UDC 681.3.51

Wolfgang Goerigk, Hans Langmaack

WILL INFORMATICS BE ABLE TO JUSTIFY THE CONSTRUCTION OF LARGE COMPUTER BASED SYSTEMS? PART II. TRUSTED COMPILER IMPLEMENTATION[1]

The present and the previous article on *Realistic Correct Systems Implementation* together address correct construction and functioning of large computer based systems. In view of so many annoying and dangerous system misbehaviors we want to 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 this sense, i.e. for the design of computer based systems, the formulation of mathematical models of information flows, and the construction of controlling software 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 into host machine code by existing host compilers, is as important. So far there are no realistic recipes to close this gap, although it is known for many years 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. We demonstrate the method of rigorous syntactic a-posteriori code inspection, which has been developed by the research group *Verifix* funded by the Deutsche Forschungsgemeinschaft (DFG).

1. Realistic Correct Systems Implementation

As an introduction to the present article, which focuses on trusted compiler implementation, we want to summarize and repeat some of the important definitions from the previous article [Goerigk/Langmaack01c] on *Realistic Correct Systems Implementation*.

Compiler construction is crucial to the construction of computer based systems, and correct realistic compilers are necessary for a convincing construction process of correct software/hardware systems. Correct realistic compilation is necessary, can be achieved and it establishes a trusted execution basis for further software development. Most faults and bugs are indeed found in software design and high-level software engineering, however, we find considerably many compiler bugs as well [GccBugList], and one or the other turns out to be critical. Unless the implementation and integration gap of software systems is closed with mathematical rigor, the risk of incorrect code generation will always seriously disturb the recognition of and the trust in certainties which are guaranteed by mature software engineering and formal methods.

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. 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. We will demonstrate this in a non-self-evident way in particular for compilers, because they are crucial to low level implementation correctness of application software. At the upper end, software engineers rely on the machine independent semantics for their high-level languages, but they have to be aware of what kind of implementation correctness makes sense and can realistically be guaranteed in their application domain.

[1]Supported by Deutsche Forschungsgemeinschaft (DFG) under the grants La426/15 and /16.

© Wolfgang Goerigk, Hans Langmaack, 2003

Correct realistic compilation is the major topic of the German joint project *Verifix* on *Correct Compilers* of the universities of Karlsruhe, Kiel and Ulm [Goerigk97d, Goos/Zimmermann99]. The goal is to develop repeatable engineering methods for the construction and correct implementation of compiler programs for realistic, industrially applicable imperative and object-oriented source programming languages and real world target and host processors, which generate efficient code that compares to unverified compilers.

A number of key ideas and achievements make this possible and help to modularize the entire effort into sufficiently small steps: an appropriate realistic notion of correct implementation and compiler correctness [Goerigk/Langmaack01c, Mueller-Olm/Wolf99], the reuse of approved compiler architecture and construction and compiler generator techniques [Goos/Zimmermann99], the method of a-posteriori program result checking [Goerigk+98], mechanical proof support [Dold+02], and finally, a trusted initial compiler executable for a realistic imperative high-level system programming language [Goerigk/Hoffmann98b, Goerigk/Hoffmann97]. The initial compiler and in particular its low-level machine implementation verification is the topic of the present paper. The compiler serves as a sound bootstrapping and implementation basis and lifts proof obligations from machine code level to much more abstract source code. It is proved to *at most* generate correct implementations of well-formed source programs; it may fail, e.g. due to resource errors, and if the source program is not well formed, then the target code may cause unacceptable errors and compute unpredictable results. But if the source program is well-formed, and if the compiler succeeds, then the target code is a correct implementation of the source program. The compiler is implemented and runs on a concrete physical processor.

In the companion article [Goerigk/Langmaack01c] on *Realistic Correct Systems Implementation* we gave an extended motivation also for the impact of the question raised in the title of the two articles. Furthermore we modularized the entire compiler verification effort in three steps and developed a mathematical theory which realistically and rigorously allows to express compiling specification correctness and high-level and low-level compiler implementation correctness compositionally by commutative diagrams.

The present article focuses on the realistic method for low-level compiler implementation verification by bootstrapping and syntactical a-posteriori code inspection, which we have developed in order to provide, for the first time, a rigorously verified and fully trusted initial compiler executable.

We want to summarize the important definitions and observations from the previous article [Goerigk/Langmaack01c]. We will repeat the definition of correct implementation between source and target programs, and the modularization of the entire proof effort in three steps. For details, however, we want to refer to [Goerigk/Langmaack01c].

1.1. Compiling Specification and Compiler Implementation. Compiling (specification) verification involves semantics. It is the first step necessary to modularize the entire correctness proof. The second step is *correct compiler implementation*, i.e. the correct transformation of the compiling specification to machine code of the compiler host processor. Proving its correctness is called *compiler implementation verification*. Correct compiler source programs are implemented by bootstrapping. In practice we can further modularize compiler implementation in *high-level* compiler implementation and *low-level* compiler machine implementation.

High-level compiler implementation corresponds to programming. Likewise, high-level compiler implementation verification is program verification. Machine level compiler implementation correctness can be established by a trusted initial compiler, or by syntactical a-posteriori result checking (as in Section 4).

Three tasks are necessary in order to present a convincing correctness proof

of a compiler executable on a compiler host machine. SL and TL are abstract source and target languages with concrete representation domains indicated by primes. HL is the compiler host language, and HML is the compiler host machine language. The compiler program τ_{HL} is written in HL and compiled into executable code τ_{HML} written in HML. $[\![\tau_{HL}]\!]_{HL}$ and $[\![\tau_{HML}]\!]_{HML}$ denote the respective semantical relations. We will later simplify the situation and identify source and host languages as well as target and host machine language. Using this notation, we can modularize the problem in three tasks:

1. Specification of a compiling relation **C** between 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 HL, 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} in HML, and low level compiler implementation verification (with respect to $[\![\tau_{HL}]\!]_{HL}$ and program representations $\varphi^{SL'}_{SL''}$ and $\varphi^{TL'}_{TL''}$).

The last two tasks establish implementation correctness relations between compiling specification, compiler source program and compiler executable. Every task will be formalized by a corresponding commutative diagram (Figure 1).

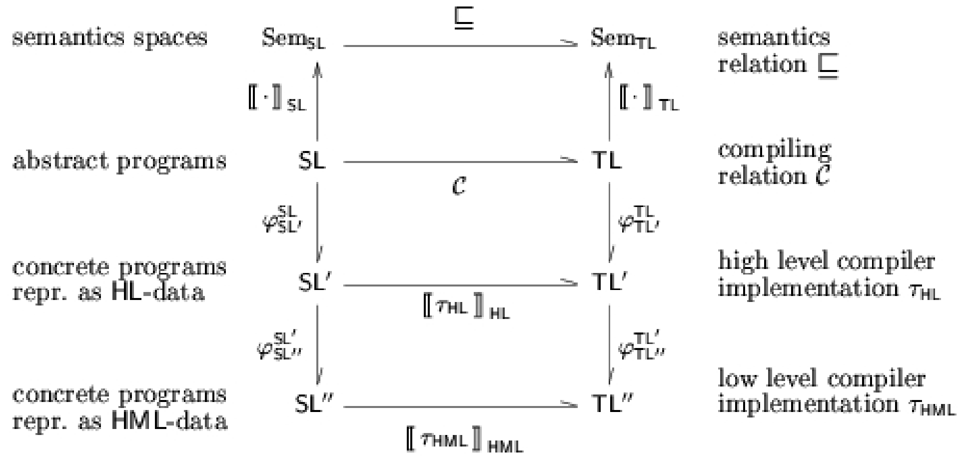


Fig 1. Three tasks for correct compiler specification and implementation

In Figure 1, $A \rightharpoonup B$ denotes the domain of relations and $A \rightarrow B$ of (partial) functions from A to B. The semantical mappings $[\![\cdot]\!]_{SL}$ and $[\![\cdot]\!]_{TL}$ are functions from source language SL resp. target language TL to their semantical domains Sem_{SL} and Sem_{TL} (i.e. the semantics of a well-formed program is uniquely determined). Compiling specifications **C** typically allow for more than only one target program, and also compiler program semantics might be non-deterministic. Data and program representations, which map abstract programs to concrete program representations and to their string representations, are in general relations, anyway.

1.2. Preservation of Relative Correctness. In this article we focus on compilers and transformational programs and formalize operational program semantics by error strict relations $f \subseteq D_i \times D_o$ between input and output domains, which are *extended* by individually associated disjoint non-empty error sets $\Omega = A \uplus U$ of acceptable and unacceptable errors. One particular error $\perp$ models infinite computations.

If f_s is a source and f_t a target program semantics, and if ρ_i and ρ_o are data representation relations (strongly error strict in both directions), then we define f_t to be a *correct implementation* of f_s *relative to* ρ_i and ρ_o and associated error sets, iff

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

holds for all $d \in D_i^s$ with $f_s(d) \subseteq D_o^s \cup A_o^s$. This exactly captures the intuitive requirements: $d \in D_i^s$ is called *admissible*, if it does not cause an unacceptable error outcome, i.e. if $f_s(d) \cap U_o^s = \varnothing$. Non-admissible inputs may cause unpredictable or arbitrary results. But if *d* is admissible, then we can trust in results computed by target program semantics: Either the result is correct (representation of a corresponding source program output) or the target program execution aborts (observably) with an acceptable (resource) error.

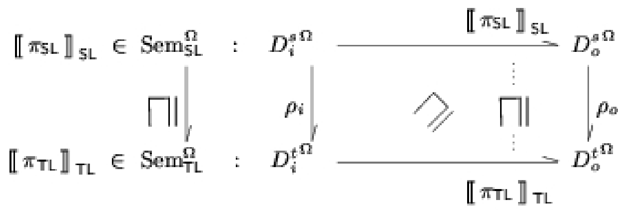


Fig 2. Correct Implementation

The symbol $\sqsubseteq$ denotes this refinement relation, and ";" denotes (diagrammatic) relational composition. $\rho_i ; f_t \sqsupseteq f_s ; \rho_o$ (or $f_t \sqsupseteq f_s$ for short) means that f_t correctly implements or *refines* or *is better than* f_s. Note that $\sqsubseteq$ depicts (basically) a relational inclusion in opposite direction. Details can be found in [Goerigk/Langmaack01c]. Our definition is a relational version of relative correctness preservation [Mueller-Olm/Wolf99, Wolf00] and can be understood as a deliberate generalization of partial respectively total correctness preservation. It is both horizontally and vertically transitive (compositional).

1.3. Precise View at the Three Steps. We have to refer to [Goerigk/Langmaack01c] for details, but we want to repeat the diagram which precisely defines the proof obligations necessary in order to work through a complete compiler correctness proof. We will focus on the lowest diagram of Figure 3 in this article, and it relates the compiler machine executable semantically to the high-level compiler implementation. They work semantically on different input and output data domains of HL and HML which are related by program representation relations [Goerigk/Langmaack01c]. The difference is indicated by primes and double-primes in Figure 3. 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.

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 commutative diagrams of Figure 2 and 3 precisely express

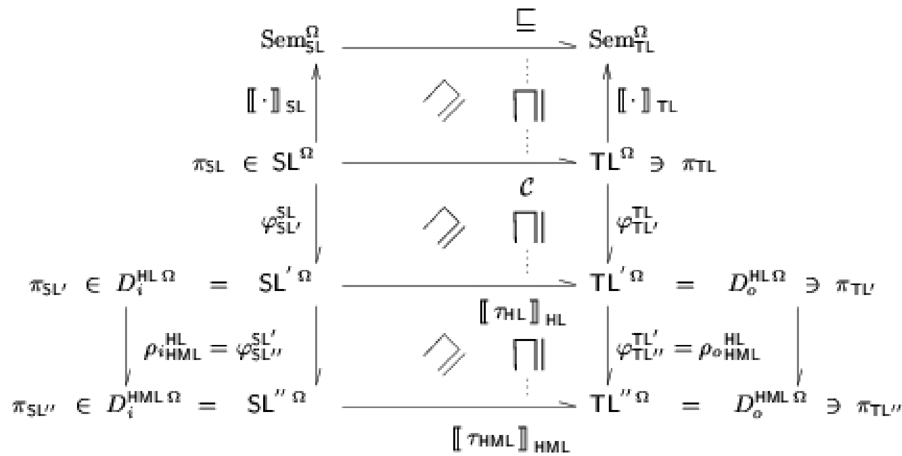


Fig 3. This diagram is again, but now precisely, illustrating the three steps for correct compiler implementation as of Figure 1

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 3 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.

1.4. 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 4.

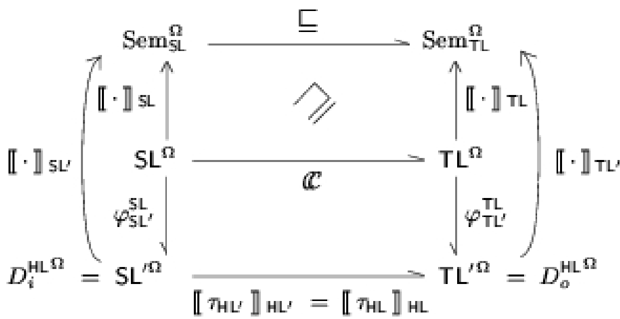


Fig 4. The situation which we will abbreviate by McKeeman's T-diagrams

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 the *T-diagram* (Figure 5)

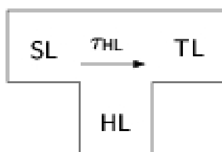


Fig 5. McKeeman's T-diagram as a shorthand for the above situation

as an abbreviation for the diagram in Figure 4. However, we have to keep in mind that the concrete situation is a bit more involved, 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.

2. Related Work on Compiler Implementation Verification

An extended discussion of related work in the field of compiler verification has been presented in our previous article [Goerigk/Langmaack01c]. Recall that we find intensive work on steps 1 and 2, i.e. on compiling specification and on high level compiler implementation verification, although often under unrealistic assumptions so that the results have to be handled with care. Step 3, however, has nearly totally been neglected so far. 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) [Langmaack97b].

Compiling verification is part of theoretical informatics and program semantics, and high level compiler implementation verification (step 2) is a field within software engineering. However, compiler implementation correctness then further depends on existing correct initial host compilers, which are not available up to now. Hackers might have intruded Trojan horses [Thompson84, Goerigk99b] via unverified start up compilers (cf. the introduction to the first article [Goerigk/Langmaack01c] and the following Section 4). This in view, we are finally left on human checkability of mechanical proof protocols (a-posteriori-proof checking) and initial compiler machine code.

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.

3. The Risk of Neglecting Machine Level Verification

Ken Thompson, inventor of the operating system Unix, stated in his Turing Award Lecture "Reflections on Trusting Trust" [Thompson84]:

> "You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code."

He underpinned his statement by sketching the construction of an executable binary machine code version of a C-compiler which was wrong, although his version successfully passed N. Wirth's strong compiler or bootstrap test [Wirth77], which is well-known to be extremely hard to deceive, and although we may even assume that the corresponding C-version of his compiler has been verified.

Let us assume there exists a binary version τ_0 of a C-compiler running on a machine M_0, and a (different) C-compiler τ_1 written in C generating code for a machine M_1. A two step bootstrapping process of τ_1 on M_0 generates a version τ_3 (of τ_1), which is now formulated in binary M_1-machine code. If we assume τ_0 and τ_1 correct, and the machine M_0 to work correctly, then τ_3 is correct as well [Langmaack97a, Goerigk96b, Hoffmann98b].

By a third bootstrapping step of τ_1, we may compile τ_1 to a new M_1-binary τ_4 using the compiler τ_3 on machine M_1. If we assume the machine M_1 to work correctly as well, then τ_4 and τ_3 are identical [Goerigk99a], at least if we assume every involved program deterministic. The third bootstrapping step (and checking identity of τ_3 and τ_4) is N. Wirth's so called strong compiler or bootstrap test. It is employed for safety reasons, if at least one of the four above correctness assumptions cannot be guaranteed.

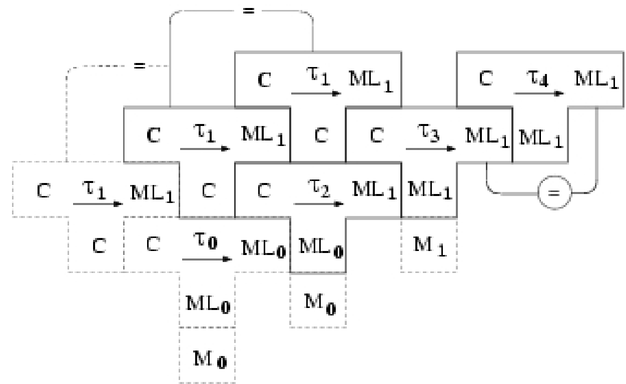


Fig 6. Wirth's strong compiler test. Note: Even if M_0 and M_1 are the same machine, τ_0 and τ_2 need not necessarily generate identical code. τ_0 and τ_2 are two different compilers. In general, we know nothing about τ_0's target code.

But let us come back to Ken Thompson's scenario: He manipulated τ_0, constructed a malicious $\overline{\tau}_0$ that finally produced a wrong $\overline{\tau}_3$, although he followed the entire above bootstrapping process and $\overline{\tau}_3$ passed the strong compiler test. Although the compiler source program τ_1 remains correct (unchanged), and even if the machines M_0 and M_1 work correctly, $\overline{\tau}_3$ incorrectly translates at least one C-source program, in his case the Unix login command. He has introduced a *Trojan Horse* in τ_0, which is a very hidden error hard to detect by tests. The manipulation of τ_0 can even be steered so that $\overline{\tau}_3$ generates incorrect target code for *exactly two* C-source programs [Goe-

rigk99b] — one of which must be the compiler source program τ_1 itself, because otherwise, due to the strong compiler test, $\overline{\tau}_3$ would be correct.

In any case, the crucial insight is that all this might happen even if τ_1 is verified on source level [Goerigk99b, Goerigk00b]. Moreover, if the user would try to convince her/himself of correctness by a test suite, as this is common practice today, she/he would have to find at least one of the two incorrectly compiled programs among those billions of (and theoretically infinitely many) test candidates.

We easily imagine that program validation by test is heavily overcharged in case of compilers if their generation employs *unverified auxiliary* software, like τ_0 in our case. We should well realize the security impact of all this. Virtually every realistic software generation basically depends on running non-verified auxiliary software. Since computers nowadays are usually connected to the world wide net, the software is more or less open to hacker attacks, and might already have been manipulated from outside. Fighting security attacks causes much harder problems than avoiding unintentional bugs (safety). Safety, in a sense, relies on statistically distributed bugs by constructors' mistakes or materials' faults, whereas for security we have to be aware of subversive intent.

Note that one possible procedure in order to uncover the malicious Trojan Horse is to perform the compiler bootstrap and hence use the compiler itself as a test case. But note: this test case would have to be performed very carefully, which means that we have to run the generated compiler $\overline{\tau}_3$, apply it to τ_1 and to compare the result τ_4 with the expected verified result that is given as part of the test suite. We doubt, that any existing compiler binary (like τ_4) has ever been verified in this sense yet. Actually, this is one of the essential tasks of the present paper.

Unless we verify τ_3 to be a correct implementation of (the verified) τ_1, any of the compiler executables τ_0, τ_2, τ_3, τ_4, any further bootstrapped compiler implementation, and any source program compiled by one of these programs might eventually cause disastrous, even catastrophic effects. We think that this well serves as a striking counter example.

4. Realistic Method for Low Level Compiler Generation

The question we are going to answer now is how we attack task A of step 3. A first idea might be to apply software engineering philosophy as for high level implementation verification, i.e. to stepwise refine the high level implementation down to executable binary machine code HML using verified transformation rules. Then, as in the 60´s, we would have developed the entire machine code written compiler by hand, although nowadays supported by using the computer as an efficient typewriter and maybe for checking correct application of the implementation rules. However, as we have seen before (section 4), we ought to have doubts trusting implementations of automatic checking routines as long as there are no *trusted*, i.e. correctly implemented initial compiler executables available. Even if trained in mathematical rigor, no software scientist would ever manually construct or, more importantly, certify, a real world compiler executable in machine code.

Therefore, we follow a different approach, the *Verifix*-idea, to generate even initial compiler implementations by an approved method, thereby incorporating the necessary verifications. We pursue N. Wirth's idea to bootstrap the compiler and add a sufficient variant of the strong bootstrap test [Goerigk/Hoffmann98].

We identify source and high level implementation language SL = HL and take a proper sublanguage ComLisp [Goerigk/Hoffmann96] of ANSI-Common-Lisp = SL^+. We further identify target and low level implementation language TL = HML and take the binary machine code of a concrete processor **M**, e.g. DEC α or INMOS-Transputer-T400. An existing ANSI-Common-Lisp system with

compiler τ_0 is running on a workstation $\mathbf{M}_0$ with machine code $TL_0 = HML_0$, so that our initial two bootstrapping steps together are a cross-compilation to TL-code using the workstation as host machine.

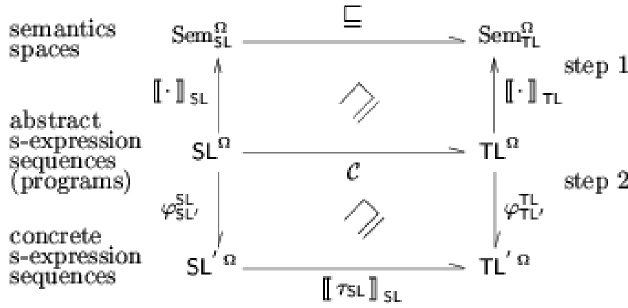


Fig 7. Steps 1 and 2 for an initial full correct compiler implementation

According to step 1 and 2 (section 1.1) we develop a correct SL to TL-compiling specification **C** and correctly implement it as a compiler program τ_{SL}, now in SL = HL itself.

High level syntactical programs in SL' and TL' are SL-data, i.e. s-expression sequences. Abstract syntactical programs in SL and TL are associated to SL', TL' by bijections $\varphi^{SL}_{SL'}$ and $\varphi^{TL}_{TL'}$. We formulate **C** such that it will relate target programs π_{TL} to well-formed source programs π_{SL} at most, i.e. for which source semantics $[[\pi_{SL}]]_{SL}$ are defined. We construct τ_{SL} such that, if applied to well-formed source programs in SL', τ_{SL}, or more precisely $[[\tau_{SL}]]_{SL}$, executed on an imagined SL-machine, will either terminate successfully or abort with an acceptable error due to target machine resource exhaustion. τ_{SL} has a pre-condition that restricts its inputs to representations of well-formed source programs. That is to say, τ_{SL} might not check for well formedness; but then, $[[\tau_{SL}]]_{SL}$ has to have an unacceptable error outcome, perhaps specified non-deterministically, if applied to SL-data that do not represent well-formed SL-programs (see the discussion on $\mathbb{C}$ in [Goerigk/Langmaack01c]).

We will also need a version τ''_{SL} which works on machine representations of s-expressions, for instance on character sequences (strings, cf. Figure 8), which form a particular subset of all s-expression sequences. Then, $\varphi^{SL'}_{SL''} = \varphi^{TL'}_{TL''}$ are the known string representations for s-expressions, and their inverses are single-valued functions (a string denotes at most one s-expression sequence). Note: τ''_{SL} is to be a correct implementation of τ_{SL}. The operator read-sequence (used in τ_{SL} in order to read s-expression sequences) is programmed in τ''_{SL} as a function procedure which reads character sequences (using operators read-char and perhaps peek-char) and returns a list of s-expressions. Analogously the output is programmed using character output, i.e. the operator write-char. $\tau''_{SL''}$ denotes a string version representing τ''_{SL}, i.e. $(\tau''_{SL}, \tau''_{SL''}) \in \varphi^{SL}_{SL''}$.

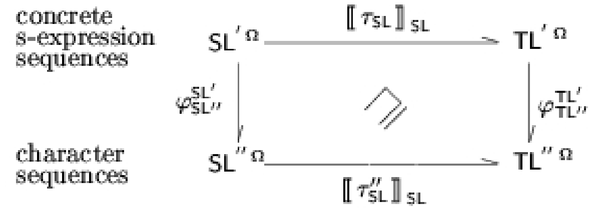


Fig 8. A correct compiler τ''_{SL} mapping character sequences in SL" to character sequences in TL"

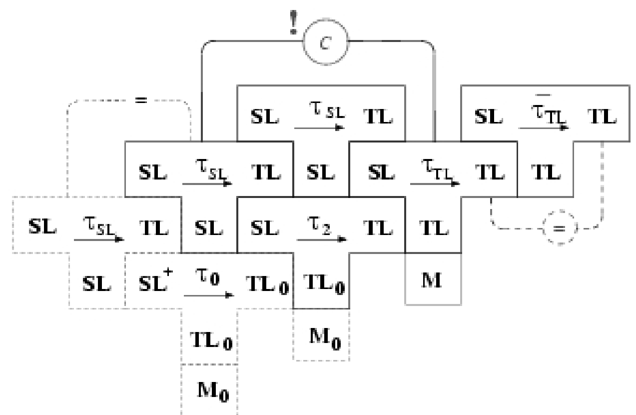


Fig 9. Bootstrapping the initial compiler. A sufficient variant of Wirth's strong compiler test (*a-posteriori-syntactical code inspection*) guarantees fully correct implementation

Let us now (unrealistically) assume for a moment, that τ_0, $\mathbf{M}_0$'s (the work-

station's) Lisp-compiler, has been correctly developed and correctly implemented in TL_0. Then a threefold bootstrapping (Figure 9) of τ_{SL} (often of a byte-sequence or string representation) eventually generates $\overline{\tau}_{TL}$, provided that the three compiler executions of τ_0, τ_2, and τ_{TL} successfully terminate with regular outputs τ_2, τ_{TL}, and $\overline{\tau}_{TL}$, respectively. Since τ_{SL} and τ_0 are assumed correct, and whenever the data abstractions $\rho_i{}^{SL}_{TL}{}^{-1}$, $\rho_o{}^{SL}_{TL}{}^{-1}$, $\rho_i{}^{SL+}_{TL_0}{}^{-1}$, and $\rho_o{}^{SL+}_{TL_0}{}^{-1}$ are single-valued functions, we have the following bootstrapping theorem [Langmaack97a, Goerigk99a, Goerigk00a]:

Theorem 4.1. (Bootstrapping theorem) The compilers τ_2, τ_{TL}, and $\overline{\tau}_{TL}$ are correct, even correctly implemented on hardware processor $\mathbf{M}_0$ resp. $\mathbf{M}$. They are all correct refinements of **C** and τ_{SL}.

In our case $\rho_i{}^{SL}_{TL}{}^{-1}$, $\rho_i{}^{SL+}_{TL_0}{}^{-1}$, $\rho_o{}^{SL}_{TL}{}^{-1}$ and $\rho_o{}^{SL+}_{TL_0}{}^{-1}$ are single-valued functions; actually, they are just the same as $\varphi^{SL'}_{SL''}{}^{-1}$, $\varphi^{SL+'}_{SL''+}{}^{-1}$, $\varphi^{TL'}_{TL''}{}^{-1}$, and $\varphi^{TL'_0}_{TL''_0}{}^{-1}$, respectively. We prove Theorem 4.1 by successive application of a bootstrapping lemma: Let τ_0 and τ_1[1] be two correct compilers and let us apply τ_0 to τ_1:

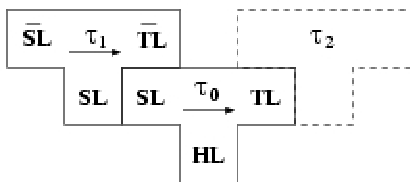


Fig 10. Bootstrapping an initial compiler implementation τ_2

What can we expect in case of a successful compilation with a regular result τ_2 respectively a representation $\tau_{2'}$? The T-diagrams represent two commutative diagrams (Figure 11 for τ_0, and Figure 12 for τ_1).

[1] We use τ_1 here instead of τ_{SL}, because the following argument works for any correct compiler source program.

Moreover, τ_0 is correct, and the result of applying τ_0 to $\tau_{1'}$ is the regular HL-datum $\tau_{2'} \in D_o^{HL\Omega} = TL'^{\Omega}$. Thus, $[[\tau_{1'}]]_{SL'} \sqsubseteq [[\tau_{2'}]]_{TL'}$, which means $\rho_i ; [[\tau_{2'}]]_{TL'} \sqsupseteq [[\tau_{1'}]]_{SL'} ; \rho_o$.

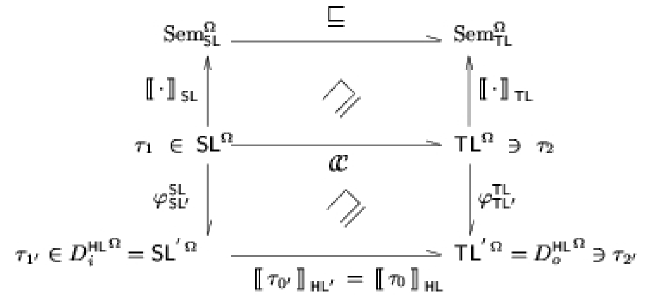


Fig 11. Extended view at Figure 10 (part 1 for τ_0)

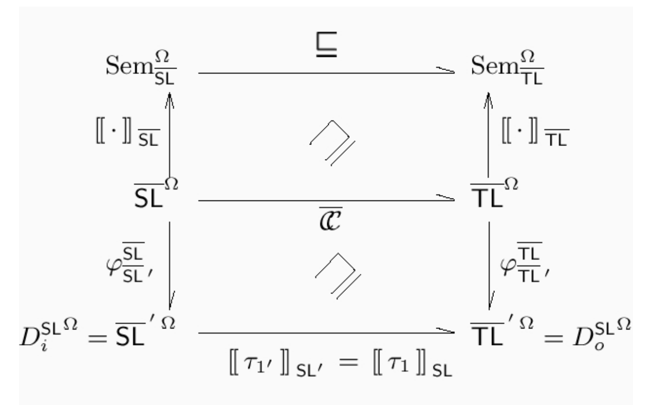


Fig 12. Extended view at Figure 10 (part 2 for τ_1)

Hence, also the diagram in Figure 13 is commutative and, thus, vertical composition of the diagrams in Figure 12 and in Figure 13 finally yields the following bootstrapping lemma:

Lemma 4.2. (Bootstrapping lemma) If $(\rho_i^{-1} ; [[\cdot]]_{\overline{SL}'})$ and $(\rho_o^{-1} ; [[\cdot]]_{\overline{TL}'})$ are reasonable semantics functions (e.g. if ρ_i^{-1}, ρ_o^{-1} are partial functions like in our case using Lisp s-expression sequences), then D_i^{TL} and D_o^{TL} may be conceived to be concretizations $\overline{SL}''$ and $\overline{TL}''$ of the programming languages $\overline{SL}$, $\overline{SL}'$ and $\overline{TL}$, $\overline{TL}'$, and in case of regular termination of $\tau_{0'}$ applied to $\tau_{1'}$, $[[\tau_{2'}]]_{TL'} = [[\tau_2]]_{TL}$ is a correct implementation of $[[\tau_{1'}]]_{SL'} = [[\tau_1]]_{SL}$ and by vertical composition also of $\overline{C}$ (Figure

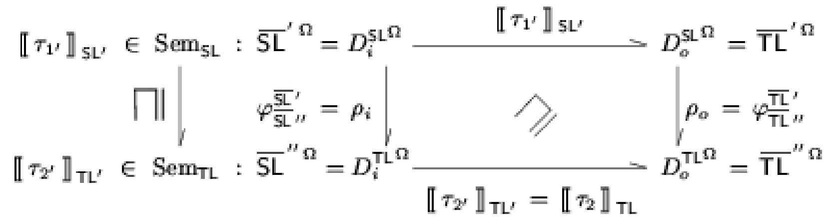


Fig 13. Commutative diagram corresponding to $[\![\tau_{1'}]\!]_{SL'} \sqsubseteq [\![\tau_{2'}]\!]_{TL'}$

12, perhaps different from ℂ). Thus, $\tau_{2'}$ and τ_2 are correct compiler programs with the T-diagram shown in Figure 14.

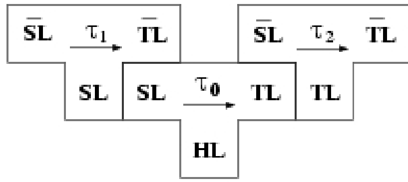


Fig 14. If τ_0 and τ_1 are correct compilers, then so is τ_2

Let us start a minor detour and come back to the discussion in the first article [Goerigk/Langmaack01c]: The proof of the bootstrapping lemma shows, that the (compiler generating) compiler $\tau_{0'}$ needs not be a regularly terminating program for all well-formed input programs in SL' in order to be useful. $\tau_{0'}$, applied to a well-formed program $\pi_{SL'}$ in SL', even to a well-formed compiler like $\pi_{1'}$, might run into an acceptable error in $A_{TL'} = A_o^{HL}$, perhaps due to a resource violation like a memory overflow.

If $\tau_{0'}$ terminates regularly, however, then the associated regular result $\tau_{2'}$ in TL' is again a well-formed and correct $\overline{SL}$ to $\overline{TL}$-compiler. In other words: Since we cannot expect $\tau_{0'}$ to yield regular results for all (arbitrarily large) source program representations, $\tau_{0'}$ will in general be correct in the sense of preservation of relative (partial) program correctness, but not in the sense of preservation of regular (total) correctness [Goerigk/Langmaack01c].

Thus, the expected error behaviors of $\overline{SL}$-programs and in particular of τ_1 and hence also of τ_2, on the one hand, might be of quite a different nature than, on the other hand, of SL and hence in particular of τ_0. $\overline{SL}$ might even be a process programming language such that we expect $\overline{SL}$-programs and their implementations in $\overline{TL}$ to be regularly (totally) correct. If τ_1 preserves regular (total) program correctness, then so will τ_2, i.e. the result of applying τ_0 to τ_1 [Goerigk/Langmaack01c].

That is to say: A partial correctness preserving compiler τ_0 may well generate a total correctness preserving compiler executable τ_2 from a corresponding total correctness preserving compiler source program τ_1. We hence have just given the proof, that there is no need for *trusted* compilers to be correct in the sense of preservation of total correctness. We do not depend on a guarantee of well-definedness while bootstrapping compilers.

This ends our detour and we come back in particular to the strong compiler (bootstrapping) test. Since τ_{SL}, restricted to well-formed SL-programs, is deterministic, we also have the following strong compiler test theorem (variants can also be found in [Langmaack97a, Goerigk99a, Goerigk00a]). Whereas the proof of the bootstrapping theorem (4.1) is a simple application of the bootstrapping lemma, the proof of the strong compiler test theorem requires a more explicit exploitation of the bootstrapping lemma.

Theorem 4.3. (Strong compiler test theorem) The compilers τ_{TL} and $\overline{\tau}_{TL}$ are equal.

Proof. Let τ_{SL} and τ_0 play the roles of τ_1 and τ_0 in the lemma. We take a representation $\tau_{SL'}$ of (the well-formed abstract s-expression) τ_{SL}. $\tau_{SL'}$ is also a rep-

resentation of the abstract SL$^+$-program τ_{SL}. SL and SL$^+$ have the same input and output data domains (of concrete s-expression sequences). τ_{SL} and $\tau_{SL'}$ have the same semantics $[\![\tau_{SL}]\!]_{SL} = [\![\tau_{SL}]\!]_{SL^+} = [\![\tau_{SL'}]\!]_{SL'} = [\![\tau_{SL'}]\!]_{SL^{+'}}$.

Let $\tau_{2'}$ be the concrete result of successfully applying τ_0 to $\tau_{SL'}$ on host machine $\mathbf{M}_0$. Due to the proof of the bootstrapping lemma we have the following commutative diagram:

$$\begin{array}{ccc} SL'^{\Omega} = D_i^{SL^{\Omega}} = D_i^{SL^{+\Omega}} & \xrightarrow{[\![\tau_{SL'}]\!]_{SL^{+'}}} & D_o^{SL^{+\Omega}} = D_o^{SL^{\Omega}} = TL'^{\Omega} \\ \varphi^{SL'}_{SL''} = \rho_i{}^{SL^+}_{TL_0} \downarrow & & \downarrow \rho_o{}^{SL^+}_{TL_0} = \varphi^{TL'}_{TL''} \\ \tau_{SL'} \in SL''^{\Omega} = D_i^{TL_0{}^{\Omega}} & \xrightarrow[{[\![\tau_{2'}]\!]_{TL_0'} = [\![\tau_2]\!]_{TL_0}}]{} & D_o^{TL_0{}^{\Omega}} = TL''^{\Omega} \ni \tau_{TL''} \end{array}$$

Recall that we introduced languages SL" and TL" as reasonable representations of SL, SL' and of TL, TL'. SL and SL' resp. TL and TL' are isomorphic, and the inverses of $\rho_i{}^{SL^+}_{TL_0}$ and $\rho_o{}^{SL^+}_{TL_0}$ are single valued functions.

Let $\tau_{TL''}$ be the concrete successful result of compiling $\tau_{SL'}$ by $\tau_{2'}$ again on the host machine $\mathbf{M}_0$. Again, due to the proof of the bootstrapping lemma we have

$$\begin{array}{ccc} SL'^{\Omega} = D_i^{SL^{\Omega}} & \xrightarrow{[\![\tau_{SL'}]\!]_{SL'}} & D_o^{SL^{\Omega}} = TL'^{\Omega} \\ \varphi^{SL'}_{SL''} = \rho_i{}^{SL}_{TL} \downarrow & & \downarrow \rho_o{}^{SL}_{TL} = \varphi^{TL'}_{TL''} \\ \tau_{SL'} \in SL''^{\Omega} = D_i^{TL^{\Omega}} & \xrightarrow[{[\![\tau_{TL''}]\!]_{TL''} = [\![\tau_{TL}]\!]_{TL}}]{} & D_o^{TL^{\Omega}} = TL''^{\Omega} \ni \bar{\tau}_{TL''} \end{array}$$

Let $\bar{\tau}_{TL''}$ be the concrete successful result of compiling $\tau_{SL'}$ by $\tau_{TL''}$, now on machine $\mathbf{M}$. Since τ_{SL} is assumed deterministic (which actually can be guaranteed by construction), $\tau_{TL''}$ and $\bar{\tau}_{TL''}$ are representations of one and the same concrete s-expression sequence $\tau_{TL'} = \bar{\tau}_{TL'}$ in $D_o^{SL^+} = D_o^{SL} =$ TL' and of $\tau_{TL} = \bar{\tau}_{TL}$ in the abstract language TL. □

We can not prove equality of $\tau_{TL''}$ and $\bar{\tau}_{TL''}$. Equality does in general not hold, because the code of $\tau_{TL''}$ and of $\bar{\tau}_{TL''}$ has been generated by two perhaps different code generation mechanisms of $\tau_{2'}$ on $\mathbf{M}_0$ resp. of $\tau_{TL''}$ on $\mathbf{M}$, i.e. by two different compilers on two different machines. The code $\tau_{TL''}$ is influenced by runtime-system code (in particular by print-routines) which τ_0, e.g. the existing SL$^+$ = ANSI-Common Lisp compiler running on $\mathbf{M}_0$, attaches as part of $\tau_{2''}$ (which generates $\tau_{TL''}$), whereas the code of $\bar{\tau}_{TL''}$ is influenced by runtime-system code which our compiler executable ($\tau_{2'}$) attaches as part of $\tau_{TL''}$ (which generates $\bar{\tau}_{TL''}$).

Let us for instance assume $\tau_{TL''}$ and $\bar{\tau}_{TL''}$ to be character sequence (string) representations of τ_{TL} resp. $\bar{\tau}_{TL}$. Since there are different correct string representations of one and the same s-expression, the $\mathbf{M}_0$-print routine of $\tau_{2'}$ might print $\tau_{TL''}$ differently to how $\tau_{TL''}$ prints $\bar{\tau}_{TL''}$. But nevertheless, both are correct printed representations of the equal s-expressions $\tau_{TL'}$ resp. $\bar{\tau}_{TL'}$.

However, we could add a further bootstrapping step comparing the string results of τ_{TL} and $\bar{\tau}_{TL}$. Since we would then use the same print routines, and because our programs are deterministic, we would finally get equal string representa-

tions $\bar{\tau}_{TL''} = \bar{\bar{\tau}}_{TL''}$, if the bootstrap succeeds. In order to see this, we exploit that τ_{TL}, $\tau_{TL''}$, $\bar{\tau}_{TL}$, $\bar{\tau}_{TL''}$ are semantically equal and that their string representations represent one and the same s-expression. Here is another important remark: The loading mechanism on machine **M** must not depend on different binary (character sequence) representations $\tau_{TL''}$ resp. $\bar{\tau}_{TL''}$ of $\tau_{TL} = \bar{\tau}_{TL}$. The loader has to load the same abstract **M**-machine program in both cases.

In any case, we could also start the entire bootstrapping process by applying τ_0 on $\mathbf{M}_0$ to $\tau''_{SL''}$. Recall that the latter is a version of our compiler source program τ_{SL} which does not use the standard operators read and print of the runtime-system of the implementing compiler to read and print s-expressions, but comprises its own reading and printing routines (implemented by character input/output-routines read-char, peek-char, and write-char). Hence, it transforms character sequences. In that case, both programs are guaranteed to deterministically compute the same sequence of characters, even on different machines. Like in the proof of Theorem 4.3 we could then show equality of the character sequences $\tau''_{TL''} = \bar{\tau}''_{TL''}$, because their representations by $\varphi^{TL'}_{TL''}$ are single-valued.

But note that this is again only true if we compare the printed character sequences. On binary level, they might for instance be represented in different character codings on $\mathbf{M}_0$ and **M**, say in 8bit-ASCII or in 16bit-Unicode.

Let us end this section with the following remark: We are well aware that the previous observations are tedious, cumbersome and by far not obvious. But on the other hand, we have too often seen compilers generating wrong code just because they have been bootstrapped or cross-compiled in a quick-and-dirty process, forgetting about potential code representation problems like for instance byte order, alignment or even different character codes used on different machines. Fortunately, our rigorous mathematical treatment of code generation and compiler bootstrapping enables to uncover any of these tedious problems and to talk about them precisely.

5. Source Level Verification is not Sufficient

Note, that all this has been proved under the unrealistic assumption that τ_0 is correct. But there is no guarantee in the situation we are aiming at: We want to construct and generate an initial (first) correctly implemented compiler executable. Nevertheless, we can use τ_0 and τ_2 as intelligent (and efficient) tools, and they will often succeed to produce the correct result. We just have to assure this fact. This is the key idea of our approach to low level compiler implementation verification.

However, the successful bootstrap test ($\tau_{TL} = \bar{\tau}_{TL}$) does not help. It is well-known that it might succeed for incorrect compiler source programs τ_{SL}. Just consider a source language construct, which is incorrectly compiled but not used in the compiler itself.

Our situation is more delicate: τ_{SL} is a verified compiler source program. Unfortunately, we can prove [Goerigk99a, Goerigk99b, Goerigk00b] even in this case: Although $\bar{\tau}_{TL}$ is successfully generated from the verified τ_{SL} by threefold bootstrapping and passes the strong compiler test, τ_{TL} is not necessarily correct. Ken Thompson's Trojan Horse, originally hidden in τ_0, might have survived so that we find it both in τ_{TL} and in $\bar{\tau}_{TL}$ (and also in τ_2).

In [Goerigk99b, Goerigk00b], we prove this fact mechanically using M. Kaufmann's and J Moore's logic and theorem prover ACL2 [Kaufmann/Moore94]. Ken Thompson's Trojan Horse can be expressed in high level language, even in the clean and abstract Boyer/Moore-logic of first order total recursive functions. We need no ugly machine code to construct such a malicious program part. The situation is even more delicate if we consider preceeding or subsequent compilation phases: If only one phase is corrupted, the only chance to uncover that error is

to rigorously check the target code of exactly that phase, while the compiler executable τ_{TL} (or τ_2) compiles τ_{SL}. No other test will help, unless the user by accident runs τ_{TL} on exactly that one additional incorrectly compiled source program that eventually causes catastrophic effects (and waits for the catastrophe to happen).

6. Realistic Method for Low Level Compiler Verification

Let us now drop our assumption that τ_0 is correct. Also note that, in general, programs are non-deterministic. By twofold bootstrapping of τ''_{SL} resp. $\tau''_{SL''}$ on machine $\mathbf{M}_0$ we generate an output string *s* which is supposed to be a string representation of τ''_{TL}. This is according to the first two steps of Figure 9. Since τ_0 and hence τ_2 might be incorrect, we have to make sure with mathematical rigor, that *s* is indeed a representation of τ''_{TL}.

Our method of low level compiler implementation verification is as follows: Let τ''_{TL} be a program written in TL such that τ''_{SL} **C** τ''_{TL} holds, i.e. let τ''_{SL}, τ''_{TL} fulfill the compiling specification which was verified in step 1 (Figure 7). Hence, τ''_{SL}'s and τ''_{TL}'s semantics are related by $\sqsubseteq$.

$$
\begin{array}{ccc}
D_i^{SL\,\Omega} = SL''^{\,\Omega} & \xrightarrow{[\![\tau''_{SL}]\!]_{SL}} & TL''^{\,\Omega} = D_o^{SL\,\Omega} \\
\rho_i{}^{SL}_{TL} \supseteq \varphi^{SL''}_{SL''} \Big\downarrow & \rotatebox{45}{$\sqsupseteq$} & \Big\downarrow \varphi^{TL''}_{TL''} \subseteq \rho_o{}^{SL}_{TL} \\
D_i^{TL\,\Omega} = SL''^{\,\Omega} & \xrightarrow[{[\![\tau''_{TL}]\!]_{TL}}]{} & TL''^{\,\Omega} = D_o^{TL}
\end{array}
$$

Fig 15. Correct implementation of correct compiler source programs. The extended semantics, defined on $D_i^{SL\,\Omega}$ resp. $D_i^{TL\,\Omega}$, carries an unacceptable error outcome in U_i^{SL} resp. U_i^{TL} indicating the cases where inputs do not represent well-formed SL- or TL-programs

$\varphi^{SL''}_{SL''}$, $\varphi^{TL''}_{TL''}$ are the natural 1-1-mappings on character sequences. How $\rho_i{}^{SL}_{TL}$, $\rho_o{}^{SL}_{TL}$ are precisely defined depends on which primitive standard input-output-routines are actually used in SL and TL.

Since $\rho_i{}^{SL}_{TL}{}^{-1} = \rho_o{}^{SL}_{TL}{}^{-1}$ is single-valued, representations of well-formed programs have unique semantics in Sem_{SL} resp. Sem_{TL}. Due to vertical composability (cf. [Goerigk/Langmaack01c]), τ''_{TL} is a correct SL" to TL"- compiler correctly implemented in TL. Figure 15 accomplishes Figure 7 and 8 and yields Figure 1 in our special situation.

Any of our data including programs are representable by s-expressions. Thus, we may assume that any of our source and target languages L have s-expression-syntaxes, i.e. syntactical programs are s-expression sequences. Input and output data domains D_i^L, D_o^L are sets of s-expression sequences as well. Note that characters are particular s-expressions, and hence character sequences (strings) are particular s-expression sequences. Not every syntactical program (s-expression) is well-formed. In fact, the set of well-formed programs is exactly the domain of definition of the semantics function $[\![\cdot]\!]_L \in \mathsf{Sem}_L$.

Compiling specifications **C** are often defined by *syntactical* rules (e.g. by term rewriting), whereas correctness of τ''_{TL} is a *semantical* matter. That is to say: We have a reduction of the correctness problem from semantics to syntax. The previous paragraphs sketch the proof of the following theorem:

Theorem 4.4. (Semantics to Syntax Reduction) If *s* is a string, if $\tau''_{TL} = (\varphi^{TL}_{TL''};\ \varphi^{TL''}_{TL''})^{-1}(s)$ and if syntactical checking of τ''_{SL} **C** τ''_{TL} is successful, then τ''_{TL} is a well-formed SL" to TL" compiler correctly implemented in TL.

It is not in all cases necessary to completely verify an algorithm beforehand in order to trust a computed result. In the proposed process for correct compiler construction, we verify τ_{SL} resp. τ''_{SL} a-priori (steps 1 and 2), but verification of τ_{TL} resp. τ''_{TL} (step 3) is an a-posteriori syntactical result checking. It allows for using unverified supporting software, e.g. compilers τ_0 and τ_2 on machine $\mathbf{M}_0$. They are used as intelligent but not necessarily correct type writers. Checking guarantees to find any error in τ''_{TL}, even intended errors like viruses or Trojan horses as of section 4.

The idea of a-posteriori result checking is old. We can find applications e.g. in high school mathematics, like checking division or linear equation solving by (matrix-vector) multiplication. The idea has found its way to algorithms theory [Blum+89], trusted compilation [Langmaack97b, Langmaack97c, Goerigk/Hoffmann98, Pnueli+98, Heberle+98, Gaul+99, Cimatti+97], and systems verification [Goerigk+98, Bartsch/Goerigk00d] in general.

6.1. Realistic Syntactical a-posteriori Code Inspection. However, since we know that realistic compiling specifications and compilers are of tangible size, we might ask if syntactical a-posteriori code checking is realistically manageable. A first idea might be to look for machine support, i.e. to write checking algorithms and programs. But we should be aware that in this way we might well run into *circulos vitiosos*. We burden ourselves with new specification and (high and machine level) implementation correctness problems for checking algorithms and programs. If we want to implement an initial compiler fully correctly on a machine, there is no way around some hand checking.

Remark: We do not condemn the use of programmed computers for proving and proof checking. If a software engineer believes in auxiliary software to be sufficiently trustful, she or he is allowed to use it in order to gain more reliable software production. However, the software engineer should then make clear which parts of the auxiliary software he/she has used and hence relies on although still not being rigorously verified modulo hardware correctness. We have shown techniques how to maliciously harm auxiliary software. It is most important for the IT-community to demonstrate sound and realistic means how to stop such ever lasting circulos vitiosos. Since source code verification may succeed, and since manual machine code verification hardly ever will do, we strongly believe that providing trustworthy compiler executables is the most promising sound basis.

Verifix has introduced [Goerigk/Hoffmann98, Hoffmann98b] three intermediate languages between SL (ComLisp) and TL (INMOS Transputer- or DEC α-code). Because it is necessary to finally produce a convincing complete rigorous proof document, these languages have particularly been chosen in order to isolate crucial compilation steps and to enable code inspection by target to source code comparison. Essential characteristics and advantages for code inspection are:

- Languages L_i are close to their preceeding languages L_{i-1} so that only few crucial translation steps are necessary per pass.
- Translation uses standard techniques, does only moderately expand and is local in the sense that it does not reorder corresponding thick code pieces.
- We avoid optimization; every transition remains well recognizable and locally checkable w.r.t. $C^{L_i}_{L_{i+1}}$ by juxtaposing corresponding code.
- Every language has a procedure or subroutine concept; source and target programs are modularized by corresponding subroutines.

These characteristics will be reflected by our checking (i.e. compiling specification and code inspection) rules in section 6.2 below. Source, intermediate and target languages are:

1. High level source language is ComLisp = SL. Programs consist of non-nested mutually recursive function procedures with call-by-value parameter passing. Variables are simple, and data are Lisp-s-expressions. Denotational, operational copy rule resp. stack semantics are well-known [Loeckx/Sieber87, Nielson/Nielson92].

2. Stack Intermediate Language SIL. Programs consist of non-nested mutually recursive parameterless procedures. Data remain s-expressions. Operators are postfixed (reverse Polish notation), parentheses are dropped, and variables are represented by frame-pointer based stack locations (usually very small relative addresses) intended to implement proce-

dure and operator parameters. Operational stack semantics is straight forward and easily comparable to the operational SL-semantics.

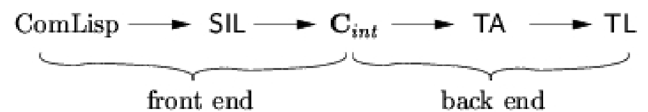
3. C-like intermediate language C_{int} similar to Java's virtual machine language. All variables are of type integer, contents are either immediate or references into two linear stack resp. heap arrays. The stack is intended to implement SIL's stack, and the heap to refine non-atomic Lisp-s-expressions (SIL-data). Every SIL-program can be implemented in C_{int} with equivalent semantics.

4. Assembly language TA. Instructions are machine dependent, e.g. Transputer or DEC α. Symbolic addresses are avoided; subroutines are called using unique numbers, variables have small relative addresses, branches stay within subroutine bodies and are instruction counter relative.

5. Machine code TL. Binary or hexadecimal notation of byte contents with more or less implicit prescription of how to load registers and memory of the target machine. The implicit prescription is materialized by a small boot program [Goerigk/Hoffmann97]. Only TA and TL are machine dependent.

Semantics of a TL-program π_{TL} is given by execution of the machine **M**, after the instruction counter has been loaded with the start address of the main part of π_{TL}. Memory cells and registers not explicitly mentioned in the loading process are assumed to contain arbitrary data, i.e. π_{TL} might behave non-deterministically, although each instruction works deterministically. π_{TL} might in general even overwrite itself. However, the programs we generate will not. In case we prove preservation of partial correctness, they will instead stop with an error message like "stack overflow", "heap overflow", "return stack overflow" or "arithmetic overflow", or due to operator undefinedness. This is guaranteed by compiling specification verification (step 1).

6.2. A Closer Look into a-posteriori Code Inspection. In the following we refer to our concrete compiler implementation from SL = ComLisp to binary Transputer machine code TL [Goerigk/Hoffmann98, Goerigk/Hoffmann97, Goerigk/Hoffmann98b]. The compiler proceeds in four separate phases. Each phase is correctly implemented in ComLisp and generates an external string representation of the intermediate and target programs.



Checking the entire transformation of a Lisp-program directly into binary Transputer-machine code is unrealistic. We would have to check, that the hexadecimal representation of the code for e.g. a function definition like

```
(DEFUN f (x y) (+ (* x y) 3))
```

which compiles into the Transputer-machine code

```
(33 z 4a
 75 e0 73 75 e1 73 fa d3 75 52 d5 75 74 f9 a2 21 f0 73 58 71
 f9 a2 21 f0 73 30 73 e4 73 31 73 e5 73 32 73 e6 73 33 73 e7
 44 70 21 3e f6 43 73 e6 43 73 e7 44 70 21 3c f6 73 34 73 e0
 73 35 73 e1 75 60 5e d5 75 31 d3 75 30 f6)
```

is a correct one. Without any further structure of the target code we would not be able to do this conscientiously. The vertical decomposition into intermediate languages gives the necessary structure. We will show this using the concrete output of our compiler implementation for the above function. It has no higher control structures (no loops nor conditionals). This slightly simplifies the presentation here because we will not have to check relative jump distances for the assembly code.

6.2.1. Checking the Front End. The first two compilation steps are machine independent. We start with our original ComLisp-function and compile it to SIL. This essentially is the transfor-

mation of expressions into postfix form. The body is compiled to `(x y * 3 +)`, augmented by relative positions of variables and intermediate results. The last statement copies the result to the result stack position 0.

```
(DEFUN f (x y)          (DEFUN F
        (+
         (*
          x                (_COPY 0 2)
          y                (_COPY 1 3)
         )                 (* 2)
         3                 (_COPYC 3 3)
        )                  (+ 2)
      )                    (_COPY 2 0))
```

Let us cite the compiling specification rules which are necessary to check that this particular source to target code transformation has been computed according to the specification. Note that the software engineer does not even have to understand the code semantically in order to check this step[1]. The purely syntactical (but semantically verified) compiling specification defines checking rules. An average educated software engineer will not be overtaxed and can obey them in an informal but nevertheless clear, succinct and rigorous mathematical proof style.

1. $\mathbf{CL}_{\mathrm{def}}[\![$ `(DEFUN` $p\,(p_1 \ldots p_k)\; f_1 \ldots f_m)]\!]_{\gamma} \;\supseteq_{\mathrm{def}}$
   $\supseteq_{\mathrm{def}}$ `(DEFUN p`
   $\mathbf{CL}_{\mathrm{progn}}[\![f_1 \ldots f_m]\!]_{\rho,\gamma,k}$
   `(_COPY` k `0))`
   where $\rho(p_i) = i\text{-}1$ for each $i = 1, \ldots, k$
2. $\mathbf{CL}_{\mathrm{progn}}[\![f_1 \ldots f_m]\!]_{\rho,\gamma,k} \;\supseteq_{\mathrm{def}}\; \mathbf{CL}_{\mathrm{form}}[\![f_1]\!]_{\rho,\gamma,k}$
   ...
   $\mathbf{CL}_{\mathrm{form}}[\![f_m]\!]_{\rho,\gamma,k}$
   where $m \geq 1$
3. $\mathbf{CL}_{\mathrm{form}}[\![(p\;\; f_1 \ldots f_n)]\!]_{\rho,\gamma,k} \supseteq_{\mathrm{def}} \mathbf{CL}_{\mathrm{form}}[\![f_1]\!]_{\rho,\gamma,k}$
   ...
   $\mathbf{CL}_{\mathrm{form}}[\![f_n]\!]_{\rho,\gamma,k+n-1}$
   `(p` k`)`
4. $\mathbf{CL}_{\mathrm{form}}[\![c]\!]_{\rho,\gamma,k} \;\supseteq_{\mathrm{def}}$ `(_COPYC c` k`)`
   where c is a constant integer, character, string or symbol `NIL` or `T`
5. $\mathbf{CL}_{\mathrm{form}}[\![v]\!]_{\rho,\gamma,k} \;\supseteq_{\mathrm{def}}$ `(_COPY` $\rho(v)$ k`)`
   where v is a local variable or formal parameter with $\rho(v)$ a defined natural number

We present the compiling specification rules in the style of a conditional term rewrite system (for details see [Goerigk/Hoffmann97, Hoffmann98b]). Ground terms are s-expressions or s-expression sequences from the syntactical domains of source and target language, i.e. of ComLisp and SIL: `<program>`, `<declarations>`, `<form>`, `<fname>`, `<ident>`, `<operator>`, `<symbol>`, <integer>, `<character>`, `<string>` resp. `<program>`$_{\mathrm{SIL}}$, `<declarations>`$_{\mathrm{SIL}}$, `<form>`$_{\mathrm{SIL}}$. Ground terms are augmented by rewrite variables[2] and unary rewrite operators like $\mathbf{CL}_{\mathrm{def}}[\![\cdot]\!]$, $\mathbf{CL}_{\mathrm{progn}}[\![\cdot]\!]$ or $\mathbf{CL}_{\mathrm{form}}[\![\cdot]\!]$ with parameters ρ, γ, k. Actually, ρ contains relative addresses for local variables and parameters, γ maps global variables to "absolute" addresses, and k is the relative result position corresponding to the structural depth of source expressions. We just presented those specification rules necessary for our example.

The system of all conditional term rewrite rules together defines multivalued (non-deterministic) operations associated to each rewrite operator, and we understand the single rules above to specify that the left hand side ground term set contains the right hand side set of ground terms *by definition* ($\supseteq_{\mathrm{def}}$). This is an *inclusion by definition*, because there might be other rules which apply to the same left hand side pattern.

The simple structure of these rules guarantees a simple checking process because of their compositionality, order preservation, at most linear expansion and because rewrite operator applications are not nested and procedure boundaries are preserved.

---

[1] Nevertheless, we will sometimes give comments on the semantics in order to make this presentation more intuitive and readable.

[2] We use the prefix *rewrite* in order to distinguish rewrite variables from those ranging over program fragments and rewrite operators from program operators.

The next step is data refinement of dynamically typed Lisp-data to a linear memory architecture. Relative addresses are multiplied by 2 (tag and value field) and copied in pairs. We have to focus on single SIL statements and compare them with pairs of target statements: In order to copy the content of *x* from relative position 0 to 2, the target code has to copy the tag field from 0 to 4 and the value field from 1 to 5. Operator calls now become subroutine calls into the runtime system — compiling specification verification proves that the runtime system procedures are correct operation refinements of the SIL-operators.

```
(DEFUN F      (DEFUN F (8)
(_COPY 0 2)    (_SETLOCAL (_LOCAL 0) 4)
               (_SETLOCAL (_LOCAL 1) 5)
(_COPY 1 3)    (_SETLOCAL (_LOCAL 2) 6)
               (_SETLOCAL (_LOCAL 3) 7)
(* 2)          (* 4)
(_COPYC 3 3)   (_SETLOCAL 3 6)
               (_SETLOCAL 3 7)
(+ 2)          (+ 4)
(_COPY 2 0))   (_SETLOCAL (_LOCAL 4) 0)
               (_SETLOCAL (_LOCAL 5) 1))
```

Again, we cite the corresponding conditional term rewrite rules from the compiling specification from SIL to C_{int} (ξ is a compiletime environment mapping program constants such as symbols, strings, and lists to linear heap representations and addresses):

1. $\mathbf{CS}_{def}$ [[(DEFUN p $f_1 \ldots f_m$)]]$_\xi$ $\supseteq_{def}$
   $\supseteq_{def}$ (DEFUN p (s)
   $\mathbf{CS}_{form}$ [[f_1]]$_\xi$
   ...
   $\mathbf{CS}_{form}$ [[f_m]]$_\xi$)
   where s is the maximal stack frame length needed by $f_1, \ldots, f_m$
2. $\mathbf{CS}_{form}$ [[(_COPY i j)]]$_\xi$ $\supseteq_{def}$
   $\supseteq_{def}$ (_SETLOCAL (_LOCAL $2i$) $2j$)
   (_SETLOCAL (_LOCAL $2i+1$) $2j+1$)
3. $\mathbf{CS}_{form}$ [[(p i)]]$_\xi$ $\supseteq_{def}$ (p $2i$)
4. $\mathbf{CS}_{form}$ [[(_COPYC n i)]]$_\xi$ $\supseteq_{def}$
   $\supseteq_{def}$ (_SETLOCAL τ $2i$)
   (_SETLOCAL n $2i+1$)
   where τ is the number tag 3 and n is an integer

This completes checking the machine independent front end. The next two steps are machine dependent. The final step generates the machine code above.

6.2.2. Checking the Back End. The first back end phase transforms control structure into linear assembly code with relative jumps. The generated subroutine body consists of procedure entry code, the main part and procedure exit code, three parts which are structured in three lists in the TA-code. Entry and exit code share the same pattern for every procedure. This phase also handles resource restrictions of the concrete 32-bit machine. But this is a semantical issue not to be checked here.

In order to check the main part, we have to compare single C_{int}-instructions with small groups of up to four or five assembly instructions. For instance, the instruction (SETLOCAL (LOCAL 0) 4) (the second line in the C_{int}-definition) is compiled to the instruction sequence LDL 3 LDNL 0 LDL 3 STNL 4 (first line of the TA-main part below). It first loads the *frame pointer*, then the content of relative position 0, which after loading the frame pointer again is finally stored into relative position 4.

```
(DEFUN F (8)                  (_DEFCODE F 51
                               (LDL 5 STNL 0 LDL 3 LDL 5
                               STNL 1 LDL 3 OPR 10 STL 3
                               LDL 5 LDNLP 2 STL 5 LDL 5
                               LDL 4 OPR 9 CJ 2 OPR 16 LDL 3
                               LDNLP 8 LDL 1 OPR 9 CJ 2
                               OPR 16)
(_SETLOCAL (_LOCAL 0) 4)       (LDL 3 LDNL 0 LDL 3 STNL 4
(_SETLOCAL (_LOCAL 1) 5)       LDL 3 LDNL 1 LDL 3 STNL 5
(_SETLOCAL (_LOCAL 2) 6)       LDL 3 LDNL 2 LDL 3 STNL 6
(_SETLOCAL (_LOCAL 3) 7)       LDL 3 LDNL 3 LDL 3 STNL 7
```

```
    (* 4)                               LDC 4 LDL 0 LDNL 30 OPR 6
    (_SETLOCAL 3 6)                     LDC 3 LDL 3 STNL 6
    (_SETLOCAL 3 7)                     LDC 3 LDL 3 STNL 7
    (+ 4)                               LDC 4 LDL 0 LDNL 28 OPR 6
    (_SETLOCAL (_LOCAL 4) 0)            LDL 3 LDNL 4 LDL 3 STNL 0
    (_SETLOCAL (_LOCAL 5) 1)            LDL 3 LDNL 5 LDL 3 STNL 1)
                                        (LDL 5 LDNLP -2 STL 5 LDL 5
                                        LDNL 1 STL 3 LDL 5 LDNL 0
         )                              OPR 6))
```

The involved rules of the compiling specification[1] from C_{int} to TA are the following:

1. $\mathbf{CC}_{\mathrm{def}}[\![(\mathtt{DEFUN}\, f\,(\boxed{\sigma})\; s_1 \ldots s_n)]\!]_{\varphi} \supseteq_{\mathrm{def}}$

   $\supseteq_{\mathrm{def}}$ (DEFCODE f $\psi(f)$
   ($entrycode$ ($\boxed{\sigma}$))
   ($\mathbf{CC}_{\mathrm{stmt}}[\![s_1]\!]_{\varphi,\sigma}$
   ...
   $\mathbf{CC}_{\mathrm{stmt}}[\![s_n]\!]_{\varphi,\sigma}$)
   ($exitcode$)

   where $\varphi = \langle\psi, |stack|, |heap|\rangle$ and ψ is a subroutine numbering
2. $\mathbf{CC}_{\mathrm{stmt}}[\![(f\; i)]\!]_{\varphi,\sigma} \supseteq_{\mathrm{def}}$

   $\supseteq_{\mathrm{def}}$ LDC i LDL $start$ LDNL $\psi(f)$ OPR 6
   where $0 \le i < \sigma$
3. $\mathbf{CC}_{\mathrm{stmt}}[\![(_\mathtt{SETLOCAL}\; e\; i)]\!]_{\varphi,\sigma} \supseteq_{\mathrm{def}}$

   $\supseteq_{\mathrm{def}}$ $\mathbf{CC}_{\mathrm{expr}}[\![e]\!]_{\varphi,\sigma}$ LDL $base$ STNL i
   where $0 \le i < \sigma$
4. $\mathbf{CC}_{\mathrm{stmt}}[\![(_\mathtt{LOCAL}\; i)]\!]_{\varphi,\sigma} \supseteq_{\mathrm{def}}$

   $\supseteq_{\mathrm{def}}$ LDL $base$ LDNL i
   where $0 \le i < \sigma$

In the first rule, σ is the stack frame length of f, and we used $\boxed{\sigma}$ (see also the above code) to stress that the procedure entry code is nearly constant, i.e. only parameterized by the number σ. $|stack|$ and $|heap|$ denote the initial stack and heap size. In the second rule, i is the relative address of the return value position, *start* contains the jump table start address, $\psi(f)$ denotes the (constant) jump table position of f's start address, and `OPR 6` is the `GCALL` operation, i.e. the subroutine jump. In the third and fourth rule, *base* contains the "absolute" stack frame base address and i is the relative address of the variable to be assigned to respectively loaded from.

Finally, it is easy to check that the TA-mnemonics and decimal operands have been transformed correctly to instruction byte sequences (cf. Figure 16). Note, that in order to understand the code semantically, we additionally would have to know the semantics for instance of the Transputer-operations called by `OPR` instructions. We *need not* know this information to perform syntactical checking; the correctness of the code follows from compiling verification, whereas we have to check the code for compliance with the specification. The TL-module number `#x33 = 51` below is a code module (indicated by the character `z`) that has a length of `#x4a = 74` bytes.

```
    (_DEFCODE F 51                          (33 z 4a
    (LDL 5 STNL 0 LDL 3 LDL 5                75 e0 73 75
    STNL 1 LDL 3 OPR 10 STL 3                e1 73 fa d3
    LDL 5 LDNLP 2 STL 5 LDL 5                75 52 d5 75
    LDL 4 OPR 9 CJ 2 OPR 16 LDL 3            74 f9 a2 21 f0 73
    LDNLP 8 LDL 1 OPR 9 CJ 2                 58 71 f9 a2
```

[1] The mechanical correctness proof of the corresponding compiling specification [Dold+02] using the PVS theorem prover unvealed a tedious bug in the procedure initialization code. Although easy to repair, we prefer to leave the bug in the code and make this remark instead. The `OPR 10` (`WSUB`, word subscript) in the third line of the code above is an address calculation which indeed does not check for overflow. Finding this bug is due to the mechanized compiling verification by using incorruptible and inexhaustible theorem prover support. Since it is a relative address calculation, it would hardly show up in any practical test run.

```
OPR 16)                          21 f0
(LDL 3 LDNL 0 LDL 3 STNL 4       73 30 73 e4
LDL 3 LDNL 1 LDL 3 STNL 5        73 31 73 e5
LDL 3 LDNL 2 LDL 3 STNL 6        73 32 73 e6
LDL 3 LDNL 3 LDL 3 STNL 7        73 33 73 e7
LDC 4 LDL 0 LDNL 30 OPR 6        44 70 21 3e f6
LDC 3 LDL 3 STNL 6               43 73 e6
LDC 3 LDL 3 STNL 7               43 73 e7
LDC 4 LDL 0 LDNL 28 OPR 6        44 70 21 3c f6
LDL 3 LDNL 4 LDL 3 STNL 0        73 34 73 e0
LDL 3 LDNL 5 LDL 3 STNL 1)       73 35 73 e1
(LDL 5 LDNLP -2 STL 5 LDL 5      75 60 5e d5 75
LDNL 1 STL 3 LDL 5 LDNL 0        31 d3 75 30
OPR 6)))                         f6)
```

The involved compiling rules from TA to TL are:

1. $\mathbf{CA}_{\text{def}}[\![\texttt{(DEFCODE f i b)}]\!] \sqsupseteq_{\text{def}}$ $\sqsupseteq_{\text{def}}$ $(\mathrm{i}_{\#x}\ \mathrm{z}\ \ |\mathrm{c}|_{\#x}\ \mathrm{c})$
   where $\mathrm{c} = \mathbf{CA}_{\text{body}}[\![\mathrm{b}]\!]$ and $\mathrm{i}_{\#x}$, $|\mathrm{c}|_{\#x}$ denote hexadecimal representations of $\mathrm{i}, |\mathrm{c}|$
2. $\mathbf{CA}_{\text{body}}[\![op_1\ e_1\ \ldots(\ldots)\ \ldots(\ldots\ op_n\ e_n)]\!] \sqsupseteq_{\text{def}}$ $\sqsupseteq_{\text{def}}$ $\mathbf{CA}_{\text{opr}}[\![op_1\ e_1]\!] \ldots \mathbf{CA}_{\text{opr}}[\![op_n\ e_n]\!]$
3. $\mathbf{CA}_{\text{opr}}[\![op\ \ e]\!] \sqsupseteq_{\text{def}}$ $prefix\,(assemble_{op}\,(op, e)$

In the second rule, $\mathbf{CA}_{\text{body}}$ ignores the list (parentheses) structure, and in the third rule we apply two auxiliary functions: $assemble_{op}$ translates the 16 basic transputer instruction mnemonics to hexadecimal digits '0' up to 'f' according to the table in Figure 16, and *prefix* generates the `pfix`/`nfix`-chains necessary to load the value of *e*, which is in particular very easy for small non-negative numbers between 0 and 15 (representable by a four bit nibble).

That is to say: In order to check the final code generation step, we only need to know the 16 instruction mnemonics, their mapping to instruction code nibbles, and the `pfix`/`nfix`-chains necessary to load large operands. So for instance `LDC 4` is transformed to 44 which loads the constant 4 into Areg, whereas `LDNL 28` is compiled into the `pfix`-chain `21 3c`, which will execute `LDNL` on $16 \times 1 + 12 = 28$.

6.2.3. The Complete Proof Structure. This ends the more detailed look into the characteristics of our technique

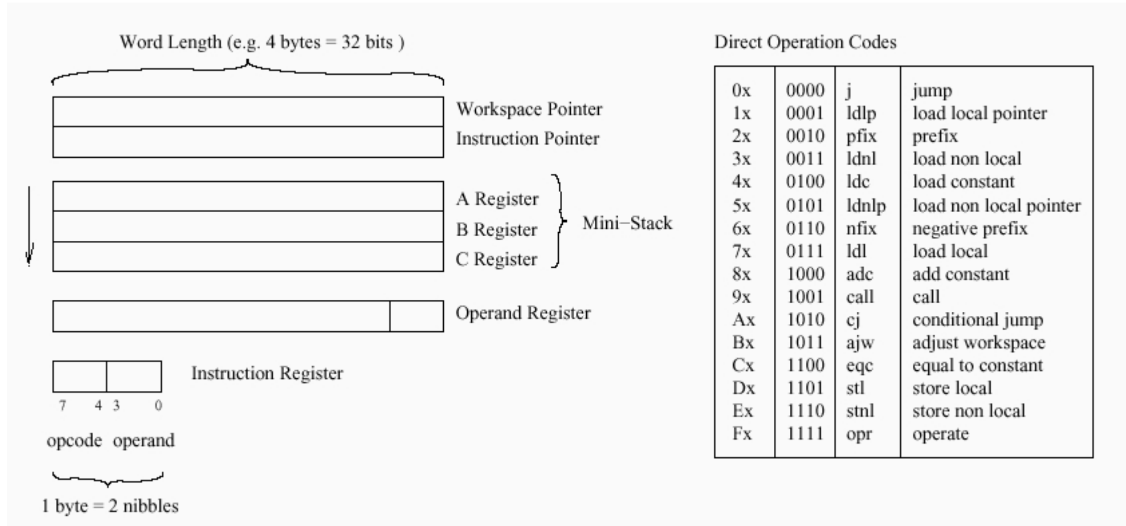


Fig 16. Transputer-architecture and direct function codes. The Transputer state consists of the registers Areg, Breg, and Creg, which form a mini stack with top Areg, the operand register Oreg, the instruction pointer (program counter) Iptr, the workspace pointer Wptr, various flags like the ErrorFlag, some more registers and the memory Mem. The registers contain Word valued quantities. The memory is byte or word addressable

of a-posteriori code inspection by comparing corresponding code parts of the respective source and target programs and checking them to be in conformance with the compiling specification rules.

In order to come back to the overall proof structure, note that program fragments like those of the previous section have been generated for the entire compiler using four unsafe initial compiler implementations produced by τ_0 on machine $\mathbf{M}_0$ for both front-end and both back-end phases.

The following large diagram (Figure 17) shows all four sub-compiling specifications $\mathbf{C}^{\mathsf{L}_{i-1}}_{\mathsf{L}_i}$, all four hand-written sub-compiler implementations $\tau''_{i,\mathsf{SL}} = \tau''_{i,\mathsf{L}_1}$ and all $16 = 4 \times 4$ sub-compilers τ''_{i,L_j} generated and printed out by bootstrapping. The specifications are verified by compiling verification (step 1), the $\tau''_{i,\mathsf{SL}}$ by high level implementation verification (step 2) and the τ''_{i,L_j} for $j > 1$ by low level implementation verification (step 3), actually by checking

$$\tau''_{i,\mathsf{L}_{j-1}} \quad \mathbf{C}^{\mathsf{L}_{j-1}}_{\mathsf{L}_j} \quad \tau''_{i,\mathsf{L}_j}$$

which is exactly what we sketched in the previous section and which we proved to be sufficient due to the semantics to syntax reduction Theorem 4.4.

However, it is not necessary to perform all those cumbersome checkings. In particular, unpleasant low level machine) code inspections below the diagonal are redundant: Since $\mathbf{C}^{\mathsf{TA}}_{\mathsf{TL}}$ and $\tau''_{4,\mathsf{SL}}$ are correct (steps 1 and 2), and since $\tau''_{4,\mathsf{TL}}$ is checked to be correctly compiled to TL, $\tau''_{4,\mathsf{TL}}$ is a fully correct TA to TL-compiler executable. We can use it to correctly compile $\tau''_{3,\mathsf{TA}}$ to a correct $\tau'''_{3,\mathsf{TL}}$, which guarantees $\tau'''_{3,\mathsf{TL}}$ to be a fully correct C_{int} to TA-compiler executable, and so forth.

That is to say: We load the correct compiler $\tau''_{4,\mathsf{TL}}$ (actually $\tau''_{4,\mathsf{TL}''}$) into machine $\mathbf{M}$ using the boot program. Correctness of that program means and hence guarantees that it follows the explicit and implicit loading prescriptions of $\tau''_{4,\mathsf{TL}''}$. Then we start the loaded compiler in $\mathbf{M}$ and let $\mathbf{M}$ read $\tau''_{3,\mathsf{TA}''}$. If $\mathbf{M}$ terminates successfully (regularly), then due to Bootstrapping Theorem 4.1 the output is a correct compiler $\tau'''_{3,\mathsf{TL}}$ (actually $\tau'''_{3,\mathsf{TL}''}$) as well, not necessarily identical to $\tau''_{3,\mathsf{TL}}$, but according to $\mathbf{C}^{\mathsf{TA}}_{\mathsf{TL}}$.

We can now concatenate $\tau'''_{3,\mathsf{TL}}$; $\tau''_{4,\mathsf{TL}}$ (actually $\tau'''_{3,\mathsf{TL}''}$; $\tau''_{4,\mathsf{TL}''}$) and obtain a correct compiler executable from C_{int}'' to TL'' due to composability of commutative diagrams (cf. [Goerigk/Langmaack01c]). If we proceed, this process will finally generate the desired correct compiler executable from SL'' to TL''.

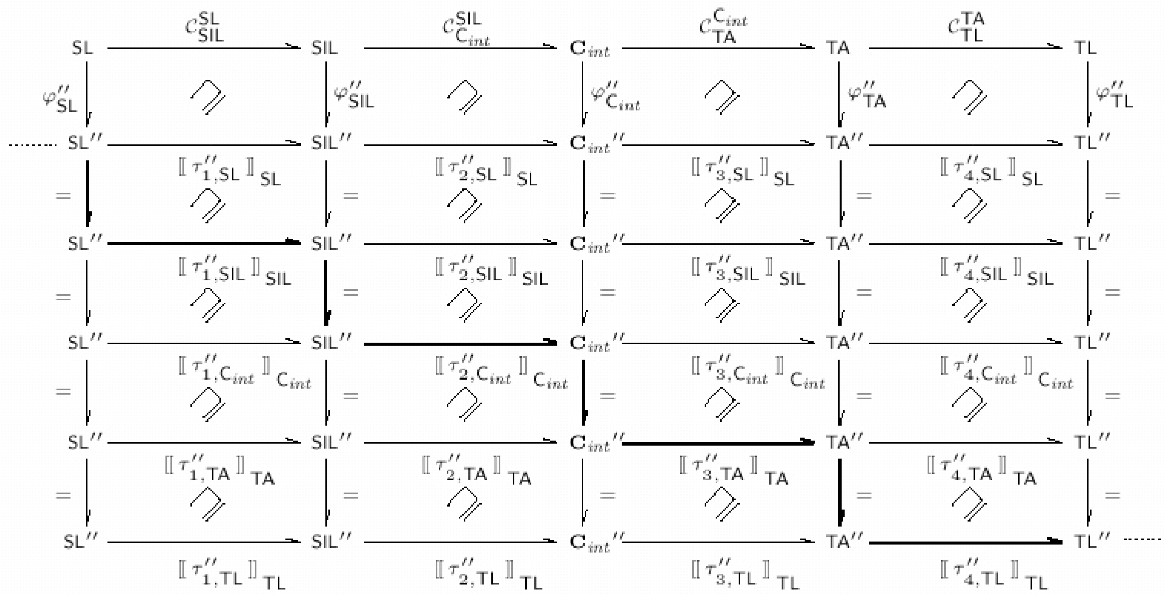


Fig 17. Special bootstrapping with four compiler phases. $\varphi''_{\mathsf{L}} =_{\mathsf{def}} (\varphi^{\mathsf{L}}_{\mathsf{L}'} ; \varphi^{\mathsf{L}'}_{\mathsf{L}''})$

$\tau'''_{3,TL''} = \tau'''_{1,TL''} ; \tau'''_{2,TL''} ; \tau'''_{3,TL''} ; \tau''_{4,TL''}$

Here we exploit *Verifix*'s hardware correctness assumption for verified low level compiler implementation (step 3). Note that we have implicitly introduced a (syntactical) concatenation operator ";" for sequential programs, which corresponds to particular sequential composition. Its semantics is obvious. Any of our languages allows for the syntactical concatenation of programs.

The compilers $\tau''_{i,SL}$ contain a parser for s-expression sequences, namely the implementation of read-sequence. It is part of the runtime system, and so far, it has to be checked down to the diagonal, in particular and unfortunately also as part of $\tau''_{4,TL}$ resp. $\tau''_{3,TA}$ in machine code. However, there is a remarkable chance to reduce the a-posteriori code inspection work load considerably: Since the print-routines are checked down to machine code TL as well, we may in principle check the (considerably larger) read-routines by (trusted) printing of their results, i.e. we may look at the parser read-sequence as an additional initial compiler phase $\tau''_{0,SL}$. Then we only need its correct implementation as a high level SL-program and no further low level implementation verification. In that case, however, unchecked code runs initially, and we need further precaution[1], namely to validate the intermediate machine state after running the code of read-sequence. We can exploit the processor's memory protection mechanism and add a few validations (runtime-checkings) to sufficiently guarantee the relevant part of the intermediate state not to be corrupted. In later work we shall report on this and give the necessary proofs.

7. Conclusions

At the end of our exposй we would like to answer the question raised in the title of this essay: Will informatics be able to justify the construction of large computer based systems? We will trace again the main lines of thought which finally lead us to a rather confident answer: Yes. In fact, internal misbehaviors and intended external violations of computer based systems need not last forever, i.e. safety and security might recover. However, this will not work out unless software production and informatics science carefully enough solve the following problem: At the end it is the executable binary real world processor code, and not only high level specifications and programs, which has to be guaranteed to behave correctly as required.

Realistic software production employs and relies on compilers for high level languages, like for instance C, C++, Ada, Java or Common Lisp. C is very close to machine level, but in our context it must be seen as a high level language with very critical and decisive compilation steps towards real processor code.

Of course, many constructors of realistic commercial and industrial compilers are doing a quite good job. They care about correct compiling specification and correct compiler implementation by high level systems programs or by sophisticated rule sets for term or graph rewrite systems. But there has been bluntly no industry oriented research nor development of techniques to implement high level written compilers such that their executable binary host machine versions are guaranteed to generate (at most) correct target machine code (section 2).

Again and again, the reason for trouble is the use of unverified auxiliary software like tools and in particular compilers, the use of so-called *software of uncertain pedigree* (SOUP, [HSE01]). There are many examples, for instance incorrect implementations of theorem provers or of cryptographic protocols. Actually, even if we (unrealistically would) assume that compiler constructors have been verifying their compiler programs perfectly on source level, compiler implementations have been and are still produced by a now over 40 years lasting unsafe boot-

[1] Unfortunately, it seems again not to be sufficient to compare the initially loaded and the generated code to be identical (cf. section 4).

strapping process using unverified compiler implementations to generate unverified compiler implementations.

In contrast to mathematicians, and also to hardware engineers, software engineers are often not so much impressed by logical gaps, especially not by those evoked by unverified compilers which passed Wirth's strong compiler test. Software constructors like to transfer such logical gaps into Wittgenstein's domain of logical scepticism, arguing that no way of reasoning will ever lead towards convincing solutions (section 2).

But unverified compilers are well outside Wittgenstein's domain and they bear real risks (as shown by Ken Thompson [Thompson84] and later by ourselves [Goerigk99a, Goerigk99b, Goerigk00b]). Meanwhile, since computers use to be connected to world wide networks, we can unfortunately not give any guarantee for any of our programs used (section 4). Actually, rumors say that a computer does not survive un-hacked for more than about eight hours continuous connection to the internet. The risk increases, and computer science will be accounted for providing solutions.

The *Verifix*-project offers industry oriented methods to solve the foundational problem of trusted program implementation, namely to produce realistic initial compilers for diligently chosen but nevertheless realistic programming languages, compilers that run correctly and hence trustworthily on real host processors and generate correct and hence trustworthy binary code for real target processors (cf. [Goerigk/Langmaack01c] and sections 4 to 6). Our particular technique for low level compiler implementation verification is new. It is a sophisticated diagonal method of so-called syntactical a posteriori-code inspection, a variant of rigorous a-posteriori-result checking (section 6).

Once we have got an initial correctly implemented compiler executable, we may safely (mechanically) bootstrap further correct compiler implementations for instance for more comfortable languages (Bootstrapping lemma and theorem, section 4). Even if the bootstrapping compiler preserves partial correctness, it is perfectly able to generate correct compiler executables which preserve total correctness. In fact, this is a very important point for software engineers and process programmers interested in trusted implementation of safety critical embedded real-time systems.

It is by no means necessary, that the language SL we have chosen for trusted compiler bootstrapping is a perfect systems programming language. Compilers for languages with more elaborated data types, nested and even higher order procedures and/or functions, object-orientation and inheritance etc. can safely be bootstrapped. SL and its initial correct compiler implementation are chosen both to be useful tools and to conscientiously provide a high level proof documentation for correct low level binary code generation. Informaticians can check their proof documentation rigorously even without deep mathematical education. Moreover, if we assume hardware to work as described in the instruction manuals, then a lot of unpleasant low level checking is even redundant due to our diagonal technique (section 6).

The usual procedure, i.e. to incrementally step up in a hierarchy of abstractions by constructing and/or implementing higher level (programming or specification) languages safely (correctly) on the lower level, does not work for initial correct compiler executables. The reason is, that machines, their physics, net lists and even their machine languages are too low level in order to adequately express and to conscientiously reason about their program behaviors semantically. Our procedure to drive correctness down towards the real physical machine is, and has to be, of a characteristically different nature, namely to express and to verify realistic correct compilation semantically on the upper level first, and then to bridge the gap towards the real machine "in a big step". Fortunately, it turns out that our techniques still allow to exploit the modularization in

appropriate steps of concretization. In fact, our intermediate languages and the four compiler phases are carefully chosen exactly in order to make this possible.

Theoretical basis of compiler correctness is the notion of correct relative implementation which *Verifix* has developed and which is much more flexible than classical correct implementation (cf. [Goerigk/Langmaack01c] and section 1.1). Software engineering theory stresses preservation of total program correctness, but E. Börger is right in his remark (Boppard, Germany, 1998): "In the past, the role of regular termination has been exaggerated in software engineering." Programs need not be proved never to fail in order to be useful. They might end in acceptable errors, and we are "sensible enough" to give a guarantee that such errors will be signaled and hence detected while programs are executed. On the other hand, if errors cannot be detected, for instance for undecidability reasons, they are unacceptable and thus the user has to avoid (circumvent) them by choosing appropriate inputs. We want to admit that this is kind of sophisticated. However, realistic software engineering requirements are sophisticated trade-offs between a lot of inherently different wishes including for instance efficiency as well. The important point is that our results allow, for the first time in this critical area, to mathematically rigorously formulate and to formalize and prove such realistic requirements to be guaranteed.

It is this line of thoughts that closes a 40 years old gap of low level compiler implementation verification and that makes us confident that informatics eventually will enable to justify the construction of large computer based systems. In particular, *Verifix* also demonstrates techniques to incrementally transfer other so far unverified software into verified software.

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.

[Bartsch/Goerigk00d] R. Bartsch and W. Goerigk. Mechanical a-posteriori Verification of Results: A Case Study for a Safety Critical AI System. In *AAAI Workshop on Model Based Validation of Intelligence MBVI'2001*, Stanford, CA, U.S.A., 2001.

[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 Crypthography*, 1989.

[Cimatti+97] A. Cimatti, F. Giunchiglia, P. Pecchiari, B. Pietra, J. Profeta, D. Romano, and P. Traverso. A Provably Correct Embedded Verifier for the Certification of Safety Critical Software. In *Proceedings of CAV '97 Conference on Computer Aided Verification*, Haifa, Israel, June 1997.

[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.

[Dold+02] A. Dold, W. Goerigk, F. W. von Henke, and V. Vialard. A Mechanically Verified Compiling Specification for a Realistic Compiler. Technical Report UIB 2002-03, University of Ulm, 2002.

[GccBugList] Gcc Project. The Gcc Bugs Archive. http://gcc.gnu.org/ml/gcc-bugs/.

[Goerigk96b] W. Goerigk. An Exercise in Program Verification: The ACL2 Correctness Proof of a Simple Theorem Prover Executable. Technical Report Verifix/CAU/2.4, CAU Kiel, 1996.

[Goerigk97d] W. Goerigk. Towards Rigorous Compiler Implementation Verification. In R. Berghammer and F. Simon, editors, *Proc. of the 1997 Workshop on Programming Languages and Fundamentals of Programming*, pages 118 – 126, Avendorf, Germany, November 1997.

[Goerigk+98] W. Goerigk, Th. Gaul, and W. Zimmermann. Correct Programs without Proof? On Checker-Based Program Verification. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science, pages 108 – 122, Springer Verlag Wien, New York, 1998.

[Goerigk/Hoffmann96] W. Goerigk and U. Hoffmann. The Compiler Implementation Language ComLisp. Technical Report Verifix/CAU/1.7, CAU Kiel, June 1996.

[Goerigk/Hoffmann97] W. Goerigk and U. Hoffmann. The Compiling Specification from ComLisp to Executable Machine Code. Technical Report Nr. 9713, Institut für Informatik, CAU, Kiel, December 1998.

[Goerigk/Hoffmann98b] W. Goerigk and U. Hoffmann. Compiling ComLisp to Executable Machine Code: Compiler Construction. Technical Report Nr. 9812, Institut für Informatik, CAU Kiel, October 1998.

[Goerigk/Hoffmann98] W. Goerigk and U. Hoffmann. Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, volume 1641 of *LNCS*, pages 122 — 136, Springer Verlag, 1998.

[Goerigk/Langmaack01c] W. Goerigk and H. Langmaack. Will Informatics be able to Justify the Construction of Large Computer Based Systems? Part I: Realistic Correct Systems Implementation. International Journal on *Problems of Programming*, 2003. Forthcoming.

[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.

[Goos/Zimmermann99] G. Goos and W. Zimmermann. Verification of Compilers. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *LNCS*, pages 201 — 230. Springer Verlag, 1999.

[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.

[Hoffmann98b] U. Hoffmann. *Compiler Implementation Verification through Rigorous Syntactical Code Inspection*. PhD thesis, Technische Fakultät der Christian-Albrechts-Universität zu Kiel, Technical Report 9814, 1998.

[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.

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

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

[Langmaack97b] H. Langmaack. The ProCoS Approach to Correct Systems. *Real Time Systems*, 13:251—273, Kluwer Academic Publishers, 1997.

[Langmaack97c] H. Langmaack. Contribution to Goodenough's and Gerhart's Theory of Software Testing and Verification: Relation between Strong Compiler Test and Compiler Implementation Verification. *Foundations of Computer Science: Potential-Theory-Cognition. LNCS* 1337, pages 321—335, Springer Verlag, 1997.

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

[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/Wolf99] M. Müller-Olm and A. Wolf. On Excusable and Inexcusable Failures: Towards an Adequate Notion of Translation Correctness. In J. M. Wing, J. Woodcock, and J.Davies, editors, *Proceedings of Formal Methods FM'99*, volume 1709 of *LNCS*, pages 1107—1127, Toulouse, France, 1999. Springer Verlag.

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

[Pnueli+98] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *Proc. 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lisbon, Portugal, March 1998.

[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.

Date received 20.03.03

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

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

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