators, clocks and connecting channels in a physical environment [Schweizer/Voss96]. Realistic computer based systems use to be large, complex, and safety and/or security critical [Laprie94]. The latter means that systems may cause heavy damages to health and economy by unintended (internal) failures and by intended (external) attacks.

As a matter of fact, the bare existence of large computer based systems is justified. We need them. But what about correct construction and functioning of such systems? Can informaticians be made responsible, be accounted for, will they be able to justify systems to work correctly as intended? We use the term *justification* in this sense, i.e. for the design of computer based systems, the formulation of mathematical models of information flows in systems and the construction of software to be such that the expected system effects and the absence of failures and violations are foreseeable as logical consequences of the models alone. Not every system has such substantial delineable area which can be justified in this rigorous sense. In our opinion, however, if such a rigorous treatment is possible, it should be required for high safety and security standards. If we can logically foresee (infer) every desired property, especially safety and security properties, then we say that the computer based system has been (mathematically) *proved correct*, has been *verified*. We use the word *verification* in this sense, whereas *validation* means finding by experiments that a system fulfills our intents. Validation is not our main topic in this essay.

1.3. Consistent Checkability of Software Production Processes. Large computer based systems are essentially controlled by embedded processors and their software. General experience shows that the hardware processors actually work by far more reliably than software does and, hence, the weaknesses of computer based systems are in most cases due to software problems. Every software production (implementation) process today still has two major gaps in trustworthiness of consistent checkability, namely the transitions from

1. software design to high level source code (high level software engineering) and
2. from high level code to integrated executable binary machine code (realistic compilation).

Both gaps are under investigation in research and development since more than 30 years, but nevertheless even realistic compilation is still a severe gap in trustworthiness [BSI96]. Strictly speaking, realistic compilers are not correct and no compiler has yet sufficiently been verified. So the question arises whether informatics and in particular theoretical computer science, programming language theory, compiler construction and software engineering do not have any useful results to help in this situation. They have. And the insights are deep and also practical. But we have to admit that the results are often depending on too many complicated assumptions which the practical user has to check for in realistic situations. And unfortunately, in practice this so far requires many properly educated engineers and a lot of mathematical and logical skill.

If we seriously look at informatics also as an engineering science, we ought to demonstrate solutions and to show that the necessary checking can be done in a thorough and convincing way. In particular, in this essay we will demonstrate a strategic solution to close the second gap, that of correct realistic compilation, which in turn is necessary for a convincing high level software engineering.

1.4. DFG-Research Project Verifix-Correct Compilers. Correct realistic compilation is the major goal of the German joint project *Verifix* on *Correct Compilers* of the universities of Karlsruhe (G. Goos, W. Zimmermann), Kiel (W. Goerigk, H. Langmaack) and Ulm (A. Dold, F.W. von Henke). *Verifix* is a DFG-funded research group since 1996. The goal is to develop repeatable engineering

repeatable engineering methods for constructing correct compilers and compiler generators

- for realistic, industrially applicable imperative and object-oriented source programming languages and
- real world target and host processors
- which, by help of mechanical proof support (e.g. by PVS [Owre+92] or ACL2 [Kaufmann/Moore94]) and by exploiting the idea of a-posteriori result checking are rigorously verified as executable host machine programs and
- which nevertheless generate efficient code that compares to unverified compilers.
- Verifix uses practically and industrially approved compiler construction methods
- and the proof methodology supplements compiler construction, not vice versa.

Compiler construction is crucial to the construction of (large) computer based systems, and correct realistic compilers are necessary for a convincing construction process of correctly working systems. Systems consist of hardware and controlling software, and software splits in system and application software. Compiler programs are system software, and even compilers like any other piece of systems or applications are executable by compiling into binary processor code. There is no doubt that it is reasonable to require compiling to be correct.

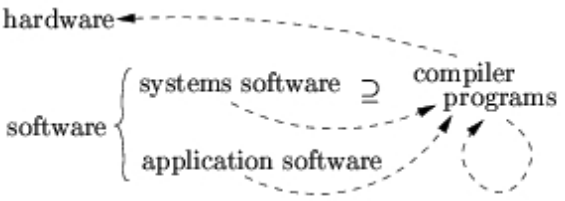


Fig 1. The central role of correct compilers

Sometimes we use the termini *fully correct* or *fully verified* if we want to emphasize that not only a mathematically specified compiling relation C^{SL}_{TL} between source language SL and target language TL is proved correct, but that also an executable compiler version implementing this relation and implemented in binary HML-machine code of a real world host processor HM is proved correct without depending on any auxiliary unchecked tool execution. Note that the term *fully correct* makes sense for arbitrary software as well, which is mathematically specified on the one hand, and implemented on a machine on the other.

We will make a difference between the terms *proved correct* and *provably correct*. We use *provably correct* in order to indicate that we are interested to develop methods which generate proof documents, often aided by computers. These documents can rigorously be checked to be proofs[1]. If the documents are checked, for instance for a concrete compiler, we use the term *proved correct*. Of course, we have to admit that checking might bear errors. So *proved* does not claim absolute truth. But it claims a rigorous attempt to gain mathematical certainty, which is much more than many other research areas can achieve.

It is not true, that investigation into correct realistic compilation does not pay off just because software design and high level software engineering probably show up many more faults than compilation. Unless we close the lower level gap with mathematical certainty, a major goal of *Verifix*, potential incorrect compilation will always critically disturb the recognition of certainties resp. uncertainties in high level software engineering. Correct realistic compilation establishes a trustworthy execution basis for further software development.

Informatics is well responsible for compilers with their program semantics and machines. Full verification of compilers is manageable and feasible. Compared to programming language theory and compiler construction practice, no other discipline of practical computer science is so well equipped with theory. Investigations in *Verifix* have brought up ideas and methods to incrementally replace compilers and to replace or encap-

[1] by more or less skilled informaticians. The required skill varies. Not every checking work requires trained mathematicians or logicians.

sulate system software components by fully verified software.

1.5. Levels of Trustworthiness and Quality. The lower level gap due to realistic compilation has been made public by governmental boards like the German BSI, but also in other countries. In 1989, BSI published IT-safety and security levels (of trustworthiness or IT-quality) [ZSI89]. In Germany, there are eight levels from Q0 (unsufficiently checked) up to Q7 (formally verified, proofs and implementation are performed by officially admitted tools).

However, an "officially admitted" tool (like a compiler or theorem prover) is not necessarily fully verified. So for instance a compiler just needs to pass an official validation test suite. It is well-known, that such tests do not suffice nor replace correctness proofs [Dijkstra76]. Official IT-certification prescriptions like those published by BSI in 1989/90/94 [ZSI89, ZSI90, BSI94] require:

> "The compilers employed must be officially validated and be admitted as implementation tools for Q7-systems by an official evaluation board."

The terminus *validated* reveals that for tools like compilers the evaluation boards at present can only validate for instance by applying official test suites. The boards do not see any other rigorous checking or proof technique and hence suspect that one is not allowed to trust in the correctness of generated machine code and hence of any compiler. Thus, consequently BSI added the following additional requirement:

> "The transformations from source to target code executed by a compiler program on a host machine must be a-posteriori checkable ("nachvollziehbar", "inspectable")."

For this reason certification authorities still do not trust in any existing realistic compiler. Instead, they often perform parallel semantical inspections of both high level code and low level machine code [Pofahl95], i.e. they check a-posteriori that the target code will work as expected from the source code.

Such a code-inspection is a rigorous a-posteriori checking of the target program π_{TL} to perform as expected from the source program π_{SL}, where the target program is the result of a successful compilation of the (well-formed) source program. The hope is that this is feasible if the mapping $\mathbf{C}^{SL}_{TL}$ (the specification of the transformation) from source to target programs is "inspectable", and hence that it is sufficient to check

$$(\pi_{SL}, \pi_{TL}) \in \mathbf{C}^{SL}_{TL}.$$

However, as long as $\mathbf{C}^{SL}_{TL}$ is not proved correct, the checking involves semantical considerations.

Inspection resp. a-posteriori result checking is an old idea [Blum+89]. And we know the method from school mathematics: since for instance integer division is felt more error prone than multiplication, we *double-check the results* of division by multiplication. We also use to double-check the results of linear equations solutions by simpler matrix-vector-multiplication. By *code inspection* with respect to compilers we mean the a-posteriori result checking of compiler generated code. Result checking is often much easier than (a-posteriori) verification. Moreover, it is an interesting observation, that industrial software engineers and certification boards trust the technique of a-posteriori result checking. Although there is a well-established and reasonably developed program verification theory, it is often not well-applicable to large systems or even large amounts of low level code. Therefore it is so interesting to observe, both from a theoretical and from a practical point of view, that realistic scientifically founded fully verified compiler construction has to reach back also to a-posteriori checking to a small, but decisive extent.

We will see this later while proving low level compiler implementation correctness [Goerigk/Langmaack01b] and hence full compiler correctness. Consequently, since this is possible and feasible, already in 1989/90 one of the authors proposed to introduce an even higher IT-safety and security level of quality Q8

(Production, proof and checking tools are fully verified, not only at high level, but also down to implementations in executable binary machine code). Otherwise, the low level gap of realistic compilation will remain open forever.

2. Code Inspection in Compiler Construction Processes

The *Verifix*-thesis is that after 40 years compiler construction and more than 30 years software engineering it should no longer be necessary to inspect low level generated code, not even for safety and security critical software. The new higher quality level Q8 should be introduced which is reaching beyond Q7. It should be desirable and required for industrial software construction, in particular for realistic compilers. We cannot entirely avoid manual low level code inspection [Fagan86], however, it is only necessary for initial compiler executables which cannot be implemented on behalf of a verified bootstrapping compiler executable. The requirement for low level code inspection only make sense in this respect. In principle, it should suffice to perform this work exactly once. However, realistic industrial correct compiler engineering additionally needs repeatable methods for constructing correct initial compiler implementations from the scratch whenever necessary.

The goal of correct realistic compiler construction in the view of *Verifix* is that compiler executables on real world host machines have to be provably correct, and if they are to be used for safety and security critical software implementation, they have to be proved correct. That means, that any executable binary machine program successfully generated from a well-formed source program is provably or proved correct with respect to the source program semantics. Machine program correctness may only depend on

- the correctness of source level application programs with respect to their specifications,
- hardware, i.e. target and host processors to work correctly as described in their instruction manuals,
- correct rigorous (mathematical, logical) reasoning

Application programmers are responsible for the first assumption and hardware designers for the second. Hence, the compiler constructor only needs consider the semantics of well-formed source programs and processors and needs not respect any further intention of system or application programmers.

Since correctness of a compiler is defined by correctness of ist resulting target programs, we only depend on this property which equally well can be established by a-posteriori result checking. In that case we sometimes use the term *verifying compiler*. Suppose we have a given unverified compiler τ_{HL} from SL to TL written (or generated) in some high level host programming language HL. Suppose that τ_{HL} compiles via intermediate languages

$$SL = IL_0, IL_1, ..., IL_m = TL, m \geq 1,$$

by compiler passes

$$\tau^1_{HL}; \tau^2_{HL}; ...; \tau^m_{HL} = \tau_{HL}$$

which for a well-formed source program π_{SL} may successfully generate intermediate programs

$$\pi_{SL} = \pi_{IL0}, \pi_{IL1}, ..., \pi_{ILm} = \pi_{TL}.$$

For all passes τ^i_{HL} we assume that we are able to write a-posteriori code inspection procedures γ^i_{HL} which we insert into τ_{HL}:

$$\tau^1_{HL}; \gamma^1_{HL}; \tau^2_{HL}; \gamma^2_{HL}; ...; \tau^m_{HL}; \gamma^m_{HL} = \tau'_{HL}.$$

Syntax and static semantics of τ_{HL} has to make sure that the unverified passes τ^i_{HL} are safely encapsulated such that their semantics cannot interfere with the inspection procedures nor with any other passes. This method is obviously relying on the existence of an initial correct and correctly implemented compiler from HL to the compiler host machine code, because it is by no means realistic to assume HL to be a binary machine language.

In practice, there is no way around an initial correct compiler executable on some host machine. Moreover, HL should at least be usable, if not comfortably usable, for systems programming and writing compilers.

Since programming languages, their semantics, hardware processors and their binary machine codes can acceptedly be modeled using mathematics, correct compiler specification and implementation are in principle mathematical tasks. Perhaps they are not that deep, however, they require an exorbitant organization which is to be convincing and without any logical gap.

Hardware engineering has, for good money reasons, developed an error handling culture by far better than software engineering [Goerke96]. Hardware errors are sensations, whereas software errors are commonplace. Since hardware errors cannot easily be repaired, they bear the risk of high economic damage. Thus, hardware engineers seek even for small gaps where faults could creep in. Seemingly, software errors can easily be repaired. But often bug fixes make things even worse, and also software errors bear a high risk.

If we assume hardware correct, then we can rely on hardware processors programmed by correct and correctly implemented software. However, if a processor is endowed with verified high level programs, but additionally with some non-verified compilers, then it generally will not work failure-free for the time being. But the goal of fully reliable systems software is not hopeless. We will come back to this.

3. Related Work on Compiler Verification

Let us ask if literature does help in order to prove the three compiler implementation steps correct. Actually, we find intensive work on steps 1 and 2, although often under unrealistic assumptions so that the results have to be handled with care. Step 3 has nearly totally been neglected. If the phrase ``compiler verification´´ is used, then most of the authors mean step 1. There is virtually no work on full compiler verification. Therefore, the ProCoS project (1989−1995) has made a clear distinction between *compiling verification* (step 1) and *compiler implementation verification* (steps 2 and 3).

Compiling verification is part of theoretical informatics and program semantics and work on it has been started by J. McCarthy and J.A. Painter in 1967 [McCarthy/Painter67]. Proof styles are operational [McCarthy/Painter67, Boerger/Rosenzweig92, Boerger/Schulte98] or denotational [Milne/Strachey76] depending on how source language semantics is defined. If a source language has loops or (function) procedures, then term rewriting or copy rule semantics is employed throughout [Langmaack73, Loeckx/Siebe87]. Other operational styles split in *natural* [Nielson/Nielson92] or *structural* [Plotkin81] operational or *state-machine-like* [Gurevich91, Gurevich95]. Denotational semantics has started with D. Scott's work [Scott70, Scott/Strachey71], and typical compiling correctness proofs can be found in [Milne/Strachey76]. The authors in [Hoare+93, Sampaio93, Mueller-Olm96, Mueller-Olm/Wolf00] use an algebraic denotational style for clearer modular proofs, based on state transformations resp. predicate transformers.

Mechanical proofs are often based on interpreter semantics, a further variant of the operational style [Stoy77], and sometimes include high level compiler implementation verification (step 2) with e.g. HL = Stanford-Pascal [Polak81] or Boyer/Moore-Lisp [Moore88, Flatau92, Moore96] or Standard-ML [Curzon93a, Curzon94]. M. Broy [Broy92] uses the Larch-prover [Garland/Guttag91]. One should keep in mind, however, that the running theorem prover implementations are, strictly speaking, not completely verified. Their correctnesses again depend on existing correct initial host compilers, which are not available up to now.

Recalling section 1, hackers might have intruded Trojan Horses [Thompson84, Goerigk99b] via unverified start up compilers. Hence, we are left on human checkability of mechanical proof protocols (a-posteriori-proof checking).

High level compiler implementation verification (step 2) is a field within software engineering. Correct implementation rules have been worked out in many formal software engineering methods and projects like VDM [Jones90], RAISE [George+92], CIP [Bauer78, Partsch90], PROSPECTRA [Hoffmann/-Krieg-Brueckner93], Z [Spivey92], B [Abrial+91], and also the PVS-system [Dold00].

Literature on low and machine level compiler implementation verification (step 3) is by far too sparse. There are only demands by some researchers like Ken Thompson [Thompson84], L.M. Chirica and D.F. Martin [Chirica/Martin86] and J S. Moore [Moore88, Moore96], no convincing realistic methods. Here is the most serious logical gap in full compiler verification. Unfortunately, a majority of software users and engineers — as opposed to mathematicians and hardware engineers — do not feel logical gaps to be relevant unless they have been confronted with a counter-example. So we need

A. convincing realistic methods for low level implementation verification (step 3)

B. striking counter-examples (failures) in case only step 3 has been left out.

Let us first step into B and hence go on with an initial discurs on the potential risk of omitting the low level machine code verification step for compilers:

It is possible to construct a malicious compiler executable which correctly compiles any but exactly two source programs [Goerigk99b, Thompson84]. One exception can be chosen arbitrarily, and the compiler can be constructed to generate arbitrary, perhaps catastrophic target code — for instance a security relevant back door in the Unix login procedure. The second exception is the compiler's own source code. The executable will reproduce itself if applied to this source code. If such a compiler gets used by accident as the pre-initial auxiliary tool for compiler implementation, without checking its result, it will produce an executable which still behaves maliciously, generates incorrect code for two programs, and one of them might eventually cause a catastrophe. It is very unlikely to find this program and thus such a hidden Trojan Horse by classical testing or compiler validation. Even Wirth's bootstrap test is passed, because the incorrect executable is constructed to reproduce itself in that case, and of course arbitrarily often, no matter if the compiler has been verified on source level or not.

Additional low-level implementation verification guarantees correctness and the absence of such malicious behavior. The initial compiler executable then provides a *trusted execution basis* necessary in particular also for achieving security. This is the topic of the companion paper [Goerigk/Langmaack01b].

4. Towards Trusted Realistic Compilation

Experience in realistic compiler construction shows that in order to construct correct compilers we are allowed to restrict ourselves to sequential imperative programming languages. For compilation we need no process programming nor any reactive or real-time behavior; compilers are sequential (and transformational) and we only depend on the correctness of their resulting target programs. This insight crucially facilitates the foundational investigation in full compiler correctness proofs including the essential (and so far missing) implementation correctness proofs for executable compiler host machine code. Implementations of realistic compilers are constructed

- in sequential, mostly in imperative languages,
- even if concurrent process or real-time languages are to be compiled.

Consequently, we will study correct realistic compilation for sequential imperative programming languages, and in particular we restrict ourselves to *transformational* programs (cf. Section 5), because we are mainly interested in the input/output relations defined by program semantics. We need full compiler correctness proofs only

• for one initial compiler per processor family with identified target and host machine language TML = HML

• and for one sufficiently high-level source language SL which allows for formulating compilers and system programs.

All further compilers, e.g. optimizing compilers or those for more expressive system programming languages SL_1, compiler generators, any further systems software like provers, proof assistants and proof checkers need no further correct implementation steps below SL or SL_1. In particular, no further machine code inspection is necessary, and executable binary versions of these programs can be generated purely mechanically following N. Wirth [Wirth77]. The resulting code is proved to be correct due to a bootstrapping theorem [Goerigk/Langmaack01b], also found in [Langmaack97a, Goerigk99a, Goerigk/Langmaack01, Goerigk-/Langmaack01a]. As a conclusion we want to stress, that the second software production process gap (cf. [Goerigk/-Langmaack01b]) can be closed if the *Verifix*-recipes of correct initial compilers and a-posteriori program-checking [Goerigk/Langmaack01b] are obeyed.

In the context of the *Verifix*-project we will furthermore demonstrate how to develop correct compiler generating tools and how to incorporate even unverified existing tools in a fully trusted and rigorously proved correct manner [Heberle+98, Gaul+99, Gaul+96]. Moreover, the specification and verification system PVS [Owre+92] is used for mechanical proof support, in particular for the formalization and verification of the compiling specifications, and for providing support for high level compiler implementation verification using a transformational approach.

In the following we want to focus on how to develop a proved correct and hence trustworthy *initial* compiler implementation which we believe is the foundational basis for the practical construction of safe and secure, i.e. trustworthy executable software.

4.1. Three Steps towards Fully Correct Compilers. In his work on Piton's verified translation [Moore88, Moore96] J S. Moore recommends three steps in order to present a convincing correctness proof of a compiler written (or implemented) in executable binary host machine code HML:

1. Specification of a compiling relation **C** between abstract source and target languages SL and TL, and compiling (specification) verification w.r.t. an appropriate semantics relation $\sqsubseteq$ between language semantics $[[\cdot]]_{SL}$, $[[\cdot]]_{TL}$.

2. Implementation of a corresponding compiler program τ_{HL} in a high level host language HL close to the specification language, and high level compiler implementation verification (w.r.t. **C** and to program representations $\varphi^{SL}_{SL'}$ and $\varphi^{TL}_{TL'}$).

3. Implementation of a corresponding compiler executable τ_{HML} written in binary host machine language HML, and low level compiler implementation verification (with respect to $[[\tau_{HL}]]_{HL}$ and program representations $\varphi^{SL'}_{SL''}$ and $\varphi^{TL'}_{TL''}$).

If we work through every step, then τ_{HL} resp. τ_{HML} is a correct or verified compiler (implementation). If we want to stress that a correctness proof has been achieved even for a compiler executable like τ_{HML} on a real processor, then we sometimes call it *fully correct* (verified) as informally explained before. Figure 2 informally outlines the three essential steps that we have to work through for the construction and verification of fully correct compilers, and in particular of fully correct initial compiler executables. We will make everything shown in this diagram precise in the following sections 5 and 6, and in particular we refer to section 6.3.1, which discusses Figure 2 precisely.

5. Transformational Programs

We model the semantics of transformational programs by partial relations (or multivalued partial functions) *f* between input domains D_i and (not neces-

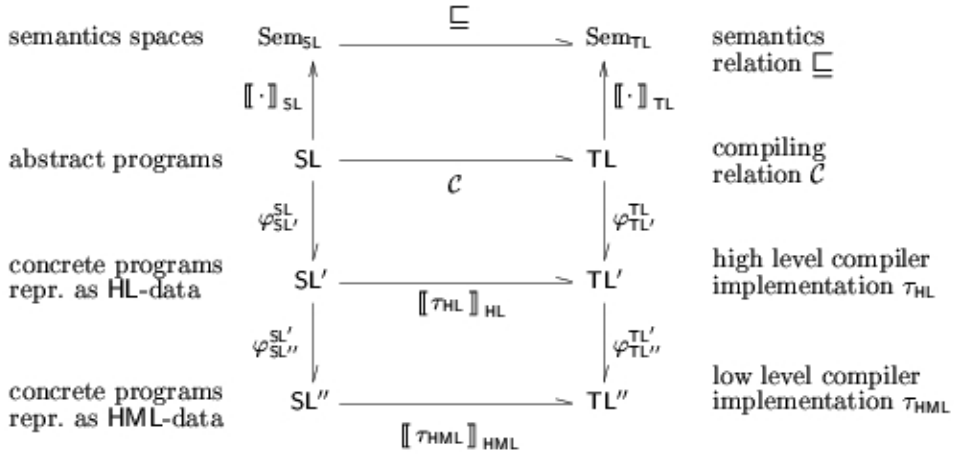


Fig 2. Three steps for correct compiler implementation τ_{HML}

sarily different) output domains D_o. Thus, a program semantics is a subset

$$f \subseteq D_i \times D_o$$

which we often will write as $f \in D_i \rightharpoonup D_o$ ($= \mathbf{2}^{D_i \times D_o}$) or as $D_i \xrightarrow{f} D_o$; sometimes we also use $f : D_i \rightharpoonup D_o$. In case f is single-valued, we may also use $\rightarrow$ instead of $\rightharpoonup$. We use ";" to denote the well-known semantical relational composition, i.e. if f_1: $D_1 \rightharpoonup D_2$ and f_2: $D_3 \rightharpoonup D_4$, then $f_1 \,;\, f_2 =_{def} =_{def} \{ (d_1,d_4) \mid \exists\, d_2 \in D_2 \cap D_3$ s.t. $(d_1,d_2) \in f_1$ and $(d_2,d_4) \in f_2 \} \in D_1 \rightharpoonup D_4$. Data or elements are often program or memory states or pairs of program and memory states (sometimes called configurations with control and data flow components) which we simply call *states* as well.

Data in D_i and D_o are considered *regular* or *non-erroneous*. In order to handle irregular data, i.e. finite and infinite errors, we assume that every data domain D is *extended* by an individually associated disjoint non-empty error set Ω, i.e.

$$D^{\Omega} =_{def} D \uplus \Omega$$

and $D \cap \Omega = \varnothing$. For every transformational program semantics f we assume an *extended* version

$$f \in D_i^{\Omega} \rightharpoonup D_o^{\Omega}$$

which we denote by the same symbol f unless this will cause ambiguities. We have $D_i^{\Omega} = D_i \uplus \Omega_i$ and $D_o^{\Omega} = D_o \uplus \Omega_o$.

Errors are final. No computation will ever recover from an error[2]. Thus, we require (the extended) f to be *error strict*, i.e. f to be total on Ω_i and $f(\Omega_i) \subseteq \Omega_o$. However, we have to respect a further phenomenon. Errors are of essentially different types. They are either unavoidable and we have to accept them, like for instance machine resource violations, or they are unacceptable and thus to be avoided. We will allow unacceptable errors to cause unpredictable (or chaotic) consequences. In order to model this phenomenon, we partition Ω in a non-empty set $A \subseteq \Omega$ of *acceptable* and a non-empty set $U =_{def} \Omega \setminus A$ of *unacceptable* or *chaotic* errors. So we require

$$\Omega_i = A_i \uplus U_i \text{ and } \Omega_o = A_o \uplus U_o$$

and a *strong* error strictness, namely that f is total on Ω_i (and thus on A_i and U_i) and

$$f(A_i) \subseteq A_o \text{ and } f(U_i) \subseteq U_o .$$

The error sets Ω are supposed to contain a particular standard error element $\perp$ which is to model infinite computation (divergence). So in particular we have $\perp_o \in \Omega_o$, and for extended program semantics f we additionally require $(d, \perp_o) \in f$ or equivalently $\perp_o \in f(d)$ whenever there is a non-terminating (infinite) computation of f starting with $d \in D_i^{\Omega}$.

[2] Note that exceptions are not errors in our sense. We think of exceptions as special cases of non-local regular control flow.

Thus, we model transformational program semantics by strongly error strict extended relations between extended input and output domains, and we additionally require them to meet the above condition for infinite computations.

5.1. Correct Implementation. Let f_s be a source and f_t a target program semantics. In the following we will explain when f_t correctly implements f_s. This requires data representation relations $\rho_i \in D_i^{s\,\Omega} \rightharpoonup D_i^{t\,\Omega}$ and $\rho_o \in D_o^{s\,\Omega} \rightharpoonup D_o^{t\,\Omega}$ between source and target input and output data. Both relations, ρ_i and ρ_o, and their inverses ρ_i^{-1} and ρ_o^{-1}, have to be strongly error strict (we sometimes call such ρ's *strongly error strict in both directions*).

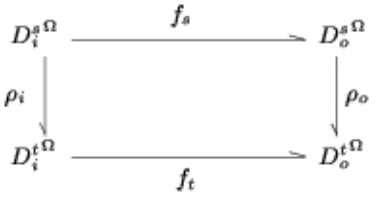


Fig 3. Source and target program semantics f_s, f_t and data representations ρ_i, ρ_o

Definition 5.1. (Correct implementation or refinement) f_t is said to be a *correct implementation* or *refinement* of f_s *relative* ρ_i and ρ_o and associated error sets, iff

$$(\rho_i ; f_t)(d) \subseteq (f_s ; \rho_o)(d) \cup A_o^t$$

holds for all $d \in D_i^s$ where $f_s(d) \subseteq D_o^s \cup A_o^s$ (which is equivalent to $(f_s ; \rho_o)(d) \subseteq D_o^t \cup \cup A_o^t$, because ρ_o is strongly error strict in both directions).

For any target program computation, the outcome d'' in $D_o^{t\,\Omega}$ is either an acceptable error output in A_o^t, or there exists a source program computation that either ends in a (regular) d' corresponding to d'' or with an unacceptable (chaotic) error output in U_o^s.

That is to say: If f_t is a correct implementation of f_s, then f_t either returns a correct result, or an acceptable error, or, if f_s can compute an unacceptable error, f_t may (chaotically) return any result, because we do not require anything in that case.

If f_t correctly implements f_s relative ρ_i and ρ_o, we will write $\rho_i ; f_t \sqsupseteq f_s ; \rho_o$ or even shorter just $f_t \sqsupseteq f_s$ (with a boldface $\sqsupseteq$). We indicate this diagram commutativity by the diagram in Figure 4 and we will later see that we can compose correct implementation diagrams both vertically and horizontally, which is a very important fact for practical software engineering and compiler generation.

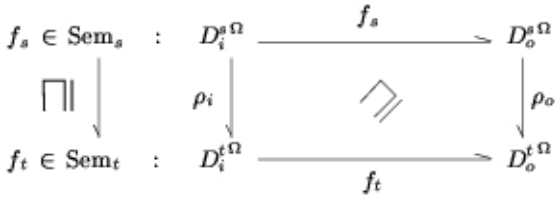


Fig 4: Commutative diagram expressing correct implementation

There are some variations of *correct (relative) implementation* as of Definition 1, which we would like to discuss: We speak of *correct acceptable implementation* resp. of *correct regular implementation* or *refinement,* iff

$$\varnothing \neq (\rho_i ; f_t)(d) \subseteq (f_s ; \rho_o)(d) \cup A_o^t \quad \textit{acceptable}$$

$$\varnothing \neq (\rho_i ; f_t)(d) \subseteq (f_s ; \rho_o)(d) \quad \textit{regular}$$

holds for all $d \in D_i^s$ where $\varnothing \neq f_s(d) \subseteq D_o^s \cup \cup A_o^s$ resp. $\varnothing \neq f_s(d) \subseteq D_o^s$. But note that — in contrast to the different program correctness notions (cf. Section 5.1.1) — all three implementation correctness notions are independent; neither one implies the other. It is remarkable, however, that concrete correctness proofs turn out to be of less complexity if $\perp$ is supposed to be unacceptable, i.e. if we prove variants of correct regular implementation [Mueller-Olm/Wolf00, Wolf00]. If $\perp$ is supposed acceptable, then our experience shows, that we have to characterize greatest fixed points exactly and to additionally use computational or fixed point induction in order to prove variants of correct acceptable implementation, e.g. preservation of partial correctness [Goerigk00a, Goerigk00b].

5.1.1. Preservation of Relative Correctness. It is an important observa-

tion, that we can exactly characterize correct implementation by *preservation of relative correctness* [Wolf00], which generalizes Floyd's and Hoare's notions of partial respectively total program correctness. Let $f \in D_i^{\Omega} \rightharpoonup D_o^{\Omega}$ be a program semantics and let $\Phi \subseteq D_i$ and $\Psi \subseteq D_o$ be predicates, i.e. subsets of regular data.

Definition 5.2. (Relative program correctness) f is called *(relatively) correct with respect to* (pre- and post-conditions) Φ and Ψ ($<\Phi> f <\Psi>$ for short), iff

$$f(\Phi) \subseteq \Psi \cup A_o$$

If there is no ambiguity, and if the implicit parameters are clear from the context, we will sometimes leave out the word *relative* and simply speak of *program correctness*. The following theorem says that f_t correctly implements f_s if and only if relative correctness of f_s implies relative correctness of f_t for all pre- and post-conditions Φ and Ψ.

Theorem 5.1. (Preservation of relative correctness) f_t correctly implements f_s $(\rho_i ; f_t \sqsupseteq f_s ; \rho_o)$ if and only if, for all $\Phi \subseteq D_i^s$ and $\Psi \subseteq D_o^s$,

$$<\Phi> f_s <\Psi> \Rightarrow < \rho_i(\Phi) > f_t < \rho_o(\Psi) >.$$

Proof. *Only if:* Let $f_s(d) \subseteq D_o^s \cup A_o^s$ imply $(\rho_i ; f_t)(d) \subseteq (f_s ; \rho_o)(d) \cup A_o^t$ for all $d \in D_i^s$ and let $f_s(\Phi) \subseteq \Psi \cup A_o^s$. Claim: $f_t(\rho_i(\Phi)) \subseteq \subseteq \rho_o(\Psi) \cup A_o^t$. In order to show this, let $d \in \Phi$. Then $f_s(d) \subseteq \Psi \cup A_o^s \subseteq D_o^s \cup A_o^s$. Hence,

$$\begin{aligned} f_t(\rho_i(d)) &\subseteq \rho_o(f_s(d)) \cup A_o^t \\ &\subseteq \rho_o(\Psi \cup A_o^s) \cup A_o^t \\ &= \rho_o(\Psi) \cup A_o^t . \end{aligned}$$

If: Let $f_s(\Phi) \subseteq \Psi \cup A_o^s$ imply $f_t(\rho_i(\Phi)) \subseteq \rho_o(\Psi) \cup A_o^t$ for all $\Phi \subseteq D_i^s$ and $\Psi \subseteq D_o^s$, and let $f_s(d) \subseteq D_o^s \cup A_o^s$. Claim: $(\rho_i ; f_t)(d) \subseteq (f_s ; \rho_o)(d) \cup A_o^t$. For this, let $\Phi =_{def} \{d\}$ and $\Psi =_{def} f_s(d) \cap D_o^s$. Then $f_s(\Phi) = f_s(d) = (f_s(d) \cap D_o^s) \cup (f_s(d) \cap A_o^s) \subseteq \subseteq \Psi \cup A_o^s$. So

$$\begin{aligned} (\rho_i ; f_t)(d) &\subseteq \rho_o(f_s(d) \cap D_o^s) \cup A_o^t \subseteq \\ &\subseteq (f_s ; \rho_o)(d) \cup A_o^t . \end{aligned}$$

□

Note that this theorem remains valid even if we relax strong error strictness; we actually need not require ρ_i and ρ_o and their inverses to be total on error sets. We only need that they respect the partition in acceptable and unacceptable errors. However, we still prefer data representation relations to be total on error sets (in both directions).

Again, we may discuss variations of the notion of relative program correctness, i.e. *acceptable* program correctness resp. *regular* program correctness with respect to pre-conditions $\Phi \subseteq D_i$ and post-conditions $\Psi \subseteq D_o$:

$<\Phi> f <\Psi>_{acc}$ iff $\varnothing \neq f(d) \subseteq \Psi \cup A_o$ (*acceptable*)
$<\Phi> f <\Psi>_{reg}$ iff $\varnothing \neq f(d) \subseteq \Psi$ (*regular*)

respectively hold for all $d \in \Phi$. Note that regular program correctness implies acceptable program correctness, which implies relative program correctness. Furthermore, correct acceptable resp. regular implementation is equivalent to preservation of acceptable resp. regular program correctness, i.e. f_t correctly acceptably resp. regularly implements f_s if and only if

$<\Phi> f_s <\Psi>_{acc} \Rightarrow <\rho_i(\Phi)> f_t <\rho_o(\Psi)>_{acc}$ (*acceptable*)
$<\Phi> f <\Psi>_{reg} \Rightarrow <\rho_i(\Phi)> f_t <\rho_o(\Psi)>_{reg}$ (*regular*)

respectively holds for all $\Phi \subseteq D_i^s$ and $\Psi \subseteq D_o^s$.

5.1.2. Classical Notions of Correct Implementation. In which sense does relative or acceptable or regular program correctness generalize the classical notions of partial or total program correctness? Let f be an original, i.e. unextended program semantics

$$f \in D_i \rightharpoonup D_o$$

and let $\Phi \subseteq D_i$ and $\Psi \subseteq D_o$ be pre- and post-conditions, respectively. f is called *partially correct w.r.t.* Φ and Ψ ($\{\Phi\} f \{\Psi\}$ for short), if $f(\Phi) \subseteq \Psi$. f is called *totally correct w.r.t.* Φ and Ψ ($[\Phi] f [\Psi]$ for short), if f is partially correct w.r.t. Φ and Ψ, i.e. $f(\Phi) \subseteq \Psi$, and additionally the domain dom_f of f comprises Φ, i.e. if additionally $dom_f \supseteq \Phi$.

Let us now choose the same particular error sets $A = A_i = A_o =_{def} \{a\}$ and $U = U_i = U_o =_{def} \{u\}$ for both domains D_i and D_o with $\perp \in \{a,u\}$ and $\Omega = \Omega_i = \Omega_o =_{def} =_{def} A \uplus U$, and extend f to $f^{ext} \in D_i^{\Omega} \rightarrow D_o^{\Omega}$ by

$$f^{ext} =_{def} f \cup ((D_i \setminus \mathrm{dom}\, f) \times \{\perp\}) \cup id_{\Omega}$$

f^{ext} is strongly error strict, regardless of $\perp$ being considered acceptable ($A_i = A_o =_{def} =_{def} \{\perp\}$, $U_i = U_o =_{def} \{u\}$) or unacceptable ($A_i = A_o =_{def} \{a\}$, $U_i = U_o =_{def} \{\perp\}$). $\mathrm{dom}(f) =_{def} =_{def} \mathrm{dom}\, f$ is the domain of f.

Partial and Total Program Correctness. Relative and acceptable correctness are equivalent to partial correctness

$$<\Phi> f^{ext} <\Psi> \Leftrightarrow <\Phi> f^{ext} <\Psi>_{acc} \Leftrightarrow \{\Phi\} f \{\Psi\}$$

if $\perp = a$ is considered acceptable, and relative, acceptable and regular correctness are equivalent to total correctness, if $\perp = u$ is considered unacceptable:

$$<\Phi> f^{ext} <\Psi> \Leftrightarrow <\Phi> f^{ext} <\Psi>_{acc} \Leftrightarrow$$
$$\Leftrightarrow <\Phi> f^{ext} <\Psi>_{reg} \Leftrightarrow [\Phi] f [\Psi]$$

Preservation of Partial and Total Program Correctness. It is not only that relative program correctness generalizes classical partial and total program correctness. Our notion of correct (relative) implementation also generalizes well-known implementation correctness notions.

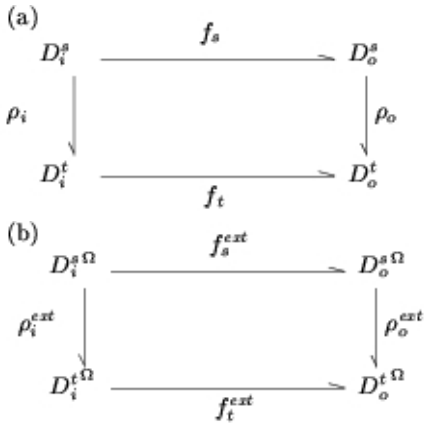


Fig 5: Classical correct implementation versus preservation of relative correctness.

Consider a classical unextended implementation diagram (a). If we extend f_s and f_t as explained before, and if we extend the data representation relations ρ_i and ρ_o by $\rho_i^{ext} =_{def} =_{def} \rho_i \cup id_{\Omega}$ respectively $\rho_o^{ext} =_{def} \rho_o \cup id_{\Omega}$ as well, then we get the extended diagram (b).

If we consider $\perp$ acceptable, then

$$\rho_i^{ext};\ f_t^{ext} \sqsupseteq f_s^{ext};\ \rho_o^{ext} \Leftrightarrow \rho_i;\ f_t \subseteq f_s;\rho_o$$

exactly expresses preservation of partial program correctness. On the other hand, if we consider $\perp$ unacceptable, then

$$\rho_i^{ext};f_t^{ext} \sqsupseteq f_s^{ext};\rho_o^{ext} \Leftrightarrow$$
$$\Leftrightarrow (\rho_i;f_t) \mid \mathrm{dom}(f_s;\rho_o) \subseteq f_s;\rho_o$$
$$\text{and} \quad \mathrm{dom}\,\rho_i;f_t \supseteq \mathrm{dom}\, f_s;\rho_o$$

expresses exactly the classical preservation of total correctness.

Hence, it is justified to transfer the terms *total* and *partial* to extended functions or relations f^{ext}, and we may use the words ``correct total (resp. partial) implementation'' instead of "correct regular (resp. relative) implementation". Also, we may replace "regular (resp. relative) program correctness" again by "total (resp. partial) program correctness".

The classical software engineering notion of correct implementation as propagated in many (also formal) software engineering approaches like for instance in VDM (Vienna Development Method) is preservation of total program correctness. We should, however, keep in mind that resources might well exhaust while machine programs are executed on real target machines, so that a compiled program semantics f_t can in general not be proved to meet the requirements of correct implementation in the latter sense. Preservation of relative correctness gives us the necessary means to define adequate notions of correct implementation also for realistic correct compilation.

5.2. Composability. We mentioned that composability (transitivity) of correct implementation is crucial for modular software construction and verification, and in particular for stepwise compilation and compiler construction and implementation. In the following we prove vertical and horizontal transitivity, i.e. that we may (vertically) decompose correct implementations into steps (or phases), and that correct implementation distributes (horizontally) over sequential (relational) composition. In both cases we

will prove a more generally applicable transitivity result as well.

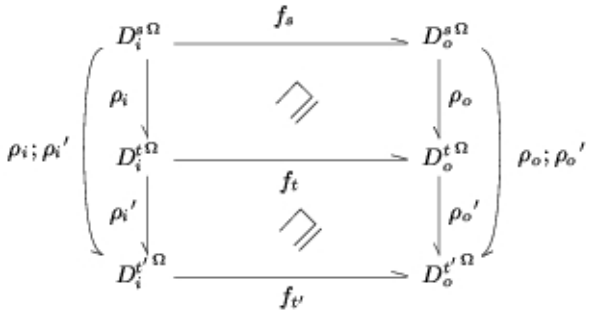


Fig 6: Vertical composition expressed by commutative diagrams: If the inner diagrams are commutative, then so is the outer one.

Theorem 5.2. (Vertical transitivity of correct implementation) If f_t correctly implements f_s, and if $f_{t'}$ does so for f_t , then $f_{t'}$ correctly implements f_s.

Proof. Let $\Phi \subseteq D_i^s$, $\Psi \subseteq D_o^s$ and let $<\Phi>\ f_s\ <\Psi>$. Then, due to Theorem 5.1, we have $<\rho_i(\Phi)>\ f_t\ <\rho_o(\Psi)>$ (commutativity of the upper diagram) and $<\rho_i'\ (\rho_i(\Phi))>\ f_{t'}\ <\rho_o'\ (\rho_o(\Psi))>$ (commutativity of the lower diagram for pre-condition $\rho_i(\Phi)$ and post-condition $\rho_o(\Psi)$). But the latter just means

$$<(\rho_i\,;\rho_i')\ (\Phi)>\ f_{t'}\ <(\rho_o\,;\rho_o')\ (\Psi)>$$

which completes the proof due to equivalence of correct implementation and preservation of (relative) correctness (Theorem 5.1). □

Theorem 5.3 (Horizontal transitivity of correct implementation) If f_t correctly implements f_s, and if $f_{t'}$ does so for $f_{s'}$, then $f_t\,;f_{t'}$ correctly implements $f_s\,;f_{s'}$.

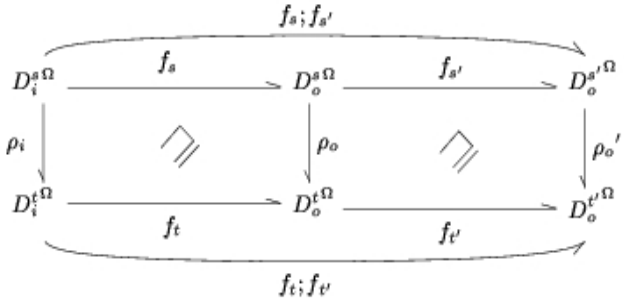


Fig 7: Horizontal composition expressed by commutative diagrams: If the inner diagrams are commutative, then so is the outer one.

Proof. Let

(1) $\rho_i\,;f_t \sqsupseteq f_s\,;\rho_o$ and

(2) $\rho_o\,;f_t \sqsupseteq f_s\,;\rho_o'$ and

(3) $(f_s\,;f_{s'})\ (d) \subseteq D_o^{s'} \cup A_o^{s'}$

Claim: $(\rho_i\,;f_t\,;f_{t'})(d) \subseteq (f_s\,;f_{s'};\rho_o')(d) \cup \cup A_o^{t'}$. Note that $(f\,;g)(d) = g(f(d))$ and that ";" is associative and monotonic in both arguments. By (3) we have $f_{s'}(f_s(d)) \subseteq D_o^{s'} \cup \cup A_o^{s'}$ and by ((2), commutativity of the right diagram) and associativity of ";" we get $(\rho_o;\ f_{t'})(f_s(d)) \subseteq (f_{s'}\,;\ \rho_o')\ (f_s(d)) \cup A_o^{t'}$ and hence

(4) $f_{t'}((f_s\ ;\ \rho_o)(d)) \subseteq (f_s\ ;\ f_s\ ;\ \rho_o')(d) \cup A_o^{t'}$ and also

(5) $f_{t'}((f_s\,;\rho_o)(d) \cup A_o^t) \subseteq (f_s\,;f_{s'};\rho_o')(d) \cup A_o^{t'}$.

The latter holds by strong error strictness of $f_{t'}$, i.e. $f_{t'}(A_o^t) \subseteq A_o^{t'}$, from (4). By strong error strictness of $f_{s'}$ and (3) we have $f_s(d) \subseteq D_o^s \cup A_o^s$. Thus, by ((1), commutativity of the left diagram) we get

(6) $(\rho_i\,;f_t)\ (d) \subseteq (f_s\,;\rho_o)(d) \cup A_o^t$

and finally, by monotonicity of relation composition, (6) and (5) we get

(7) $f_{t'}((\rho_i\,;f_t)\ (d)) \subseteq (f_s\,;f_{s'};\rho_o')(d) \cup A_o^{t'}$

which is nothing but our claim. An alternative equivalent calculation would be

$$\begin{aligned}(\rho_i\,;f_t\,;f_{t'})(d) &= f_{t'}((\rho_i\,;f_t)(d)) \subseteq f_{t'}((f_s\,;\rho_o)(d) \cup \\ &\cup A_o^t) \subseteq f_{t'}((f_s\,;\rho_o)(d)) \cup A_o^{t'} = \\ &= (\rho_o\,;f_{t'})(f_s\,(d)) \cup A_o^{t'} \subseteq ((f_{s'};\rho_o')(f_s\,(d)) \cup \\ &\cup A_o^{t'}) \cup A_o^{t'} = (f_s\,;f_{s'};\rho_o')(d) \cup A_o^{t'}.\end{aligned}$$ □

Unfortunately, we have to prove horizontal transitivity in a different style. A proof as elegant as for vertical composability would require program semantics, in particular $f_{t'}$, to be error strict in both directions, which is of course not the case in general. Note also that both theorems would not hold if we would relax the strong error strictness conditions for data representations.

In practice we need more generally applicable versions of both the vertical and horizontal transitivity theorem. We will often find intermediate input and output data domains not to be exactly the same.

Let, for vertical composition, commutative inner diagrams and the outer

diagram with its data representations ρ_i ; ρ_i' and ρ_o ; ρ_o' be as in Figure 8.

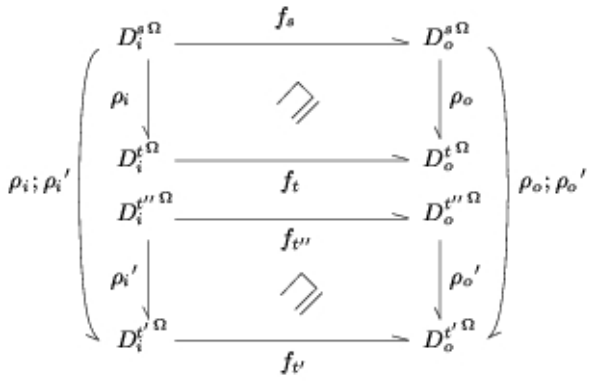


Fig 8. Weak vertical composability: If the inner diagrams are commutative and f_t and $f_{t''}$ appropriately coincide, then the outer diagram is commutative as well.

In order to ensure that ρ_i ; ρ_i' and ρ_o ; ρ_o' are error strict in both directions, we require the intermediate error sets to agree, i.e. $A_i^t = A_i^{t''}$, $U_i^t = U_i^{t''}$, $A_o^t = A_o^{t''}$, and $U_o^t = U_o^{t''}$. Furthermore, let

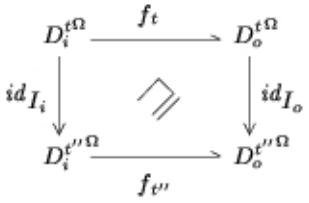
$$I_i =_{def} \rho_i(D_i^{s\,\Omega}) \cap \rho_i'^{-1}(f_t^{-1}(D_o^{t'\Omega})) \subseteq D_i^{t\,\Omega} \cap D_i^{t''\Omega}$$

$$I_o =_{def} D_o^{t\,\Omega} \cap \rho_o'^{-1}(D_o^{t'\Omega}) \subseteq D_o^{t\,\Omega} \cap D_o^{t''\Omega}$$

denote appropriate restrictions of domain and codomain (on the intermediate level) of input and output data representations, respectively. Then, if f_t contains $f_{t''}$ on $I_i \times I_o$, we can prove the following vertical composition corollary:

Corollary 5.1. If the two inner diagrams of Figure 8 are commutative, if the intermediate error sets agree, and if $f_t|_{I_i \times I_o} \supseteq f_{t''}|_{I_i \times I_o}$ then the outer diagram is commutative as well.

Proof. Consider the following (coupling) diagram:



Since $f_t|_{I_i \times I_o} \supseteq f_{t''}|_{I_i \times I_o}$ it is immediately clear that this diagram is commutative. Thus, the outer diagram is commutative due to vertical composition Theorem 5.2.

For horizontal composition, let commutative (inner) diagrams and the outer diagram with sequential compositions f_s ; $f_{s'}$ respectively f_t ; $f_{t'}$ be as in Figure 9. Again, for strong error strictness reasons, we require the intermediate error sets to agree, i.e. $A_o^s = A_i^{s'}$, $U_o^s = U_i^{s'}$, $A_o^t = A_i^{t'}$, and $U_o^t = U_i^{t'}$.

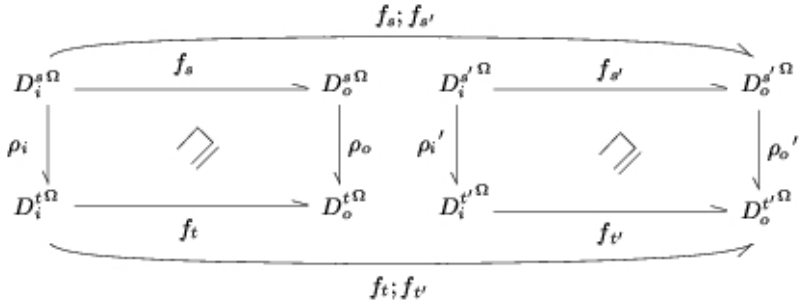


Fig 9: Weak horizontal composability: If the inner diagrams are commutative and the data representations ρ_o and ρ_i' appropriately coincide, then the outer diagram is commutative as well.

Let, again,

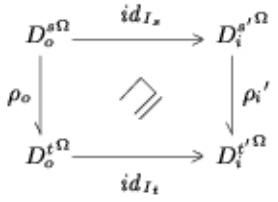
$$I_s =_{def} f_s(D_i^{s\,\Omega}) \cap D_i^{s'\Omega} \subseteq D_o^{s\,\Omega} \cap D_i^{s'\Omega}$$

$$I_t =_{def} f_t(\rho_i(D_i^{s\,\Omega})) \cap f_{t'}^{-1}(D_o^{t'\Omega}) \subseteq D_o^{t\,\Omega} \cap D_i^{t'\Omega}$$

denote appropriate restrictions of (intermediate) domain and codomain of source and target semantics, respectively. Then, if the data representation relation ρ_o is contained in ρ_i' on $I_s \times I_t$, we can prove the following horizontal composition corollary:

Corollary 5.2. If the two inner diagrams of Figure 9 are commutative, if the intermediate error sets agree, and if $\rho_o|_{I_s \times I_t} \subseteq \rho_i'|_{I_s \times I_t}$ then the outer diagram is commutative as well.

Proof. Consider the following (coupling) diagram:



Since $\rho_o\,|_{I_s \times I_t} \subseteq \rho_i'\,|_{I_s \times I_t}$ it is immediately clear that this diagram is commutative. Thus, the outer diagram is commutative due to horizontal composition Theorem 5.3. □

Both more general theorems have been proved by construction of commutative coupling diagrams. In both cases, we have been looking for as weak as possible conditions under which we are allowed to compose commutative diagrams. Note, however, that we might sometimes be able to prove commutativity of more complex diagrams from commutativity of constituent diagrams even under weaker assumptions. Note also, that we might be able to prove correct implementation for a composite diagram, even though one or more component diagrams are not commutative.

Let us finally note, that every theorem and corollary in this section does hold for any of our notions of correct implementation, i.e. not only for correct relative (partial) implementation but also for correct acceptable and correct regular (total) implementation. We have defined a family of correct implementation notions which allows for more elaborated and sophisticated adjustments to whatever the practical requirements for correct implementation really are. And the essential (proof engineering) quality of composability and hence modularizeability is guaranteed for any choice.

6. Correct Compiler Programs

Main constituents of a programming language are its set L of abstract syntactical programs and its language semantics $[\![\cdot]\!]_{\mathsf{L}} : \mathsf{L} \to \mathsf{Sem}_{\mathsf{L}}$, a partial function from L into an associated semantics space $\mathsf{Sem}_{\mathsf{L}}$. The domain of $[\![\cdot]\!]_{\mathsf{L}}$ is the set of so-called *proper* or *well-formed* programs. In case of a well-formed π and of sequential imperative programming languages we are aiming at, $[\![\pi]\!]_{\mathsf{L}}$ can be defined as a relation between extended input and output data domains $\mathrm{D}_i^{\mathsf{L}\,\Omega}$ and $\mathrm{D}_o^{\mathsf{L}\,\Omega}$ as discussed in the previous sections:

$$[\![\pi]\!]_{\mathsf{L}} \in \mathsf{Sem}_{\mathsf{L}} =_{def} (\mathrm{D}_i^{\mathsf{L}\,\Omega} \rightharpoonup \mathrm{D}_o^{\mathsf{L}\,\Omega})$$

For a source language SL, a target language TL and proper source programs $\pi_s \in \mathsf{SL}$ and $\pi_t \in \mathsf{TL}$ with semantics $f_s = [\![\pi_s]\!]_{\mathsf{SL}}$ and $f_t = [\![\pi_t]\!]_{\mathsf{TL}}$, section 5 defines the semantical relation $f_t \sqsupseteq f_s$ of correct implementation:

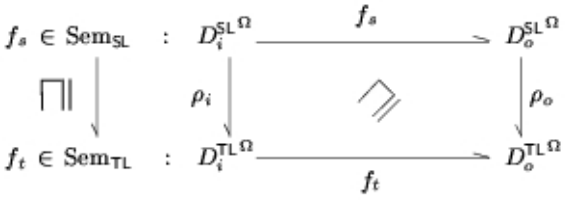


Fig 10. Correct implementation for sequential imperative programs

Data representations ρ_i and ρ_o and associated acceptable and unacceptable error sets are implicit parameters of $\sqsupseteq$. Note also, that $\sqsupseteq$ implicitly defines if we mean preservation of relative (partial), acceptable or regular (total) correctness. We have not yet fixed any one of these parameters.

6.1. Compiling Specifications. Every compiler program establishes a mapping between source and target programs, actually between source and target program representations like for instance character sequences on the one and linkable object code format on the other hand. In order to talk about this mapping abstractly and to relate source and target programs semantically, we assume that we have or can define a compiling (or transformation) specification

$$\mathbf{C} : \mathsf{SL} \rightharpoonup \mathsf{TL},$$

a mathematical relation between abstract source and target programs. $\mathbf{C}$ might be given by a closed inductive definition, more or less constructive, or by a set of

bottom-up rewrite rules to be applied by a term or graph rewrite system (e.g. bottom up rewrite systems, BURS [Pelegri/Graham88]) as for instance used in rule-based code generators.

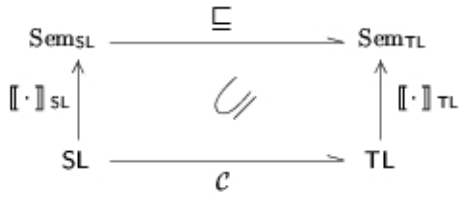


Fig 11. Correctness of the compiling specification **C**

Definition 6.1 (Correct compiling specification) We call **C** *correct*, if and only if for any well-formed source program $\pi_s \in$ SL, every associated target program $\pi_t \in \mathbf{C}(\pi_s)$ is a correct implementation of π_s, i.e. if and only if the diagram in Figure 11 is commutative in the sense

$$([\![\cdot]\!]_{SL}^{-1}; \mathbf{C}) \subseteq (\sqsubseteq; [\![\cdot]\!]_{TL}^{-1}).$$

Note that we do not require **C** to be defined for all well-formed SL-programs, and we will also not require this property for compiler program semantics. Due to resource restrictions of finite host machines we won't be able to prove it for compiler programs, anyway, because realistic source languages contain arbitrarily large programs.

For any two programming languages SL and TL there is an implicitly given *natural correct compiling specification* that simply relates any well-formed source program in SL to every of its correct implementations in TL:

$$\mathbb{C} =_{def} [\![\cdot]\!]_{SL}; \sqsubseteq; [\![\cdot]\!]_{TL}^{-1}$$

The following calculation shows, that $\mathbb{C}$ is correct (Actually, if we only consider well-formed programs, it is the largest correct compiling specification.):

$$([\![\cdot]\!]_{SL}^{-1}; \mathbb{C}) =$$
$$= ([\![\cdot]\!]_{SL}^{-1}; [\![\cdot]\!]_{SL}; \sqsubseteq; [\![\cdot]\!]_{TL}^{-1}) \subseteq (\sqsubseteq; [\![\cdot]\!]_{TL}^{-1}).$$

But how is a correct **C** related to $\mathbb{C}$ in general? Of course, $\mathbb{C} \not\subseteq \mathbf{C}$ (**C** might even be empty, e.g. for the pathological compiler which fails on every source program). Restricted to well-formed source programs, **C** is a subset of $\mathbb{C}$. However, in general **C** might relate non well-formed SL-programs (which have no semantics) to TL-programs. Perhaps well-formedness is hard to decide or even undecidable. So compilers sometimes generate or have to generate target code also for improper source programs without explicitly signaling an error. The user should be careful, keep this in mind and avoid non-well-formed compiler inputs. In any case, in general $\mathbf{C} \not\subseteq \mathbb{C}$ as well.

Hence, so far **C** is unrelated to $\mathbb{C}$, and so will be any correct implementation of **C**. This observation suggests to view at the program sets SL and TL as data domains and extend them by particular unacceptable error elements. This will allow us to also formally express in particular the well-formedness precondition that source programs have to fulfill if they are to be correctly compiled. We will do so also for the semantics spaces Sem_{SL} and Sem_{TL}.

For SL^{Ω} and TL^{Ω} we need an unacceptable error *nwf* (for "non-well-formed") in U_{SL} and U_{TL}, and for Sem_{SL}^{Ω} and Sem_{TL}^{Ω} we need an unacceptable error *uds* (for "undefined semantics") in $U_{Sem_{SL}}$ and $U_{Sem_{TL}}$. **C**, $\mathbb{C}$, $[\![\cdot]\!]_{SL}$, $[\![\cdot]\!]_{TL}$ and $\sqsubseteq$ are extended so that these artificial error elements are related to each other and to non-well-formed programs in SL and TL. Again we denote the extended relations by the same symbols.

Observe in Figure 12 (b), that we used $\sqsupseteq_v$ to indicate the difference between (horizontal) correct implementation $\sqsupseteq$ and (vertical) correct compiler implementation $\sqsupseteq_v$ (see section 6.3.3). For $\sqsupseteq_v$ (respectively $\sqsupseteq_v$) only preservation of relative (partial) correctness makes sense in practice, i.e. commutativity of diagram (b) is only valid if $\sqsupseteq_v$ expresses correct relative (partial) implementation.

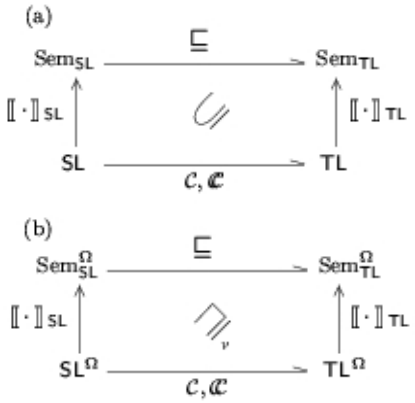


Fig 12. Correctness of extended compiling specifications. The original diagrams (a) (for **C** and $\mathbb{C}$) are commutative if and only if the corresponding extended diagrams (b) are.

The extended **C** is a correct implementation of the extended $\mathbb{C}$ (Figure 13) and hence this step homogeneously fits on top of a stack of further commutative diagrams establishing correct transformation and implementation steps, all correctly tied together and related to $\mathbb{C}$ by vertical composability due to Theorem 5.2 and Corollary 5.1 from section 5. As any compiler program, also the compiling specification **C** reflects design decisions. It already selects particular target programs from the set of all possible correct implementations.

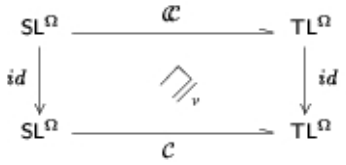


Fig 13. Correctness definition for extended compiling specifications

Theorem 6.1 (Correct compiling specifications) A compiling specification **C** is correct, if and only if it is a correct implementation of $\mathbb{C}$.

Proof: *If:* by Figure 13, Figure 12 (b) and vertical composition.

Only if: Let $\pi_s \in \mathsf{SL}$ and $\mathbb{C}(\pi_s) \subseteq \subseteq \mathsf{TL} \cup A_{\mathsf{TL}}$ (*), and let $\pi_t \in \mathbf{C}(\pi_s)$. We have to show: $\pi_t \in \mathbb{C}(\pi_s) \cup A_{\mathsf{TL}}$. First note that π_s is well-formed, i.e. has semantics $[\![\pi_s]\!]_{\mathsf{SL}} \in \mathsf{Sem}_{\mathsf{SL}}$, because otherwise π_t and $\mathbb{C}(\pi_s)$ would be the unacceptable error *nwf* $\notin \mathsf{TL} \cup A_{\mathsf{TL}}$ which contradicts (*). If $\pi_t \in A_{\mathsf{TL}}$, then we are done. $\pi_t \in U_{\mathsf{TL}}$ is impossible, because π_t would be *nwf* and hence π_s not well-formed.

So let $\pi_t \in \mathsf{TL}$. Then, $([\![\pi_s]\!]_{\mathsf{SL}}, \pi_t) \in \in ([\![\cdot]\!]_{\mathsf{SL}}^{-1} ; \mathbf{C})$. Due to correctness of **C** (Figure 12 (a)) we have $([\![\pi_s]\!]_{\mathsf{SL}}, \pi_t) \in \in (\sqsubseteq ; [\![\cdot]\!]_{\mathsf{TL}}^{-1})$, that is to say $(\pi_s, \pi_t) \in \in ([\![\cdot]\!]_{\mathsf{SL}} ; \sqsubseteq ; [\![\cdot]\!]_{\mathsf{TL}}^{-1})$. But the latter is exactly the definition of $\mathbb{C}$, hence we have $\pi_t \in \mathbb{C}(\pi_s)$.

6.2. Correct Compiler Programs. In order to prove that a compiler program (sometimes also called compiler implementation or simply compiler) is correct, we want to relate its semantics to the compiling specification. It is often a good advice to write a compiler in its own source language. In general, though, the compiler will be implemented in a high level or a low level machine host language HL with semantics space

$$\mathsf{Sem}_{\mathsf{HL}}{}^{\Omega} = (D_i^{\mathsf{HL}\,\Omega} \rightharpoonup D_o^{\mathsf{HL}\,\Omega})^{\Omega}.$$

If we want to call an HL-program τ_h a compiler from SL to TL, then we need representation relations φ_s and φ_t to represent SL- and TL-programs as data in $\mathsf{SL}'^{\,\Omega} =_{def} D_i^{\mathsf{HL}\,\Omega}$ resp. in $\mathsf{TL}'^{\Omega} =_{def} D_o^{\mathsf{HL}\,\Omega}$. Note that there are a lot of data in SL′ and TL′ which do not represent programs, but for a consistent presentation we prefer to let a compiler program τ_h just be like any other HL-program.

The situation is as described in Figure 14. However, in order to treat languages of concrete program representations like SL′ and TL′ as reasonable programming languages, we require that $[\![\cdot]\!]_{\mathsf{SL}'} =_{def} \varphi_s^{-1} ; [\![\cdot]\!]_{\mathsf{SL}}$ and $[\![\cdot]\!]_{\mathsf{TL}'} =_{def} =_{def} \varphi_t^{-1} ; [\![\cdot]\!]_{\mathsf{TL}}$ are single-valued partial functions. Thus, any concrete representation of a well-formed SL- or TL-program has a unique semantics.

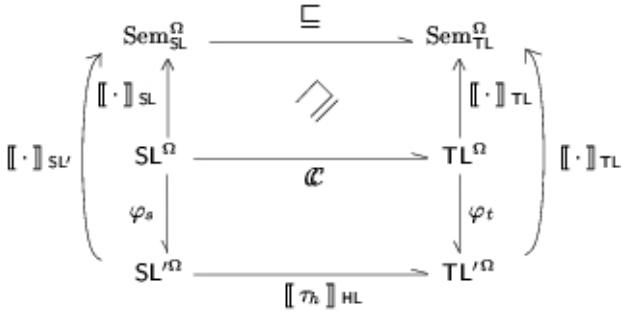


Fig 14. Compiler programs related to compiling specifications. If the lower diagram is commutative as well, we call τ_h a correct compiler program

Definition 6.2 (Correct compiler program) We call τ_h a *correct compiler program*, iff $[\![\tau_h]\!]_{HL} \sqsupseteq \mathbb{C}$, i.e. iff $[\![\tau_h]\!]_{HL}$ is a correct implementation of $\mathbb{C}$.

If τ_h is a correct compiler, then the lower diagram of Figure 14 and, due to vertical composition, also the outer diagram is commutative. Actually, we could equivalently have required $\sqsupseteq$-commutativity of the outer diagram. This would entail commutativity of both inner diagrams and in particular of the lower diagram (the upper diagram is commutative anyway).

The proof of the latter remark is a bit more detailed, but analogous to that of Theorem 6.1: Let $\pi_s \in SL$ and $\varphi_t(\mathbb{C}(\pi_s)) \subseteq$ $\subseteq TL' \cup A_{TL'}$ (*), and let $\pi'_t \in [\![\pi_s]\!]_{SL}(\varphi_s(\pi_s))$. We have to show $\pi'_t \in \varphi_t(\mathbb{C}(\pi_s)) \cup A_{TL'}$. First note that π_s is well-formed, i.e. has semantics $[\![\pi_s]\!]_{SL}$. Otherwise, $\mathbb{C}$ would assign $nwf \in U_{TL}$ to π_s which contradicts (*). If $\pi'_t \in A_{TL'}$, we are done. If $\pi'_t \in U_{TL'}$, then by commutativity of the outer diagram $[\![\pi_s]\!]_{SL} \in (\sqsubseteq; [\![\cdot]\!]_{TL'}^{-1})^{-1}(U_{TL'}) = \{uds\}$ contradicting well-formedness of π_s. So let $\pi'_t \in TL'$. Commutativity of the outer diagram means $([\![\cdot]\!]_{SL'}^{-1}; [\![\tau_h]\!]_{HL})([\![\pi_s]\!]_{SL}) \subseteq$ $\subseteq (\sqsubseteq; [\![\cdot]\!]_{TL'}^{-1})([\![\pi_s]\!]_{SL}) \cup A_{TL'}$ and therefore $\pi'_t \in (\sqsubseteq; [\![\cdot]\!]_{TL'}^{-1})([\![\pi_s]\!]_{SL}) = \varphi_t([\![\cdot]\!]_{TL}^{-1}(\sqsubseteq$ $\sqsubseteq ([\![\pi_s]\!]_{SL})))$. Since π'_t and $[\![\pi_s]\!]_{SL}$ are regular in TL' resp. Sem_S, we have $\pi'_t \in$ $\in \varphi_t(\mathbb{C}(\pi_s))$ by definition of $\mathbb{C}$. □

Any correct compiler program τ_h induces an associated correct extended compiling specification $\mathbf{C}_\tau =_{def} (\varphi_s; [\![\tau_h]\!]_{HL}; \varphi_t^{-1}) : SL^\Omega \rightharpoonup TL^\Omega$ such that

$$[\![\tau_h]\!]_{HL} \sqsupseteq \mathbf{C}_\tau \sqsupseteq \mathbb{C}.$$

If $[\![\tau_h]\!]_{HL}$ is a correct implementation of any correct specification $\mathbf{C}$, i.e., $[\![\tau_h]\!]_{HL} \sqsupseteq \mathbf{C} \sqsupseteq \mathbb{C}$, then τ_h is a correct compiler program (cf. Figure 15). That is to say: A compiler program is correct, if and only if it is the correct implementation of a correct compiling specification.

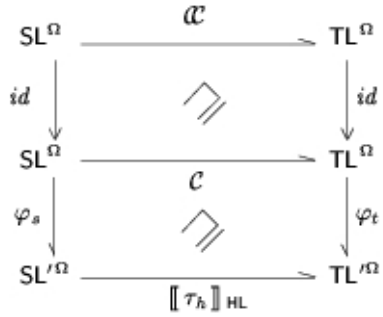


Fig 15. Compiler programs and compiling specifications. Due to vertical composition, a correct implementation of a correct compiling specification is a correct compiler

But what does happen, if we apply a correct compiler program to the representation of a well-formed source program? It should not be a surprise, that we will get *at most* a representation of a correct implementation of the source program:

Theorem 6.2. Let τ_h be a correct compiler program and let $\pi'_s \in \varphi_s(\pi_s)$ be the representation of a well-formed SL-program. Then any regular $\pi'_t \in$ $\in [\![\tau_h]\!]_{HL}(\pi'_s)$ represents a correct implementation π_t of π_s.

Proof: Since π_s is well-formed, $\mathbb{C}(\pi_s) \subseteq TL$ and hence, since τ_h is a correct compiler program, we have $(\varphi_s; [\![\tau_h]\!]_{HL})(\pi_s) \subseteq$ $\subseteq (\mathbb{C}; \varphi_t)(\pi_s) \cup A_o^{HL}$. Thus,

$$\pi'_t \in [\![\tau_h]\!]_{HL}(\pi'_s) \subseteq [\![\tau_h]\!]_{HL}(\varphi_s(\pi_s)) \subseteq$$
$$\subseteq (\mathbb{C}; \varphi_t)(\pi_s) \cup A_o^{HL}.$$

But π'_t is regular, i.e. $\pi'_t \notin A_o^{HL}$. Therefore, $\pi'_t \in (\mathbb{C};\varphi_t)\ (\pi_s)$. So $\pi'_t \in \varphi_t(\pi_t)$ for a $\pi_t \in \in \mathbb{C}(\pi_s)$, which means $[\![\pi_t]\!]_{TL} \sqsupseteq [\![\pi_s]\!]_{SL}$. □

Let us call a concrete SL´- or TL´-datum π'_s or π'_t a well-formed SL´- or TL´-program, if it represents a well formed SL- or TL-program. Then π'_s or π'_t have semantics $[\![\pi'_s]\!]_{SL'}$ respectively $[\![\pi'_t]\!]_{TL'}$. Thus, it is defined when π'_t correctly implements π'_s. According to Theorem 6.2, every regular result $\pi'_t \in [\![\tau_h]\!]_{HL}(\pi'_s)$ of a correct compiler τ_h, applied to a well-formed π'_s, correctly implements π'_s.

That is to say: A correct compiler, applied to a well-formed source program, returns *at most* correct implementations of that source program.

6.3. Discussion and First Summary. We want to summarize some important observations and give some additional explanations. In particular, we will relate the definitions and notions defined in the previous sections to our informal sketch of a conscientious compiler correctness proof as of section 4.1 and in particular of Figure 2. Moreover, we will discuss McKeeman's T-diagram notation, give some remarks on the difference between correct implementation of user programs and of compiler programs, and finally we want to discuss correct implementation for optimizing compilers.

6.3.1. Precise View at the Three Steps. Let us first come back to the diagram shown in Figure 2 (page 13) in section 4.1. In the previous sections (5, 6.1 and 6.2) we have exactly defined every single notion mentioned in section 4.1 and, hence, we now know precisely every conjecture we have to prove in order to implement an SL to TL compiler correctly as an executable program on a host processor HM.

In Figure 2, every data set, program set and semantics space, every program semantics, data representation, program representation, semantics function, compiling specification, compiler semantics and semantics relation has to be appropriately extended by unacceptable and acceptable error elements. The following commutative diagrams (Figure 16 and 17) precisely express that **C** is a correct compiling specification, and that τ_{HL} respectively τ_{HML} are correct compiler programs.

Programs and their representations have equal semantics. But we should explicitly note that in diagram 16 the compiler program τ_{HML} is *not* a representation of τ_{HL}. These two programs have in general different semantics, but the former is a correct implementation of the latter.

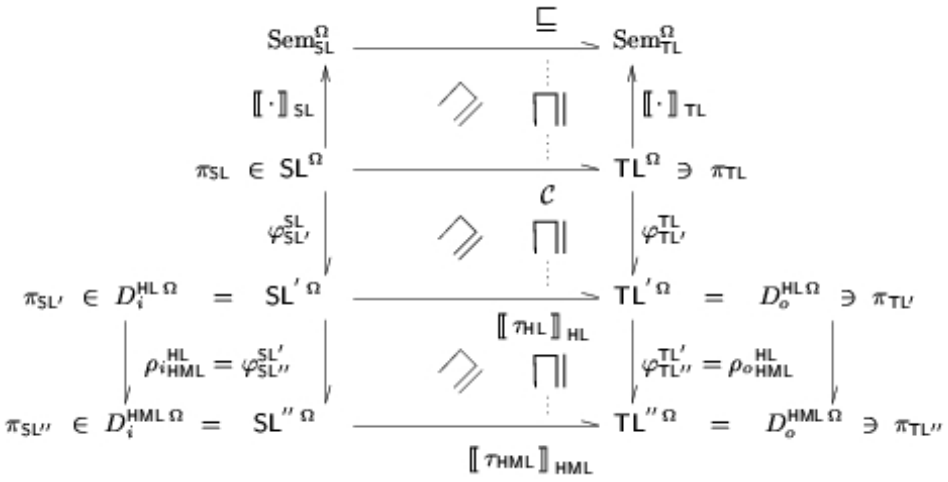


Fig 16. This diagram is again illustrating the three steps for correct compiler implementation as of Figure 2 on page ??

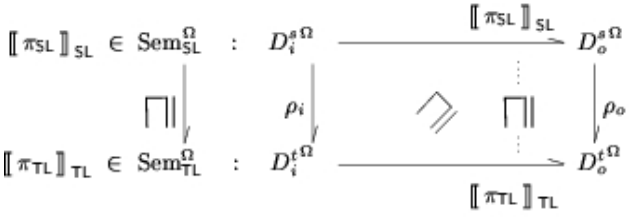


Fig 17. This diagram is again illustrating correct implementation as of Figure 4 on page ??. Note that $[\![\pi_{SL}]\!]_{SL} = [\![\pi'_{SL}]\!]_{SL'} = [\![\pi''_{SL}]\!]_{SL''}$ and analogously for TL

6.3.2. T-Diagram Notation. McKeeman's so-called *T-diagrams* allow to illustrate the effects of iterated compiler applications in an intuitive way. We use them as shorthand notations for particular diagrams as of Figure 18.

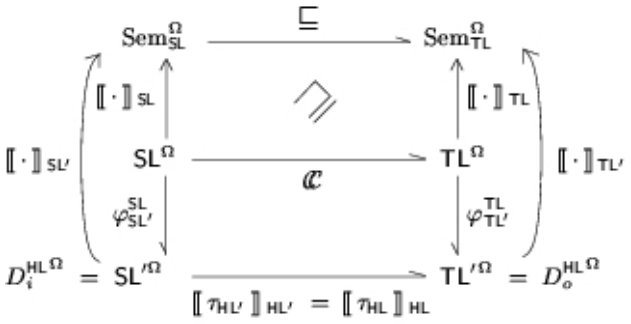


Fig 18. The situation which we will abbreviate by McKeeman's T-diagrams

Recall that $\mathbb{C}$ is the natural correct compiling specification from SL to TL. Well-formed programs and their (syntactical) φ-representations have equal semantics, and $\tau_{HL'} \in \varphi^{HL}{}_{HL'}(\tau_{HL})$ is a well-formed HL′-program compiling syntactical SL′-programs to syntactical TL′-programs. HL′ is the domain of perhaps more concrete syntactical representations of HL-programs. In this situation we use *T-diagram* (Figure 19) as an abbreviation for the diagram in Figure 18. However, we have to keep in mind that the concrete situation is a bit more involved,

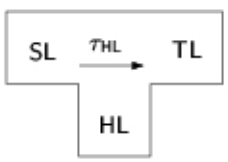


Fig 19. McKeeman's T-diagram as a short-hand for the above situation

that the T-diagrams do not explicitly express crucial differences between various program representations. We need to distinguish programs, program semantics and (syntactical) program representations in order to suffice requirements from practice. We cannot put practitioners short by elegant but too nebulous idealizations.

6.3.3. Correct Implementation versus Correct Compiler Implementation. If we bootstrap compiler programs, we have in general to distinguish between two different notions of correct implementation. Source programs are to be correctly implemented by target programs (relating source to target programs) on the one hand, and the compiler itself is to be correctly implemented on the host machine (which relates the compiler source program to the compiler machine program).

Error behavior and required parameterization of application programs π_{SL}, π_{TL} and their representations $\pi_{SL'}$, $\pi_{TL'}$, $\pi_{SL''}$, $\pi_{TL''}$ are in general of a different nature and independent of the expected error behavior and required parameterization for the compiler, i.e. for the specification C and compiler programs τ_{HL}, τ_{HML} and their syntactical representations.

For instance, let us assume SL to be a process programming language. The process programmer would not like to witness any uncertainty nor error at computation time of source programs π_{SL} respectively $\pi_{SL'}$, $\pi_{SL''}$. That is source programs are written such that

$$\varnothing \neq [\![\pi_{SL}]\!]_{SL}(d_i^s) \subseteq D_o^s \qquad (1)$$

holds whenever π_{SL} is applied to an input $d_i^s \in D_i^s \setminus [\![\pi_{SL}]\!]_{SL}^{-1}(U_o^s)$ outside the domain of computations which possibly end in unacceptable errors. But this involves regular termination and hence total correctness of π_{SL} which the process programmer requires to be preserved for any correct implementation π_{TL}. He/she wants that

$$\varnothing \neq [\![\pi_{TL}]\!]_{TL}(d_i^t) \subseteq D_o^t \qquad (2)$$

holds whenever the target program π_{TL} is applied upon the representation $d_i^t \in \rho_i(d_i^s)$, $d_i^s \notin [\![\pi_{SL}]\!]_{SL}^{-1}(U_o^s)$, of the corresponding input. Correct regular (total) implementation together with (1) guarantees (2), because due to section 5.1, we have

$$\varnothing \neq [\![\pi_{TL}]\!]_{TL}(d_i^t) \subseteq \rho_o([\![\pi_{SL}]\!]_{SL}(d_i^s)) \subseteq D_o^t.$$

On the other hand, the same process programmer will (and in general has to) accept compile time error reports, like for instance HM-memory overflow, while π_{SL} is compiled to π_{TL}, i.e. while the compiler machine implementation τ_{HML} is executed and applied upon a (syntactical) representation π_{SL}'' of the source program π_{SL}. With respect to compilation, the process programmer wants a guarantee at execution time of π_{TL}'' whenever τ_{HML} has succeeded on HM and has generated the target program representation π_{TL}'', which means that (2) is established by successful execution of the compiler implementation τ_{HML}.

Note that there are no obvious natural mappings between the error sets A_o^s, U_o^s , A_o^t, U_o^t crucial for the correct implementation of source programs by target programs on the one hand, and the error sets $A_{TL'} =_{def} A_o^{HL}$, $U_{TL'} =_{def} U_o^{HL}$, $A_{TL''} =_{def} A_o^{HML}$ and $U_{TL'} =_{def} U_o^{HML}$ crucial for the correct implementation notion for generating the compiler machine executable τ_{HML} on the other hand. We have to distinguish these two correct implementation relations.

6.3.4. Notes on Optimizing Compilers. As already mentioned, many existing and in particular optimizing compiler programs τ_h do not check all pre conditions necessary for correct compilation of source programs. In particular optimizing transformations often need pre-conditions which, for practical reasons, are too hard to decide or are even algorithmically undecidable [3]. Therefore, in general compiling specifications **C** or compiler program semantics $[\![\tau_h]\!]_{HL}$ might yield unreasonable target programs even for well-formed programs for which additional optimization pre-conditions do not hold.

Our more elaborated view at correct implementation offers a remedy which exploits the notion of acceptable errors in A_{TL}. Let us think of a compiler warning (like for instance "Warning: array index check omitted in line ...") as a *potential error message*, i.e. as an indication for an eventually generated target program π_t to potentially belong to the set of acceptable errors in A_{TL} in the following sense: "We [the compiler] give you [the compiler user] the following target program π_t, but it contains optimizations which require additional pre conditions $\Phi \subseteq D_i^s$ for your source program to hold. If you cannot guarantee Φ, please take this as an error message, because π_t might not be correct."

That is to say: Besides the usual compiling specification **C** every source program π_s carries an additional (optimization) pre-condition $\Phi = PC(\pi_s) \subseteq D_i^s$, $PC : SL \to 2^{D_i^s}$, which leads to a modified source language semantics

$$[\![\pi_s]\!]_{SL,PC} =_{def} [\![\pi_s]\!]_{SL} \cup (D_i^s \setminus PC(\pi_s)) \times \{pcf\},$$

where $pcf \in U_o^s$ reads as "(optimization) pre-condition false". Now, if compiling specification or compiler program deliver a target program together with an optimization warning, then this guarantees a weaker correct implementation of π_s by π_t, namely that

$$(\rho_i ; [\![\pi_t]\!]_{TL})(d_i^s) \subseteq ([\![\pi_s]\!]_{SL}; \rho_o)(d_i^s) \cup A_o^t$$

holds for all $d_i^s \in PC(\pi_s)$ with $[\![\pi_s]\!]_{SL}(d_i^s) \subseteq$ $\subseteq D_i^s \cup A_o^s$. This weaker notion of correct implementation (with respect to $[\![\cdot]\!]_{SL}$) can equivalently be expressed by usual correct implementation, but with respect to the weaker semantics relation $[\![\cdot]\!]_{SL,PC}$.

A well-known optimization is the so-called *redundant (dead) code elimination*, which might violate preservation of relative (partial) correctness, e.g. which might eliminate the code that for some

[3] For many programming languages it is algorithmically undecidable whether or not for instance variables are initialized before they are used, or if programs terminate regularly.

input data $d_i^s \in D_i^s$ would cause a runtime error like for instance a division by zero. The source program π_s might be partially (relatively) correct w.r.t. some pre- and post-conditions Φ' resp. Ψ', whereas the optimized target program π_t is not. It might return a regular but incorrect result if applied to $d_i^t \notin \rho_i(\Phi)$. If the additional optimization pre condition Φ does not hold for the input, π_t might dangerously deceive the user. Its result might have nothing in common with any regular source program result d_o^s.

A different optimization is the so-called *unswitching*, which might violate preservation of total (regular) correctness. Unswitching is an optimization that moves conditional branches outside loops and in particular changes the sequential order of conditional expressions while transforming

$$\textit{while}(b, \textit{if}(c, \pi_1, \pi_2)) \mapsto \textit{if}(c, \textit{while}(b, \pi_1), \textit{while}(b, \pi_2)).$$

This transformation does in general not preserve regular (total) correctness. A process programmer, who has proved regular (total) correctness of a source program π_s (containing the left statement) would dangerously be cheated by the program π_t (containing an implementation of the right statement instead) if π_t is applied to input data $d_i^t \notin \rho_i(\Phi)$ such that b evaluates to *false* and c causes a runtime error. In that case, π_t would incorrectly irregularly abort with an error, which might lead to a dangerous situation if π_t controls for instance a safety critical process.

7. From Correct Compiler Programs to Trusted Compiler Implementations

At this point we have developed a mathematical theory of expressing compiling and compiler implementation correctness in a compositional, rigorous and realistic way using special commutative diagrams. In particular Figure 16 on page 23 precisely defines the three essential proof obligations corresponding to the three tasks necessary in order to provide a rigorous and conscientious correctness proof for an initial trusted compiler executable.

It is the machine (low) level compiler implementation correctness proof by bootstrapping and a-posteriori code inspection, which is new and for the first time closes the implementation gap rigorously and trustworthily. It needs further explanation. Therefore, the forthcoming second part of our article on *Trusted Compiler Implementation* [Goerigk/Langmaack01b] will go into detail on this proof for the initial trusted compiler executable which we have constructed and completely verified. The article will also give a security related motivation for low level implementation verification and show that otherwise correctness and thus trustworthiness cannot be guaranteed.

For machine level compiler implementation verification we will use a specialized bootstrapping technique with a-posteriori code-inspection. Initially, a Common Lisp system runs on and generates code for our compiler on an auxiliary processor. We use it as an unverified compiler generator which transforms its specification (the compiler source program) and generates an (auxiliary) compiler machine executable. The generated executable depends on untrusted auxiliary software and is not verified. Its results have to be checked. Fortunately, we need to use this auxiliary executable only once. N. Wirth's technique of writing the compiler in its own source language enables to bootstrap the compiler source again, and this run, if successful, generates a target program which should be the specified target code for the compiler. Although in general every result of the auxiliary executable would have to be checked, we need a-posteriori result checking only for exactly one run.

This is because we realize an important difference to the general procedure: As soon as the syntactical check succeeds and guarantees, that the generated binary is as specified by the semantically correct compiling specification, a semantics to syntax reduction theorem applies and guarantees that the gener-

ated binary is a (semantically) correct compiler executable on the target machine. Compiling specification verification guarantees, that specification compliance implies semantical correctness.

Source and intermediate languages for the initial compiler are carefully chosen in order to be able to finally produce a convincing complete rigorous proof document. They have particularly been chosen to isolate crucial compilation steps and to enable code inspection by target to source code comparison.

The compiler will *horizontally* be decomposed in a (sequential) composition of subsequent passes. *Vertical* composition originates from horizontal composition: The compiler generates a tower of subsequent intermediate programs of decreasing abstraction level between high-level source and machine level target program. Each step (pass) corresponds semantically to a commutative diagram expressing that the lower-level intermediate program correctly implements the higher-level program.

For the initial compiler every intermediate program representation is made explicit. Every intermediate language has an explicit composition operator expressing the (horizontal) sequential composition of passes. If the compiler is applied to itself, then every compiler pass explicitly transforms step by step the sequential composition of n (= 4) passes to again a sequential composition of n passes. The process finally corresponds semantically to an n by n matrix of commutative diagrams (actually n+1 by n including the specification on top), plugged together by horizontal and vertical composition theorems.

Semantical machine code inspection — to semantically understand and to assure the correctness of generated target code — is tedious, cumbersome, error-prone and probably unmanageable. But also syntactical machine level code inspection, i.e. to compare source and expected low-level machine code with respect to the (correct) compiling specification, is cumbersome and should and can be further minimized. Horizontal and vertical composition with only two small low level transformations will split the code inspection task in a number of small and easier parts. Lowest level machine code inspection is only necessary for the final code generation. Moreover, the bootstrapping process can already make use of checked lower level passes to guarantee implementation correctness for higher level intermediate code, which further helps saving much and in particular low level machine code inspection *below the diagonal* of the matrix mentioned above.

The initial ComLisp compiler uses five subsequent passes including front end and code generation. They are by no means arbitrary. Front end, transformation of recursive procedures using stack technique, of recursive data types implemented on a heap memory, of control structure to linear machine code, and code generation producing binary executable code, all these passes coincide with passes identifiable also in compilers in general. They also adequately modularize the theorem prover supported mechanical proof of compiling correctness.

Acknowledgments We would like to thank our colleagues in the *Verifix* project for many fruitful discussions, in particular Axel Dold, Thilo Gaul, Gerhard Goos, Andreas Heberle, Friedrich von Henke, Ulrich Hoffmann, Vincent Vialard, Wolf Zimmermann. Special thanks to Markus Müller-Olm and Andreas Wolf for their contributions to the notion of relative program correctness and its preservation.

[Abrial+91] J.R. Abrial, M.K.O. Lee, D.S. Neilson, P.N. Scharbach, and I.H. Sørensen. The B-method. In *VDM'91: Formal Software Development Methods Volume 2*, volume 552 of *LNCS*, pages 398−405. Springer-Verlag, 1991.

[Bauer78] F.L. Bauer. Program development by stepwise transformations — the project CIP. In F. L. Bauer and M. Broy, editors, *Program Construction*, volume 69 of *Lecture Notes in Computer Science*, pages 237−266. Springer Verlag, 1978.

[Blum+89] M. Blum, M. Luby, and R. Rubinfeld. Program result checking against adaptive programs and in cryptographic settings. In *DIMACS Workshop on Distributed Computing and Cryptography*, 1989.

[Boerger/Rosenzweig92] E. Börger and D. Rosenzweig. The WAM-definition and Compiler Correctness. Technical Report TR-14/92, Dip. di Informatica, Univ. Pisa, Italy, 1992.

[Broy92] M. Broy. Experiences with software specification and verification using LP, the larch proof assistant. Technical Report 93, Digital Systems Research Center, Palo Alto, July 1992.

[Boerger/Schulte98] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *23rd International Symposium on Mathematical Foundations of Computer Science*, LNCS. Springer, 1998.

[BSI94] BSI — Bundesamt für Sicherheit in der Informationstechnik.BSI-Zertifizierung. BSI 7119, Bonn, 1994.

[BSI96] BSI — Bundesamt für Sicherheit in der Informationstechnik}. OCOCAT-S Projektausschreibung, 1996.

[Chirica/Martin86] L. M. Chirica and D. F. Martin. Toward Compiler Implementation Correctness Proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185−214, April 1986.

[Curzon93a] P. Curzon. Deriving Correctness Properties of Compiled Code. *Formal Methods in System Design*, 3:83-115, August 1993.

[Curzon94] P. Curzon. The Verified Compilation of Vista Programs. Internal Report, Computer Laboratory, University of Cambridge, January 1994.

[Dijkstra76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Dold00] Axel Dold. *Formal Software Development using Generic Development Steps*. Logos-Verlag, Berlin, 2000. Dissertation, Universität Ulm.

[Fagan86] M. E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, SE-12(7):744-751, 1986.

[Flatau92] A. D. Flatau. *A verified implementation of an applicative language with dynamic storage allocation*. PhD thesis, University of Texas at Austin, 1992.

[Garland/Guttag91] S.J. Garland and J.V. Guttag. A guide to lp, the larch prover. Technical Report SRC Report 82, Digital Systems Research 'Center, 1991.

[Gaul+96] Th. Gaul, G. Goos, A. Heberle, W. Zimmermann, and W. Goerigk. An Architecture for Verified Compiler Construction. In *Joint Modular Languages Conference JMLC'97*, Linz, Austria, March 1997.

[George+92] C. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielson, S. Prehn, and K. R. Wagner. *The Raise Specification Language*. Prentice Hall, New York, 1992.

[Goerigk/Langmaack01] W. Goerigk and H. Langmaack. Compiler Implementation Verification and Trojan Horses. In D. Bainov, editor, *Proc. 9th International Colloquium on Numerical Analysis and Computer Science with Applications*, Plovdiv, Bulgaria, 2001.

[Goerigk/Langmaack01a] W. Goerigk and H. Langmaack. Compiler Implementation Verification and Trojan Horses (Extended Draft Version). Technical report, Institut für Informatik, CAU, 2001. Available under http://www.informatik.uni-kiel.de/~wg/Berichte/Plovdiv.ps.

[Goerigk/Langmaack01b] W. Goerigk and H. Langmaack. Will Informatics be able to Justify the Construction of Large Computer Based Systems? Part II: Trusted Compiler Implementation. International Journal on *Problems in Programming*, 2002. Forthcoming.

[Goerke96] W. Goerke. Über Fehlerbewußtseinskultur. Mündlicher Diskussionsbeitrag, Informatik-Kolloquium, IBM, Böblingen, 1996. Unpublished, 1996.

[Goerigk99a] W. Goerigk. On Trojan Horses in Compiler Implementations. In F. Saglietti and W. Goerigk, editors, *Proc. des Workshops Sicherheit und Zuverlgssigkeit softwarebasierter Systeme*, ISTec Report ISTec-A-367, ISBN 3-00-004872-3, Garching, August 1999.

[Goerigk99b] W. Goerigk. Compiler Verification Revisited. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[Goerigk00a] W. Goerigk. Trusted Program Execution. Habilitation thesis. Techn. Faculty, Christian-Albrechts-Universität zu Kiel, May 2000. To be published.

[Goerigk00b] W. Goerigk. Proving Preservation of Partial Correctness with ACL2: A Mechanical Compiler Source Level Correctness Proof. In M. Kaufmann and J S. Moore, editors, *Proceeding of the ACL2'2000 Workshop*, University of Texas, Austin, Texas, U.S.A., October 2000.

[Gurevich91] Y. Gurevich. Evolving Algebras; A Tutorial Introduction. *Bulletin EATCS*, 43:264-284, 1991.

[Gurevich95] Y. Gurevich. Evolving Algebras: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[Gaul+99] Th. Gaul, W. Zimmermann, and W. Goerigk. Construction of Verified Software Systems with Program-Checking: An Application To Compiler Back-Ends. In A. Pnueli and P. Traverso, editors, *Proc. FLoC'99 International Workshop on Runtime Result Verification*, Trento, Italy, 1999.

[Heberle+98]A. Heberle, Th. Gaul, W. Goerigk, G. Goos, and W. Zimmermann. Construction of Verified Compiler Front-Ends with Program-Checking. In *Proceedings of {PSI} '99: Andrei Ershov Third International Conference on Perspectives Of System Informatics*, volume 1755 of *Lecture Notes in Computer Science*, Novosibirsk, Russia, 1999. Springer Verlag.

[Hoare+93] C. A. R. Hoare, He Jifeng, and A. Sampaio. Normal Form Approach to Compiler Design. *Acta Informatica*, 30:701-739, 1993.

[Hoffmann/Krieg-Brueckner93] B. Hoffmann and B. Krieg-Brückner. *Program Development by Specification and Transformation*. Springer, Berlin, 1993.

[HSE01] C. Jones, R.E. Bloomfield, P.K.D. Froome, and {P.G.} Bishop. Methods for assessing the safety integrity of safety-related software of uncertain pedigree (SOUP). Contract Research Report 337/2001, Health and Safety Executive, Adelard, London, UK, 2001.

[Jones90] C. B. Jones. *Systematic Software Development Using VDM, 2nd ed.* Prentice Hall, New York, London, 1990.

[Kaufmann/Moore94] M. Kaufmann and J S. Moore. Design Goals of ACL2. Technical Report 101, Computational Logic, Inc., August 1994.

[Langmaack73] H. Langmaack. On Correct Procedure Parameter Transmission in Higher Programming Languages. *Acta Informatica*, 2(2):110-142, 1973.

[Langmaack97a] H. Langmaack. Softwareengineering zur Zertifizierung von Systemen: Spezifikations-, Implementierungs-, Übersetzerkorrektheit. *Informationstechnik und Technische Informatik it+ti*, 39(3):41−47, 1997.

[Laprie94] J. C. Laprie, editor. *Dependability: basic concepts and terminology*. Springer Verlag for IFIP WG 10.4, 1994.

[Loeckx/Sieber87] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification (Second edition)*. John Wiley and Sons, New York, N.Y., 1987.

[Mueller-Olm96] M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1997.

[Moore88] J S. Moore. Piton: A verified assembly level language. Techn. Report 22, Comp. Logic Inc, Austin, Texas, 1988.

[Moore96] J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Kluwer Academic Press, Dordrecht, The Netherlands, 1996.

[Mueller-Olm/Wolf00] M. Müller-Olm and A. Wolf. On the Translation of Procedures to Finite Machines. In G. Smolka, editor, *Programming Languages and Systems. Proceedings of ESOP 2000*, volume 1782 of *LNCS*, pages 290−304, Berlin, March 2000.

[McCarthy/Painter67] J. McCarthy and J. A. Painter. Correctness of a compiler for arithmetical expressions. In J.T. Schwartz, editor, *Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.

[Milne/Strachey76] R. Milne and Ch. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976.

[Nielson/Nielson92] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Chichester, 1992.

[Owre+92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings 11th International Conference on Automated Deduction CADE*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748−752, Saratoga, NY, October 1992. Springer-Verlag.

[Partsch90] H. Partsch. *Specification and Transformation of Programs*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[Pelegri/Graham88] Eduardo Pelegri-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application of {BURS} theory. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 15th Annual ACM Symposium on Principels of Programming Languages (POPL '88)*, pages 294−308, San Diego, CA, USA, January 1988. ACM Press.

[Plotkin81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[Pofahl95] E. Pofahl. Methods Used for Inspecting Safety Relevant Software. In W. J. Cullyer, W. A. Halang and B. J. Krämer (eds.): High Integrity Programmable Electronic Systems, Dagstuhl-Sem.-Rep. 107, p 13,1995.

[Polak81] W. Polak. Compiler specification and verification. In J. Hartmanis G. Goos, editor, *Lec-*

ture Notes in Computer Science, number 124 in LNCS. Springer-Verlag, 1981.

[Sampaio93] A. Sampaio. *An Algebraic Approach to Compiler Design*. PhD thesis, Oxford University Computing Laboratory, Programming Research Group, October 1993. Technical Monograph PRG-110, Oxford University.

[Scott70] D. S. Scott. Outline of a Mathematical Theory of Computation. In *Proceedings of the 4th Annual Princeton Conference on Information Sciences and Systems*, pages 169—176, Princeton, 1970.

[Spivey92] J. M. Spivey. *The Z Notation*. Prentice-Hall International Series in Computer Science. Prentice Hall, 1992.

[Scott/Strachey71] D. Scott and Ch. Strachey. Toward a mathematical semantics for computer languages, pages 19—46. April 1971.

[Stoy77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA., London, 1977.

[Schweizer/Voss96] G. Schweizer and M. Voss. Systems engineering and infrastructures for open computer based systems. *Lecture Notes in Computer Science*, 1030:317—??, 1996.

[Thompson84] K. Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761—763, 1984. Also in *ACM Turing Award Lectures: The First Twenty Years 1965—1985*, ACM Press, 1987, and in *Computers Under Attack: Intruders, Worms, and Viruses*, ACM Press, 1990.

[Wirth77} N. Wirth. *Compilerbau, eine Einführung*. B.G. Teubner, Stuttgart, 1977.

[Wolf00] A. Wolf. *Weakest Relative Precondition Semantics - Balancing Approved Theory and Realistic Translation Verification*. PhD thesis, Technische Fakultät der Christian-Albrechts-Universität, Report No. 2013, Kiel, February 2001.

[ZSI89] ZSI — Zentralstelle für Sicherheit in der Informationstechnik.IT-Sicherheitskriterien. Bundesanzeiger Verlagsgesellschaft, Köln, 1989.

[ZSI90] ZSI — Zentralstelle für Sicherheit in der Informationstechnik. IT-Evaluationshandbuch. Bundesanzeiger Verlagsgesellschaft, Köln, 1990.

Date received: 9.11.2002

About authors

Prof. Dr. Wolfgang Goerigk
Professor of Institute of Informatics and Applied Mathematics

Prof. Dr. Hans Langmaack
Professor of Institute of Informatics and Applied Mathematics

Місце роботи авторів
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel, Kiel, Germany

Email: wg@informatik.uni-kiel.de,
hl@informatik.uni-kiel.de