INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Will Informatics be able to Justify the Construction of Large Computer Based Systems?

Wolfgang Goerigk, Hans Langmaack

Bericht Nr. 2015
Mai 2001

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstraße 40, D-24098 Kiel
Germany

Will Informatics be able to Justify the Construction of Large Computer Based Systems?

Wolfgang Goerigk, Hans Langmaack

wg@informatik.uni-kiel.de
hl@informatik.uni-kiel.de

Bericht Nr. 2015
Mai 2001

Dieser Bericht ist als persönliche Mitteilung zu verstehen.

Abstract

The present article addresses 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.

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-selfevident 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. We will reflect realistic requirements and introduce the notion of relative correctness and its preservation, in view of errors which can be accepted or have to be avoided.

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).

Keywords

implementation verification and correctness, compiler verification, preservation of correctness, a-posteriori-result checking, code inspection, computer based systems, safety, security, Trojan horses, trusted program execution

Zusammenfassung

In der vorliegenden Arbeit behandeln wir die korrekte Konstruktion und Wirkungsweise großer und sicherheitsrelevanter Computer-basierter Systeme. Wir stellen die Frage, ob Informatik, ob Informatiker gerechterweise für Inkorrektheiten zur Verantwortung gezogen werden können, ob sie das richtige Konstruieren und Funktionieren werden rechtfertigen können. Dabei verstehen wir das Wort Rechtfertigung in diesem Sinn: Der Entwurf eines Systems, das zugrundeliegende mathematisch-logische Modell der Informationsflüsse und die konstruierte steuernde Software müssen dergestalt sein, daß gewünschte Wirkungen, interne Fehlerfreiheit und Robustheit gegen Fehlbedienungen und heimtückische Attacken von außen als logische Konsequenzen einzig des mathematischen Modells vorhersagbar sind.

Hardware-Technologen sind aus gutem Grund sehr sensibel für Hardwarefehler, deren Auftreten einer Sensation gleichkommt. Systemausfälle gehen deshalb meist und mit wachsendem Anteil auf gedanklich-logische Fehler in der Software zurück. Folgen wir dieser Beobachtung und akzeptieren wir die klare Trennung der Verantwortungsbereiche zwischen Hardware- und Systemsoftwareingenieur, dann wird klar: Softwareengineering darf sich auf korrektes Funktionieren der Hardware verlassen und diese auch ausnutzen, um mit ähnlich hoher Sensibilität für Fehler rigoros den Nachweis der Korrektheit ablaufender Maschinenimplementierungen zu führen. Wir werden dies speziell für Übersetzer zeigen, und zwar so, dass Korrektheit nur abhängig ist von korrektem Funktionieren der Hardware und der Richtigkeit gedanklich-logischer Schlüsse. Nach oben hin ist maschinenunabhängige Programmiersprachsemantik Gegenstand des Kontrakts zwischen Softwareimplementierer und Anwendungsprogrammierer, der aber auch Klarheit hinsichtlich der Art gewünschter Implementierungskorrektheit beinhalten muss. Sie muss Sinn machen und realistisch garantiert werden können. Relative Programmkorrektheit und ihre Erhaltung, in dieser Arbeit eingeführt, fasst realistische Anforderungen auch hinsichtlich zu akzeptierender und zu vermeidender Fehler.

Seit über 40 Jahren haben theoretische Informatik, Softwareengineering und Übersetzerkonstruktion wichtige Beiträge zu korrekter Übersetzungsspezifikation und korrekter höhersprachlicher Übersetzerimplementierung geleistet. Für den genauso wichtigen dritten Schritt, die Übersetzung (das Bootstrapping) von Compilerprogrammen in den Maschinencode von Wirtmaschinen, gibt es bislang keine realistischen Methoden oder Rezepte. Und das, obwohl lange bekannt ist – spätestens seit Ken Thompsons Turing Award Lecture im Jahre 1984 – dass ausführbarer Code durch versteckte Trojanische Pferde in Übersetzerimplementierungen gefährlich manipuliert werden kann, selbst wenn die Übersetzer Quellcodeinspektionen, Validierung und härteste Tests bestehen. Im vorliegenden Aufsatz zeigen wir auf, wie diese gefährliche Lücke geschlossen werden kann. Die Forschergruppe *Verifix* der Universitäten Karlsruhe, Kiel und Ulm, gefördert von der Deutschen Forschungsgemeinschaft (DFG), hat dazu die Methode der rigorosen syntaktischen a-posteriori-Code-Inspektion entwickelt.

Schlüsselworte

Implementierungskorrektheit und -verifikation, Übersetzerverifikation, Erhaltung von Programmkorrektheit, a-posteriori-Resultatprüfung, Codeinspektion, Computer-basierte Systeme, Sicherheit, Trojanische Pferde, zuverlässige Programmausführung

Contents

1 Introduction

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

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

1.1 Motivation and Outline

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

Since hardware turns out to be quite reliable, the question comes down to software, *i.e.*, to abstract and mathematically treatable components of systems. The rigid nature of matter educates hardware technologists to be extremely sensitive towards hardware failures; they are felt as sensations. So system faults are mostly and increasingly caused by software. This observation is crucial and matches the clear delimitation of responsibilities between hardware and system software engineering. Software engineers are permitted to assume hardware to work correctly, and to exploit this assumption for equally sensitive, rigorous low level implementation verification of software. Once software engineers and application programmers can count on (trust in) the correctness of their low level machine implementations and integrated systems, an equal status of sensitivity also for software faults becomes justifiable.

That is to say, at the low end, system software engineering meets hardware engineering, and at the upper end, compiler constructors and system software engineers meet application programmers and software engineers. They want to express their software in problem-oriented languages and rely on machine independent semantics. Compiler constructors cannot be made responsible for application program faults. Actually, a compiler has to be constructed without any knowledge about the intended meaning of application programs. The contract has to be with respect to program semantics. As a return, the application programmer expects correctly implemented machine executa-

bles. But both, application and system programmers have to be aware of what kinds of implementation correctnesses make sense and can realistically be guaranteed. We will reflect realistic requirements and introduce the notion of relative correctness and its preservation and variants in view of data and program representation and of errors which can be accepted or others which have to be avoided.

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

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

There is a current trend for system software to be required *open source*, enabling source code scrutiny for operating system components, networking software and also for compilers and other tools and utilities. This will definitely unveal a lot of bugs and even malicious code like Trojan horses or so-called *easter eggs*. However, we want to stress, that the open source idea crucially depends on trusted compilation. Source level scrutiny does not sufficiently guarantee trustworthiness of executable software [Tho84, Goe00a, Goe00c, Goe00b]. There are sophisticated and intelligent techniques to completely hide Trojan horses in binary compiler executables, which are not part of the alleged source code, but might cause unexpected, arbitrary even catastrophic effects if target programs are eventually executed. No source code scrutiny, no source level verification, no compiler validation, virtually no test, not even the strong compiler bootstrap test does help. Note that in this situation it is also very unlikely that any of the known security techniques will help, because not even the application programmer can give a guarantee that her/his delivered application has not been compromised by auxiliary software utilities or compilers used during software construction.

We do not want to blame or condemn anybody personally. The problem is enormous, very awkward, and in certain central areas it seems nearly unreasonable to ask for problem solutions in depth. However, because of potential disastrous consequences, in-

formaticians must attack the problem and seek for solutions to give rigorous guarantees. And we shall see that in the crucial area of correct compiler construction, informatics science can achieve quite a lot. There is a remarkable interplay of informatics as a foundational structure science and as an engineering science.

Industrial software production for realistic safety and security critical applications enforces immense checking efforts. The amount of work is apparently unmanageable, and consequently, it is left undone. That is to say: On the one hand, we observe dangerous omissions in industrial practice. On the other hand, we realize how enormous the problems are. Thus, informatics science, in particular viewed at as an engineering science, has to attack them as problems of basic research – supported by research funding institutions like Deutsche Forschungsgemeinschaft (DFG) or German Federal Board of Safety and Security in Information Technology (Bundesamt für Sicherheit in der Informationstechnik, BSI), institutions which we find in all states with high science and technology standards. We are not allowed to leave industry people alone with their responsibility for necessary efforts which are seemingly unsurmountable at present. It is necessary to clearly identify the problems and to work towards methods for their rigorous solution which work out in practice. Mathematics as the classical structure science helps. Again and again, mathematicians invent ways of gaining and communicating insights, which are rigorous and convincing even though or maybe even because they are not completely formal. We shall see that in the central area of correct compiler construction and implementation our techniques of insight can cope with realistic software tasks, and of course then can serve as a model outside the area of compilers. Actually, it is the too often neglected low level implementation verification of so-called *initial* compilers (see section 10) which involves a certain amount of manual checking and proving. For principle reasons we cannot leave all checking work to programmed computers if we do not want to run into circuli vitiosi. Trust in any executed program would further on dangerously and hopelessly depend on auxiliary software of uncertain pedigree [JBFB01].

1.2 Notions Mentioned in the Title

Let us briefly explain the notions which we mentioned in the title of this essay: *Informatics* (*Computer Science* in the Anglo-American literature) is the scientific discipline of design, construction and networking and of application and programming of computers (often called processors). Although not the most modern one, this definition is well stressing the two important areas of work: hardware and software. *Computer based systems* (CBS) are engineering systems with embedded computers, programs, sensors, actuators, clocks and connecting channels in a physical environment [SV96]. Realistic computer based systems use to be large, complex, and safety and/or security critical [Lap94]. The latter means that systems may cause heavy damages to health and economy by unintended (internal) failures and by intended (external) attacks.

As a matter of fact, the bare existence of large computer based systems is justified. We need them. But what about correct construction and functioning of such systems? Can informaticians be made responsible, be accounted for, will they be able to justify systems to work correctly as intended? We use the term *justification* in this sense, *i.e.*, for the design of computer based systems, the formulation of mathematical models of information flows in systems and the construction of software to be such that

the expected system effects and the absence of failures and violations are foreseeable as logical consequences of the models alone. Not every system has such substantial delineable area which can be justified in this rigorous sense. In our opinion, however, if such a rigorous treatment is possible, it should be required for high safety and security standards. If we can logically foresee (infer) every desired property, especially safety and security properties, then we say that the computer based system has been (mathematically) *proved correct*, has been *verified*. We use the word *verification* in this sense, whereas *validation* means finding by experiments that a system fulfills our intents. Validation is not our main topic in this essay.

1.3 Consistent Checkability of Software Production Processes

Large computer based systems are essentially controlled by embedded processors and their software. General experience shows that the hardware processors actually work by far more reliably than software does and hence, the weaknesses of a computer based systems are in most cases due to software problems.

Every software production (implementation) process today still has two major gaps in trustworthiness of consistent checkability, namely the transitions from

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

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

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

1.4 DFG-Research Project Verifix - Verified Compilers

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

- for realistic, industrially applicable imperative and object-oriented source programming languages and

- real world target and host processors
- which, by help of mechanical proof support (*e.g.*, by PVS [ORS92] or ACL2 [KM94]) and by exploiting the idea of a-posteriori result checking are rigorously verified as executable host machine programs and
- which nevertheless generate efficient code that compares to unverified compilers.
- *Verifix* uses practically and industrially approved compiler construction methods
- and the proof methodology supplements compiler construction, not vice versa.

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

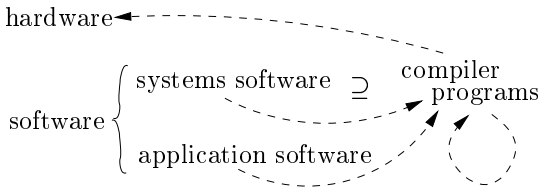


Figure 1. The central role of correct compilers

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

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

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

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

engineering. Correct realistic compilation establishes a trustworthy execution basis for further software development.

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

1.5 Levels of Trustworthiness and Quality

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

However, an "officially admitted" tool (like a compiler or theorem prover) is not necessarily fully verified. So for instance a compiler just needs to pass an official validation test suite. It is well-known, that such tests do not suffice nor replace correctness proofs [Dij76]. Official IT-certification prescriptions like those published by BSI in 1989/90/94 [ZSI89, ZSI90, BSI94] require:

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

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

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

For this reason certification authorities still do not trust in any existing realistic compiler. Instead, they often perform parallel semantical inspections of both high level code and low level machine code [Pof95], *i.e.*, they check a-posteriori that the target code will work as expected from the source code.

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

$$(\pi_{\mathsf{SL}}, \pi_{\mathsf{TL}}) \in \mathcal{C}^{\mathsf{SL}}_{\mathsf{TL}}$$

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

Inspection resp. a-posteriori-result checking is an old idea [BLR89]. And we know the method from school mathematics: since for instance integer division is felt more error prone than multiplication, we *double-check the results* of division by multiplication. We also use to double-check the results of linear equations solutions by simpler matrix-vector-multiplication. By *code inspection* with respect to compilers we mean the a-posteriori-result checking of compiler generated code. Result-checking is often much easier that (a-posteriori) verification. Moreover, it is an interesting observation, that industrial software engineers and certification boards trust the technique of a-posteriori-result checking. Although there is a well-established and reasonably developed program verification theory, it is often not well-applicable to large systems or even large amounts of low level code. Therefore it is so interesting to observe, both from a theoretical and from a practical point of view, that realistic scientifically founded fully verified compiler construction has to reach back also to a-posteriori-checking to a small, but decisive extent.

We will see this later while proving low level compiler implementation correctness (section 10) and hence full compiler correctness. Consequently, since this is possible and feasible, already in 1989/90 one of the authors proposed to introduce an even higher IT-safety and security level of quality Q8 (Production, proof and checking tools are fully verified, not only at high level, but also down to implementations in executable binary machine code). Otherwise, the low level gap of realistic compilation will remain open forever.

2 Code Inspection in Compiler Construction Processes

The *Verifix*-thesis is that after 40 years compiler construction and more than 30 years software engineering it should no longer be necessary to inspect low level generated code, not even for safety and security critical software. The new higher quality level Q8 should be introduced which is reaching beyond Q7. It should be desirable and required for industrial software construction, in particular for realistic compilers. We cannot entirely avoid manual low level code inspection[Fag86], however, it is only necessary for initial compiler executables which cannot be implemented on behalf of a verified bootstrapping compiler executable. The requirement for low level code inspection only make sense in this respect. In principle, it should suffice to perform this work exactly once. However, realistic industrial correct compiler engineering additionally needs repeatable methods for constructing correct initial compiler implementations from the scratch whenever necessary.

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

- the correctness of source level application programs with respect to their specifications,

- hardware, *i.e.*, target and host processors to work correctly as described in their instruction manuals,
- correct rigorous (mathematical, logical) reasoning

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

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

$$\mathsf{SL} \;=\; \mathsf{IL}_0, \mathsf{IL}_1, \ldots, \mathsf{IL}_m \;=\; \mathsf{TL}, \quad m \geq 1\,.$$

by compiler passes

$$\tau^1_{\mathsf{HL}}\,;\, \tau^2_{\mathsf{HL}}\,;\, \ldots\,;\, \tau^m_{\mathsf{HL}} \;=\; \tau_{\mathsf{HL}}$$

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

$$\pi_{\mathsf{SL}} \;=\; \pi_{\mathsf{IL}_0}\,,\, \pi_{\mathsf{IL}_1}\,,\, \ldots\,,\, \pi_{\mathsf{IL}_m} \;=\; \pi_{\mathsf{TL}}\,.$$

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

$$\tau^1_{\mathsf{HL}}\,;\, \gamma^1_{\mathsf{HL}}\,;\, \tau^2_{\mathsf{HL}}\,;\, \gamma^2_{\mathsf{HL}}\,;\, \ldots\,;\, \tau^m_{\mathsf{HL}}\,;\, \gamma^m_{\mathsf{HL}} \;=\; \overline{\tau}_{\mathsf{HL}}\,.$$

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

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

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

Hardware engineering has, for good money reasons, developed an error handling culture by far better than software engineering [Goe96b]. Hardware errors are sensations, whereas software errors are commonplace. Since hardware errors cannot easily be repaired, they bear the risk of high economic damage. Thus, hardware engineers seek even for small gaps where faults could creep in. Seemingly, software errors can

easily be repaired. But often bug fixes make things even worse, and also software errors bear a high risk.

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

3 Towards Trusted Realistic Compilation

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

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

Consequently, we will study correct realistic compilation for sequential imperative programming languages, and in particular we restrict ourselves to *transformational* programs (cf. section 4), because we are mainly interested in the input/output relations defined by program semantics. We need full compiler correctness proofs only

- for one initial compiler per processor family with identified target and host machine language $\mathsf{TML} = \mathsf{HML}$
- and for one sufficiently high-level source language SL which allows for formulating compilers and system programs.

All further compilers, *e.g.*, optimizing compilers or those for more expressive system programming languages $\overline{\mathsf{SL}}$, compiler generators, any further systems software like provers, proof assistants and proof checkers need no further correct implementation steps below SL or $\overline{\mathsf{SL}}$. In particular, no further machine code inspection is necessary, and executable binary versions of these programs can be generated purely mechanically following N. Wirth [Wir77]. The resulting code is proved to be correct due to a bootstrapping theorem (cf. section 8, also found in [Lan97b, Goe99, GL01a, GL01b]). As a conclusion we want to stress, that the second software production process gap (cf. section 1.3) can be closed if the *Verifix*-recipes of correct initial compilers and a-posteriori-program-checking (cf. section 10 and [Lan97c, GH98b, Hof98, Lan97b, Lan97a]) are obeyed.

In the context of the *Verifix*-project we will furthermore demonstrate how to develop correct compiler generating tools and how to incorporate even unverified existing tools in a fully trusted and rigorously proved correct manner [HGG+99, GZG99, GGH+97]. Moreover, the specification and verification system PVS [ORS92] is used for mechanical

proof support, in particular for the formalization and verification of the compiling specifications, and for providing support for high level compiler implementation verification using a transformational approach.

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

3.1 Three Steps towards Fully Correct Compilers

In his work on Piton's verified translation [Moo88, Moo96] J S. Moore recommends three steps in order to present a convincing correctness proof of a compiler written (or implemented) in executable binary host machine code HML:

1. Specification of a compiling relation $\mathcal{C}$ between abstract source and target languages SL and TL, and compiling (specification) verification w.r.t. an appropriate semantics relation $\sqsubseteq$ between language semantics $[\![\cdot]\!]_{\mathsf{SL}}$, $[\![\cdot]\!]_{\mathsf{TL}}$.
2. Implementation of a corresponding compiler program τ_{HL} in a high level host language HL close to the specification language, and high level compiler implementation verification (w.r.t. $\mathcal{C}$ and to program representations $\varphi^{\mathsf{SL}}_{\mathsf{SL}'}$ and $\varphi^{\mathsf{TL}}_{\mathsf{TL}'}$).
3. Implementation of a corresponding compiler executable τ_{HML} written in binary host machine language HML, and low level compiler implementation verification (with respect to $[\![\tau_{\mathsf{HL}}]\!]_{\mathsf{HL}}$ and program representations $\varphi^{\mathsf{SL}'}_{\mathsf{SL}''}$ and $\varphi^{\mathsf{TL}'}_{\mathsf{TL}''}$).

If we work through every step, then τ_{HL} resp. τ_{HML} is a correct or verified compiler (implementation). If we want to stress that a correctness proof has been achieved even for a compiler executable like τ_{HML} on a real processor, then we sometimes call it *fully*

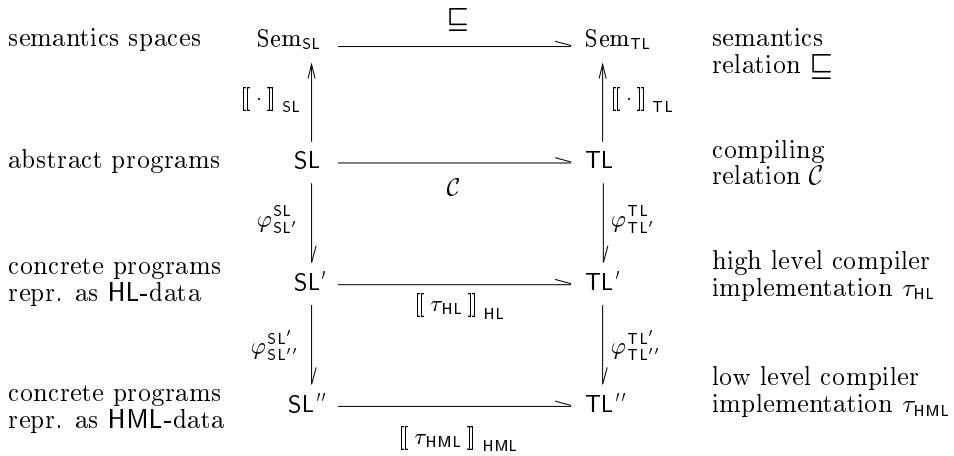


Figure 2. Three steps for correct compiler implementation τ_{HML}

correct (verified) as informally explained before. Figure 2 informally outlines the three essential steps that we have to work through for the construction and verification of fully correct compilers, and in particular of fully correct initial compiler executables. We will make everything shown in this diagram precise in the following sections 4 and 5, and in particular we refer to section 5.3.1, which discusses Figure 2 precisely.

4 Transformational Programs

We model the semantics of transformational programs by partial relations (or multi-valued partial functions) f between input domains D_i and (not necessarily different) output domains D_o. Thus, a program semantics is a subset

$$f \subseteq D_i \times D_o$$

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

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

$$D^\Omega =_{\text{def}} D \,\dot\cup\, \Omega$$

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

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

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

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

$$\Omega_i = A_i \,\dot\cup\, U_i \quad \text{and} \quad \Omega_o = A_o \,\dot\cup\, U_o$$

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

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

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

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

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

4.1 Correct Implementation

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

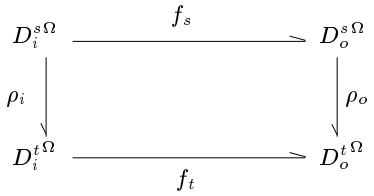


Figure 3. Source and target program semantics f_s, f_t and data representations ρ_i, ρ_o

Definition 4.1 (Correct implementation or refinement). *f_t is said to be a* correct implementation *or* refinement *of f_s relative ρ_i and ρ_o and associated error sets iff*

$$(\,\rho_i\,;\,f_t\,)\,(d) \;\subseteq\; (\,f_s\,;\,\rho_o\,)\,(d) \,\cup\, A_o^t \tag{1}$$

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

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

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

If f_t correctly implements f_s relative ρ_i and ρ_o, we will write

$$\rho_i\,;\,f_t \quad \sqsupseteq \quad f_s\,;\,\rho_o \tag{2}$$

or even shorter just $f_t \sqsupseteq f_s$ (with a boldface $\sqsupseteq$). We indicate this diagram commutativity by the diagram in Figure 4 and we will later see that we can compose correct

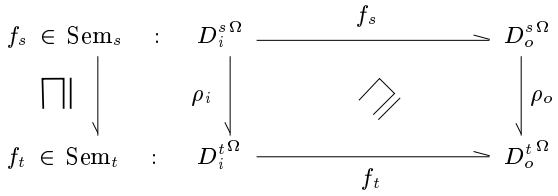


Figure 4. Commutative diagram expressing correct implementation

implementation diagrams both vertically and horizontally, which is a very important fact for practical software engineering and compiler generation.

There are some variations of *correct (relative) implementation* as of Definition 4.1, which we would like to discuss: We speak of *correct acceptable implementation* resp. of a *correct regular implementation* or *refinement* iff

$$\begin{array}{lllll} \emptyset & \neq & (\rho_i; f_t)\,(d) & \subseteq & (f_s;\rho_o)(d) \cup A_o^t \quad (\textit{acceptable}) \\ \emptyset & \neq & (\rho_i; f_t)\,(d) & \subseteq & (f_s;\rho_o)(d) \qquad\quad (\textit{regular}) \end{array}$$

holds for all $d \in D_i^s$ where $\emptyset \neq f_s(d) \subseteq D_o^s \cup A_o^s$ resp. $\emptyset \neq f_s(d) \subseteq D_o^s$. But note that – in contrast to the different program correctness notions (cf. section 4.2) – all three implementation correctness notions are independent; neither one implies the other. It is remarkable, however, that concrete correctness proofs turn out to be of less complexity if $\perp$ is supposed to be unacceptable, *i.e.*, if we prove variants of correct regular implementation [MOW00, Wol01]. If $\perp$ is supposed acceptable, then our experience shows, that we have to characterize greatest fixed points exactly and to additionally use computational or fixed point induction in order to prove variants of correct acceptable implementation, *e.g.*, preservation of partial correctness [Goe00c, Goe00b].

4.2 Preservation of Relative Correctness

It is an important observation, that we can exactly characterize correct implementation by *preservation of relative correctness* [Wol01], which generalizes Floyd's and Hoare's notions of partial respectively total program correctness. Let $f \in {D_i}^{\Omega} \rightharpoonup {D_o}^{\Omega}$ be a program semantics and let $\Phi \subseteq D_i$ and $\Psi \subseteq D_o$ be predicates, *i.e.*, subsets of regular data.

Definition 4.2 (Relative program correctness). *f is called* (relatively) correct with respect to *(pre- and post-conditions) Φ and Ψ ($\langle\, \Phi \,\rangle\, f \,\langle\, \Psi \,\rangle$ for short), iff*

$$f(\Phi) \;\subseteq\; \Psi \cup A_o \tag{3}$$

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

Theorem 4.1 (Preservation of relative correctness). *f_t correctly implements f_s ($\rho_i; f_t \sqsupseteq f_s; \rho_o$) if and only if*

$$\langle\, \Phi \,\rangle\, f_s \,\langle\, \Psi \,\rangle \implies \langle\, \rho_i(\Phi) \,\rangle\, f_t \,\langle\, \rho_o(\Psi) \,\rangle$$

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

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

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

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

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

□

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

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

$$\begin{array}{lll} \langle\, \Phi \,\rangle\, f\, \langle\, \Psi \,\rangle_{acc} & \text{iff} & \emptyset \neq f\,(d) \subseteq \Psi \cup A_o \quad (\mathit{acceptable}) \\ \langle\, \Phi \,\rangle\, f\, \langle\, \Psi \,\rangle_{reg} & \text{iff} & \emptyset \neq f\,(d) \subseteq \Psi \quad (\mathit{regular}) \end{array}$$

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

$$\begin{array}{lll} \langle\, \Phi \,\rangle\, f_s\, \langle\, \Psi \,\rangle_{acc} & \Longrightarrow & \langle\, \rho_i(\Phi) \,\rangle\, f_t\, \langle\, \rho_o(\Psi) \,\rangle_{acc} \quad (\mathit{acceptable}) \\ \langle\, \Phi \,\rangle\, f\, \langle\, \Psi \,\rangle_{reg} & \Longrightarrow & \langle\, \rho_i(\Phi) \,\rangle\, f_t\, \langle\, \rho_o(\Psi) \,\rangle_{reg} \quad (\mathit{regular}) \end{array}$$

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

4.3 Classical Notions of Correct Implementation

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

$$f \in D_i \rightharpoonup D_o$$

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

Let us now choose the same particular error sets $A = A_i = A_o =_{\mathrm{def}} \{a\}$ and $U = U_i = U_o =_{\mathrm{def}} \{u\}$ for both domains D_i and D_o with $\bot \in \{a, u\}$ and $\Omega = \Omega_i = \Omega_o =_{\mathrm{def}} A \dot\cup U$, and extend f to

$$f^{ext} \in {D_i}^{\Omega} \rightharpoonup {D_o}^{\Omega} \text{ by } f^{ext} =_{\mathrm{def}} f \cup ((D_i \setminus \mathrm{dom}_f) \times \{\bot\}) \cup id_\Omega\,.$$

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

Partial and Total Program Correctness

Relative and acceptable correctness are equivalent to partial correctness

$$\langle\, \Phi \,\rangle\, f^{ext}\, \langle\, \Psi \,\rangle \iff \langle\, \Phi \,\rangle\, f^{ext}\, \langle\, \Psi \,\rangle_{acc} \iff \{\Phi\}\, f\, \{\Psi\}$$

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

$$\langle\, \Phi \,\rangle\, f^{ext}\, \langle\, \Psi \,\rangle \iff \langle\, \Phi \,\rangle\, f^{ext}\, \langle\, \Psi \,\rangle_{acc} \iff \langle\, \Phi \,\rangle\, f^{ext}\, \langle\, \Psi \,\rangle_{reg} \iff [\, \Phi \,]\, f\, [\, \Psi \,]\,.$$

Preservation of Partial and Total Program Correctness

It is not only that relative program correctness generalizes classical partial and total program correctness. Our notion of correct (relative) implementation also generalizes well-known implementation correctness notions. If we consider $\perp$ acceptable, then

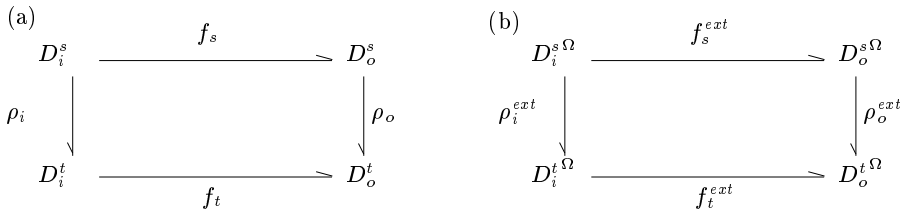


Figure 5. Classical correct implementation versus preservation of relative correctness. Consider a classical unextended implementation diagram (a). If we extend f_s and f_t as explained before, and if we extend the data representation relations ρ_i and ρ_o by $\rho_i^{ext} =_{\text{def}} \rho_i \cup id_\Omega$ respectively $\rho_o^{ext} =_{\text{def}} \rho_o \cup id_\Omega$ as well, then we get the extended diagram (b).

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

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

$$\rho_i^{ext}; f_t^{ext} \sqsupseteq f_s^{ext}; \rho_o^{ext} \quad \Longleftrightarrow \quad (\rho_i\,;\, f_t)\,|\,_{\text{dom}_{f_s\,;\rho_o}} \subseteq f_s\,;\,\rho_o \quad \text{and} \quad \text{dom}_{\rho_i\,;\,f_t} \supseteq \text{dom}_{f_s\,;\,\rho_o}$$

expresses exactly the classical preservation of total correctness.

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

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

4.4 Composability

We mentioned that composability (transitivity) of correct implementation is crucial for modular software construction and verification, and in particular for stepwise compilation and compiler construction and implementation. In the following we prove vertical and horizontal transitivity, *i.e.*, that we may (vertically) decompose correct

implementations into steps (or phases), and that correct implementation distributes (horizontally) over sequential (relational) composition. In both cases we will prove a weaker, more generally applicable transitivity result as well.

Theorem 4.2 (Vertical transitivity of correct implementation). *If f_t correctly implements f_s, and if $f_{t'}$ does so for f_t, then $f_{t'}$ correctly implements f_s.*

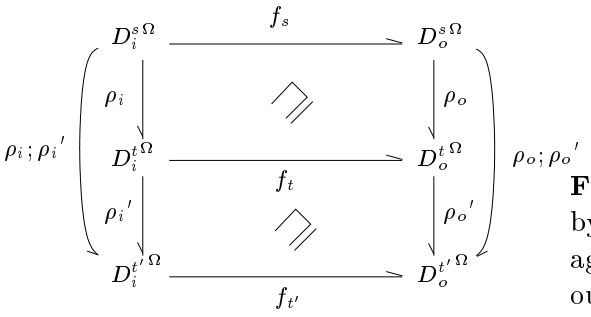


Figure 6. Vertical composition expressed by commutative diagrams: If the inner diagrams are commutative, then so is the outer one.

Proof. Let $\Phi \subseteq D_i^s$, $\Psi \subseteq D_o^s$ and let $\langle\, \Phi \,\rangle\, f_s \,\langle\, \Psi \,\rangle$.

Then, due to Theorem 4.1, we have $\langle\, \rho_i(\Phi) \,\rangle\, f_t \,\langle\, \rho_o(\Psi) \,\rangle$ (commutativity of the upper diagram) and $\langle\, \rho_i{}'(\rho_i(\Phi)) \,\rangle\, f_{t'} \,\langle\, \rho_o{}'(\rho_o(\Psi)) \,\rangle$ (commutativity of the lower diagram for pre-condition $\rho_i(\Phi)$ and post-condition $\rho_o(\Psi)$). But the latter just means

$$\langle\, (\rho_i;\rho_i{}')\,(\Phi) \,\rangle\, f_{t'} \,\langle\, (\rho_o;\rho_o{}')\,(\Psi) \,\rangle$$

which completes the proof due to equivalence of correct implementation and preservation of (relative) correctness (Theorem 4.1). □

Theorem 4.3 (Horizontal transitivity of correct implementation). *If f_t correctly implements f_s, and if $f_{t'}$ does so for $f_{s'}$, then $f_t; f_{t'}$ correctly implements $f_s; f_{s'}$.*

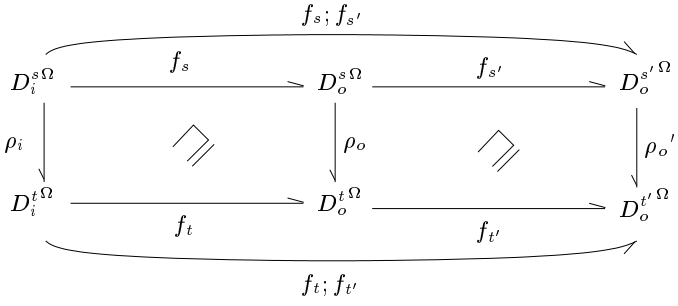


Figure 7. Horizontal composition expressed by commutative diagrams: If the inner diagrams are commutative, then so is the outer one.

Proof. Let

$$\rho_i; f_t \;\sqsupseteq\; f_s; \rho_o \quad \text{and} \tag{4}$$
$$\rho_o; f_{t'} \;\sqsupseteq\; f_{s'}; \rho_o{}' \quad \text{and} \tag{5}$$
$$(f_s; f_{s'})\,(d) \;\subseteq\; D_o^{s'} \cup A_o^{s'} \tag{6}$$

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

Note that $(f;g)(d) = g(f(d))$ and that ";" is associative and monotonic in both arguments. By (6) we have $f_{s'}(f_s(d)) \subseteq D_o^{s'} \cup A_o^{s'}$ and by (5, commutativity of the right diagram) and associativity of ";" we get $(\rho_o; f_{t'})(f_s(d)) \subseteq (f_{s'}; {\rho_o}')(f_s(d)) \cup A_o^{t'}$ and hence

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

$$\text{and also} \quad f_{t'}((f_s;\rho_o)(d) \cup A_o^t) \subseteq (f_s; f_{s'}; {\rho_o}')(d) \cup A_o^{t'} \quad (8)$$

The latter holds by strong error strictness of $f_{t'}$, *i.e.*, $f_{t'}(A_o^t) \subseteq A_o^{t'}$, from (7). By strong error strictness of $f_{s'}$ and (6) we have $f_s(d) \subseteq D_o^s \cup A_o^s$. Thus, by (4, commutativity of the left diagram) we get

$$(\rho_i; f_t)(d) \subseteq (f_s;\rho_o)(d) \cup A_o^t \quad (9)$$

and finally, by monotonicity of relation composition, (9) and (8) we get

$$f_{t'}((\rho_i; f_t)(d)) \subseteq (f_s; f_{s'}; {\rho_o}')(d) \cup A_o^{t'} \quad (10)$$

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

$$\begin{array}{lllll} (\rho_i; f_t; f_{t'})(d) & = & f_{t'}((\rho_i; f_t)(d)) & \subseteq & f_{t'}((f_s;\rho_o)(d) \cup A_o^t) \\ & \subseteq & f_{t'}((f_s;\rho_o)(d)) \cup A_o^{t'} & = & (\rho_o; f_{t'})(f_s(d)) \cup A_o^{t'} \\ & \subseteq & ((f_{s'}; {\rho_o}')(f_s(d)) \cup A_o^{t'}) \cup A_o^{t'} & = & (f_s; f_{s'}; {\rho_o}')(d) \cup A_o^{t'} \end{array} \quad \square$$

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

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

Let, for vertical composition, commutative inner diagrams and the outer diagram with its data representations $\rho_i; {\rho_i}'$ and $\rho_o; {\rho_o}'$ be as in Figure 8.

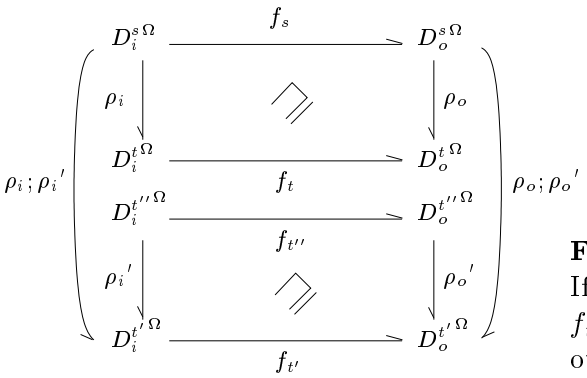


Figure 8. Weak vertical composability: If the inner diagrams are commutative and f_t and $f_{t''}$ appropriately coincide, then the outer diagram is commutative as well.

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

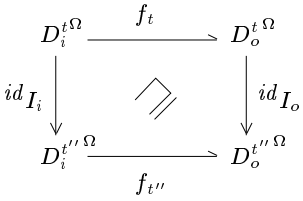
Furthermore, let

$$\begin{array}{lll} I_i & =_{\text{def}} & \rho_i(D_i^{s\,\Omega}) \cap {\rho_i}'^{-1}(f_{t'}^{-1}(D_o^{t'\,\Omega})) \quad \subseteq \quad D_i^{t\,\Omega} \cap D_i^{t''\,\Omega} \\ I_o & =_{\text{def}} & D_o^{t\,\Omega} \cap {\rho_o}'^{-1}(D_o^{t'\,\Omega}) \qquad\qquad\qquad\quad \subseteq \quad D_o^{t\,\Omega} \cap D_o^{t''\,\Omega} \end{array}$$

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

Corollary 4.1. *If the two inner diagrams of Figure 8 are commutative, if the intermediate error sets agree, and if* $f_t\,|_{I_i \times I_o} \supseteq f_{t''}\,|_{I_i \times I_o}$*, then the outer diagram is commutative as well.*

Proof. Consider the following (coupling) diagram:



Since $f_t\,|_{I_i \times I_o} \supseteq f_{t''}\,|_{I_i \times I_o}$, it is immediately clear that this diagram is commutative. Thus, the outer diagram is commutative due to vertical composition Theorem 4.2. □

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

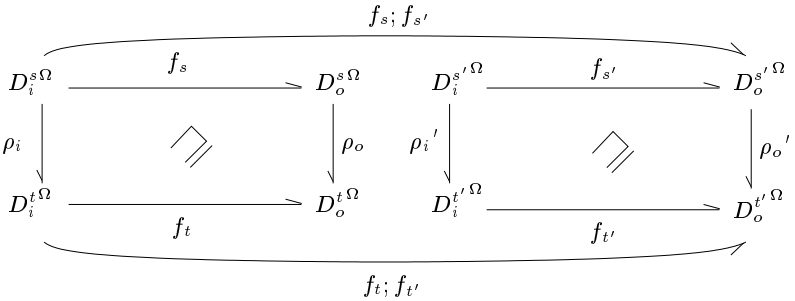


Figure 9. Weak horizontal composability: If the inner diagrams are commutative and the data representations ρ_o and ${\rho_i}'$ appropriately coincide, then the outer diagram is commutative as well.

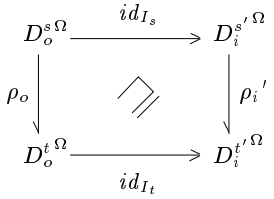
Let, again,

$$\begin{array}{lll} I_s & =_{\text{def}} & f_s(D_i^{s\,\Omega}) \cap D_i^{s'\,\Omega} \qquad\qquad\quad \subseteq \quad D_o^{s\,\Omega} \cap D_i^{s'\,\Omega} \\ I_t & =_{\text{def}} & f_t(\rho_i(D_i^{s\,\Omega})) \cap f_{t'}^{-1}(D_o^{t'\,\Omega}) \quad \subseteq \quad D_o^{t\,\Omega} \cap D_i^{t'\,\Omega}. \end{array}$$

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

Corollary 4.2. *If the two inner diagrams of Figure 9 are commutative, if the intermediate error sets agree, and if* $\rho_o\,|_{I_s \times I_t} \subseteq {\rho_i}'\,|_{I_s \times I_t}$*, then the outer diagram is commutative as well.*

Proof. Consider the following (coupling) diagram:



Since $\rho_o\,|_{I_s \times I_t} \subseteq \rho_i{}'\,|_{I_s \times I_t}$, it is immediately clear that this diagram is commutative. Thus, the outer diagram is commutative due to horizontal composition Theorem 4.3. □

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

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

5 Correct Compiler Programs

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

$$[\![\pi]\!]_{\mathsf{L}} \in \mathrm{Sem}_{\mathsf{L}} =_{\mathrm{def}} (D_i^{\mathsf{L}\Omega} \rightharpoonup D_o^{\mathsf{L}\Omega}) .$$

For a source language SL, a target language TL and proper source programs $\pi_s \in \mathsf{SL}$ and $\pi_t \in \mathsf{TL}$ with semantics $f_s = [\![\pi_s]\!]_{\mathsf{SL}}$ and $f_t = [\![\pi_t]\!]_{\mathsf{TL}}$, section 4 defines the semantical relation $f_t \sqsupseteq f_s$ of correct implementation:

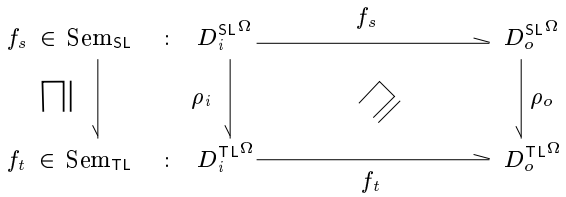


Figure 10. Correct implementation for sequential imperative programs

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

5.1 Compiling Specifications

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

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

a mathematical relation between abstract source and target programs. $\mathcal{C}$ might be given by a closed inductive definition, more or less constructive, or by a set of bottom-up rewrite rules applied by a term or graph rewrite system (*e.g.*, bottom up rewrite systems, BURS [PLG88]) as for instance used in rule-based code generators.

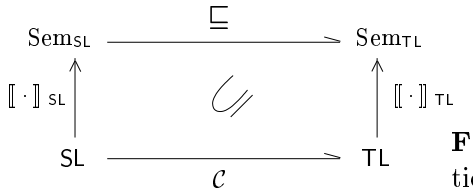


Figure 11. Correctness of the compiling specification $\mathcal{C}$

Definition 5.1. *We call* $\mathcal{C}$ correct, *if and only if for any well-formed source program* $\pi_s \in \mathsf{SL}$, *every associated target program* $\pi_t \in \mathcal{C}(\pi_s)$ *is a correct implementation of* π_s, *i.e., if and only if the diagram in Figure 11 is commutative in the sense*

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

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

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

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

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

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

But how is a correct $\mathcal{C}$ related to $\mathscr{C}$ in general? Of course, $\mathscr{C} \not\subseteq \mathcal{C}$ ($\mathcal{C}$ might even be empty, *e.g.*, for the pathological compiler which fails on every source program). Restricted to well-formed source programs, $\mathcal{C}$ is a subset of $\mathscr{C}$. However, in general $\mathcal{C}$ might relate non well-formed SL-programs (which have no semantics) to TL-programs. Perhaps well-formedness is hard to decide or even undecidable. So compilers sometimes generate or have to generate target code also for improper source programs without

explicitly signaling an error. The user should be careful, keep this in mind and avoid non-well-formed compiler inputs. In any case, in general $\mathcal{C} \not\subseteq \mathscr{C}$ as well.

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

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

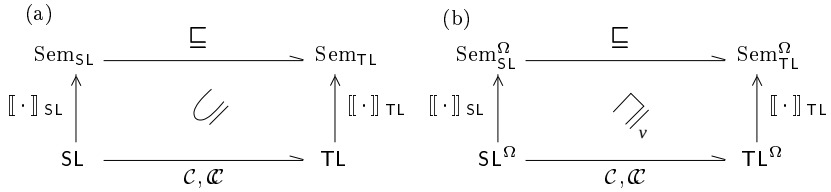


Figure 12. Correctness of extended compiling specifications. The original diagrams (a) (for $\mathcal{C}$ and $\mathscr{C}$) are commutative if and only if the corresponding extended diagrams (b) are.

Observe in Figure 12 (b), that we used $\sqsupseteq_v$ to indicate the difference between (horizontal) correct implementation $\sqsupseteq$ and (vertical) correct compiler implementation $\sqsupseteq_{\boldsymbol{v}}$ (see section 5.3.3). For $\sqsupseteq_v$ (respectively $\sqsupseteq_{\boldsymbol{v}}$) only preservation of relative (partial) correctness makes sense in practice, *i.e.*, commutativity of diagram (b) is only valid if $\sqsupseteq_v$ expresses correct relative (partial) implementation.

The extended $\mathcal{C}$ is a correct implementation of the extended $\mathscr{C}$ (Figure 13) and hence this step homogeneously fits on top of a stack of further commutative diagrams establishing correct transformation and implementation steps, all correctly tied together and related to $\mathscr{C}$ by vertical composability due to Theorem 4.2 and Corollary 4.1 from section 4. As any compiler program, also the compiling specification $\mathcal{C}$ reflects design decisions. It already selects particular target programs from the set of all possible correct implementations.

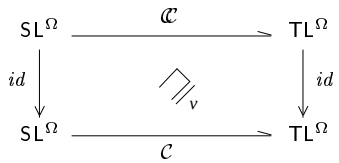


Figure 13. Correctness definition for extended compiling specifications

Theorem 5.1 (Correct compiling specifications). *A compiling specification $\mathcal{C}$ is correct, if and only if it is a correct implementation of $\mathscr{C}$.*

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

Only if: Let $\pi_s \in \mathsf{SL}$ and $\mathscr{C}(\pi_s) \subseteq \mathsf{TL} \cup A_{\mathsf{TL}}$ (*), and let $\pi_t \in \mathcal{C}(\pi_s)$. We have to show: $\pi_t \in \mathscr{C}(\pi_s) \cup A_{\mathsf{TL}}$. First note that π_s is well-formed, *i.e.*, has semantics $[\![\pi_s]\!]_{\mathsf{SL}} \in \mathrm{Sem}_{\mathsf{SL}}$, because otherwise π_t and $\mathscr{C}(\pi_s)$ would be the unacceptable error *nwf* $\notin \mathsf{TL} \cup A_{\mathsf{TL}}$ which contradicts (*). If $\pi_t \in A_{\mathsf{TL}}$, then we are done. $\pi_t \in U_{\mathsf{TL}}$ is impossible, because π_t would be *nwf* and hence π_s not well-formed. So let $\pi_t \in \mathsf{TL}$. Then, $([\![\pi_s]\!]_{\mathsf{SL}}, \pi_t) \in ([\![\cdot]\!]_{\mathsf{SL}}^{-1}; \mathcal{C})$. Due to correctness of $\mathcal{C}$ (Figure 12 (a)) we have $([\![\pi_s]\!]_{\mathsf{SL}}, \pi_t) \in (\sqsubseteq; [\![\cdot]\!]_{\mathsf{TL}}^{-1})$, that is to say $(\pi_s, \pi_t) \in ([\![\cdot]\!]_{\mathsf{SL}}; \sqsubseteq; [\![\cdot]\!]_{\mathsf{TL}}^{-1})$. But the latter is exactly the definition of $\mathscr{C}$, hence we have $\pi_t \in \mathscr{C}(\pi_s)$. □

5.2 Correct Compiler Programs

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

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

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

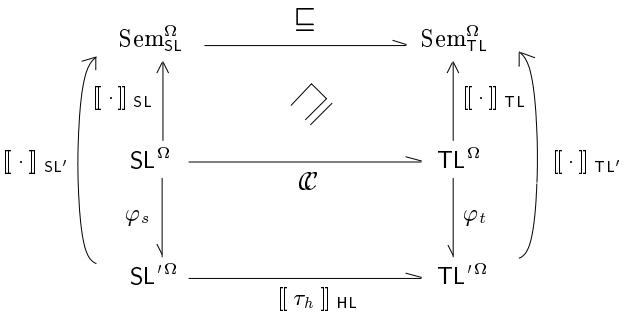


Figure 14. Compiler programs related to compiling specifications. If the lower diagram is commutative as well, we call τ_h a correct compiler program

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

Definition 5.2 (Correct compiler program). *We call τ_h a* correct compiler program, *iff $[\![\tau_h]\!]_{\mathsf{HL}} \sqsupseteq \mathscr{C}$, i.e., iff $[\![\tau_h]\!]_{\mathsf{HL}}$ is a correct implementation of $\mathscr{C}$.*

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

The proof of the latter remark is a bit more detailed, but analogous to that of Theorem 5.1: Let $\pi_s \in \mathsf{SL}$ and $\varphi^{\mathsf{TL}}_{\mathsf{TL}'}(\mathscr{C}(\pi_s)) \subseteq \mathsf{TL}' \cup A_{\mathsf{TL}'}$ (*), and let $\pi'_t \in [\![\pi_s]\!]_{\mathsf{SL}}(\varphi^{\mathsf{SL}}_{\mathsf{SL}'}(\pi_s))$. We have to show $\pi'_t \in \varphi^{\mathsf{TL}}_{\mathsf{TL}'}(\mathscr{C}(\pi_s)) \cup A_{\mathsf{TL}'}$. First note that π_s is well-formed, *i.e.*, has semantics $[\![\pi_s]\!]_{\mathsf{SL}}$. Otherwise, $\mathscr{C}$ would assign $nwf \in U_{\mathsf{TL}}$ to π_s which contradicts (*). If $\pi'_t \in A_{\mathsf{TL}'}$, we are done. If $\pi'_t \in U_{\mathsf{TL}'}$, then by commutativity of the outer diagram $[\![\pi_s]\!]_{\mathsf{SL}} \in (\sqsubseteq; [\![\cdot]\!]_{\mathsf{TL}'})^{-1}(U_{\mathsf{TL}'}) = \{uds\}$ contradicting well-formedness of π_s. So let $\pi'_t \in \mathsf{TL}'$. Commutativity of the outer diagram means $([\![\cdot]\!]^{-1}_{\mathsf{SL}'}; [\![\pi_h]\!]_{\mathsf{HL}})([\![\pi_s]\!]_{\mathsf{SL}}) \subseteq (\sqsubseteq; [\![\cdot]\!]^{-1}_{\mathsf{TL}'})([\![\pi_s]\!]_{\mathsf{SL}}) \cup A_{\mathsf{TL}'}$ and therefore $\pi'_t \in (\sqsubseteq; [\![\cdot]\!]^{-1}_{\mathsf{TL}'})([\![\pi_s]\!]_{\mathsf{SL}}) = \varphi^{\mathsf{TL}}_{\mathsf{TL}'}([\![\cdot]\!]^{-1}_{\mathsf{TL}}(\sqsubseteq([\![\pi_s]\!]_{\mathsf{SL}})))$. Since π'_t and $[\![\pi_s]\!]_{\mathsf{SL}}$ are regular in TL' resp. $\mathrm{Sem}_{\mathsf{SL}}$, we have $\pi'_t \in \varphi^{\mathsf{TL}}_{\mathsf{TL}'}(\mathscr{C}(\pi_s))$ by definition of $\mathscr{C}$. □

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

$$[\![\tau_h]\!]_{\mathsf{HL}} \sqsupseteq \mathcal{C}_\tau \sqsupseteq \mathscr{C}.$$

If $[\![\tau_h]\!]_{\mathsf{HL}}$ is a correct implementation of any correct specification $\mathcal{C}$, *i.e.*,

$$[\![\tau_h]\!]_{\mathsf{HL}} \sqsupseteq \mathcal{C} \sqsupseteq \mathscr{C},$$

then τ_h is a correct compiler program (cf. Figure 15).

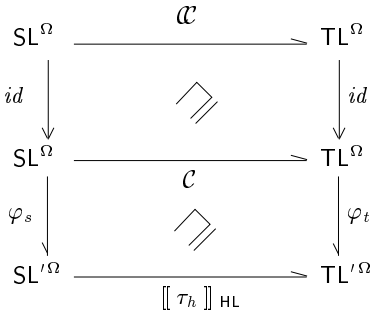


Figure 15. Compiler programs and compiling specifications. Due to vertical composition, a correct implementation of a correct compiling specification is a correct compiler

That is to say: A compiler program is correct, if and only if it is the correct implementation of a correct compiling specification.

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

Theorem 5.2. *Let τ_h be a correct compiler program and let $\pi'_s \in \varphi_s(\pi_s)$ be the representation of a well-formed* SL*-program. Then any regular $\pi'_t \in [\![\tau_h]\!]_{\mathsf{HL}}(\pi'_s)$ represents a correct implementation π_t of π_s.*

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

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

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

Let us call a concrete SL'- or TL'-datum π'_s or π'_t a well-formed SL'- or TL'-program, if it represents a well-formed SL- or TL-program. Then π'_s or π'_t have semantics $[\![\,\pi'\,]\!]_{\mathsf{SL}'}$ respectively $[\![\,\pi'\,]\!]_{\mathsf{TL}'}$. Thus, it is defined when π'_t correctly implements π'_s. According to Theorem 5.2, every regular result $\pi'_t \in [\![\,\tau_h\,]\!]_{\mathsf{HL}}(\pi'_s)$ of a correct compiler τ_h, applied to a well-formed π'_s, correctly implements π'_s.

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

5.3 Discussion and First Summary

We want to summarize some important observations and give some additional explanations. In particular, we will relate the definitions and notions defined in the previous sections to our informal sketch of a conscientious compiler correctness proof as of section 3.1 and in particular of Figure 2. Moreover, we will discuss McKeeman's T-diagram notation, give some remarks on the difference between correct implementation of user programs and of compiler programs, and finally we want to discuss correct implementation for optimizing compilers.

5.3.1 Precise View at the Three Steps

Let us first come back to the diagram shown in Figure 2 (page 18) in section 3.1. In the previous sections (4, 5.1 and 5.2) we have exactly defined every single notion mentioned in section 3.1 and, hence, we now know precisely every conjecture we have to prove in order to implement an SL to TL-compiler correctly as an executable program on a host processor HM.

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

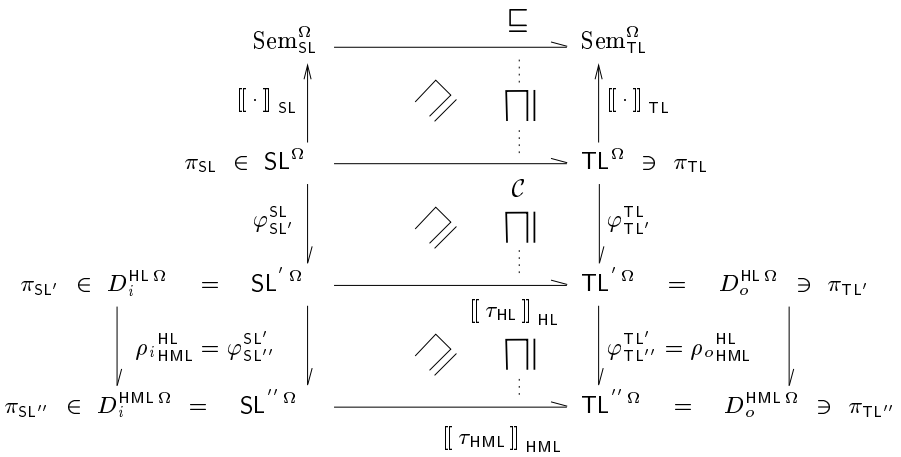


Figure 16. This diagram is again illustrating the three steps for correct compiler implementation as of Figure 2 on page 18

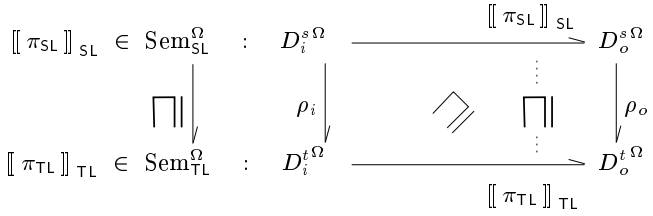


Figure 17. This diagram is again illustrating correct implementation as of Figure 4 on page 20. Note that $[\![\pi_{\mathsf{SL}}]\!]_{\mathsf{SL}} = [\![\pi_{\mathsf{SL}'}]\!]_{\mathsf{SL}'} = [\![\pi_{\mathsf{SL}''}]\!]_{\mathsf{SL}''}$ and analogously for TL

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

5.3.2 T-Diagram Notation

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

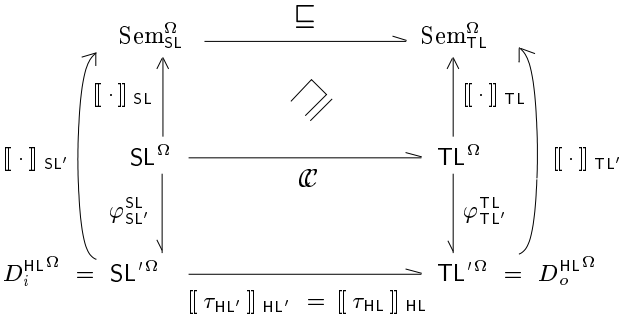


Figure 18. The situation which we will abbreviate by McKeeman's T-diagrams

Recall that $\mathscr{C}$ is the natural correct compiling specification from SL to TL. Well-formed programs and their (syntactical) φ-representations have equal semantics, and $\tau_{\mathsf{HL}'} \in \varphi^{\mathsf{HL}}_{\mathsf{HL}'}(\tau_{\mathsf{HL}})$ 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 19) as

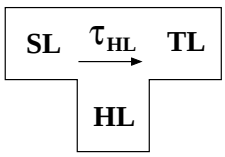


Figure 19. McKeeman's T-diagram as a short-hand for the above situation

an abbreviation for the diagram in Figure 18. 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.

5.3.3 Correct Implementation versus Correct Compiler Implementation

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

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

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

$$\emptyset \neq [\![\, \pi_{\mathsf{SL}} \,]\!]_{\mathsf{SL}} (d_i^s) \subseteq D_o^s \tag{11}$$

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

$$\emptyset \neq [\![\, \pi_{\mathsf{TL}} \,]\!]_{\mathsf{TL}} (d_i^t) \subseteq D_o^t \tag{12}$$

holds whenever the target program π_{TL} is applied upon the representation $d_i^t \in \rho_i (d_i^s)$, $d_i^s \notin [\![\, \pi_{\mathsf{SL}} \,]\!]_{\mathsf{SL}}^{-1} (U_o^s)$, of the corresponding input. Correct regular (total) implementation together with (11) guarantees (12), because due to section 4.1, we have

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

On the other hand, the same process programmer will (and in general has to) accept compile time error reports, like for instance HM-memory overflow, while π_{SL} is compiled to π_{TL}, *i.e.*, while the compiler machine implementation τ_{HML} is executed and applied upon a (syntactical) representation $\pi_{\mathsf{SL}''}$ of the source program π_{SL}. With respect to compilation, the process programmer wants a guarantee at execution time of $\pi_{\mathsf{TL}''}$ whenever τ_{HML} has succeeded on HM and has generated the target program representation $\pi_{\mathsf{TL}''}$, which means that (12) is established by successful execution of the compiler implementation τ_{HML}.

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

5.3.4 Notes on Optimizing Compilers

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

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

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

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

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

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

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

A well-known optimization is the so-called *redundant (dead) code elimination*, which might violate preservation of relative (partial) correctness, *e.g.*, which might eliminate the code that for some input data $d_i^s \in D_i^s$ would cause a runtime error like for instance a division by zero. The source program π_s might be partially (relatively) correct w.r.t. some pre- and post-conditions $\tilde{\Phi}$ resp. $\tilde{\Psi}$, whereas the optimized target program π_t is not. It might return a regular but incorrect result if applied to $d_i^t \notin \rho_i(\Phi)$. If the additional optimization pre-condition Φ does not hold for the input, π_t might dangerously deceive the user. Its result might have nothing in common with any regular source program result d_o^s.

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

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

tional expressions while transforming

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

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

6 Related Work on Compiler Verification

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

Compiling verification is part of theoretical informatics and program semantics and work on it has been started by J. McCarthy and J.A. Painter in 1967 [MP67]. Proof styles are operational [MP67, BR92, BS98] or denotational [MS76] depending on how source language semantics is defined. If a source language has loops or (function) procedures, then term rewriting or copy rule semantics is employed throughout [Lan73, LS87]. Other operational styles split in *natural* [NN92] or *structural* [Plo81] operational or *state-machine-like* [Gur91, Gur95]. Denotational semantics has started with D. Scott's work [Sco70, SS71], and typical compiling correctness proofs can be found in [MS76]. The authors in [HJS93, Sam93, MO97, MOW00] use an algebraic denotational style for clearer modular proofs, based on state transformations resp. predicate transformers.

Mechanical proofs are often based on interpreter semantics, a further variant of the operational style [Sto77], and sometimes include high level compiler implementation verification (step 2) with *e.g.*, HL = Stanford-Pascal [Pol81] or Boyer/Moore-Lisp [Moo88, Fla92, Moo96] or Standard-ML [Cur93, Cur94]. M. Broy [Bro92] uses the Larch-prover [GG91]. One should keep in mind, however, that the running theorem prover implementations are, strictly speaking, not completely verified. Their correctnesses again depend on existing correct initial host compilers, which are not available up to now. Recalling section 1, hackers might have intruded Trojan horses [Tho84, Goe00a] via unverified start up compilers. Hence, we are left on human checkability of mechanical proof protocols (a-posteriori-proof checking).

High level compiler implementation verification (step 2) is a field within software engineering. Correct implementation rules have been worked out in many formal software engineering methods and projects like VDM [Jon90], RAISE [GHH+92], CIP

[Bau78, Par90], PROSPECTRA [HKB93], Z [Spi92], B [ALN+91], and also the PVS-system [Dol00].

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 [Tho84], L.M. Chirica and D.F. Martin [CM86] and J S. Moore [Moo88, Moo96], 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.

7 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" [Tho84]:

> "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 [Wir77], 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 [Lan97b, Goe96a, Hof98].

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 [Goe99], 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.

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

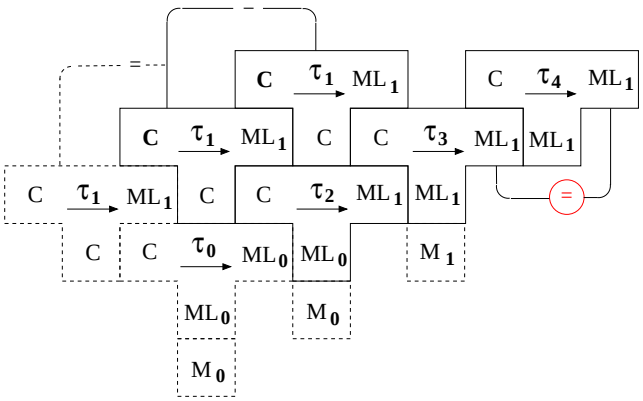


Figure 20. Wirth's strong compiler test. Note: Even if M_0 and M_1 are the same machine, τ_0 and τ_2 need not necessarily generate identical code. τ_0 and τ_2 are two different compilers. In general, we know nothing about τ_0's target code.

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 [Goe00a] – 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 [Goe00a, Goe00b]. 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 $\overline{\tau_3}$ on 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.

8 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 7), 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 [GH98b].

We identify source and high level implementation language SL = HL and take a proper sublanguage ComLisp [GH96] 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 $\mathsf{TL}_0 = \mathsf{HML}_0$, so that our initial two bootstrapping steps together are a cross-compilation to TL-code using the workstation as host machine.

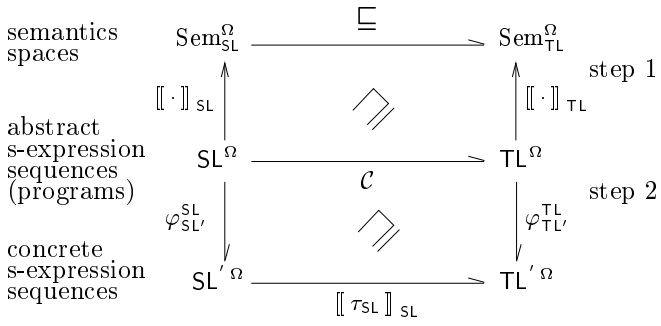


Figure 21. Steps 1 and 2 for an initial full correct compiler implementation

According to step 1 and 2 (section 3.1, page 18) we develop a correct SL to TL-compiling specification $\mathcal{C}$ 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 $\varphi^{\mathsf{SL}}_{\mathsf{SL}'}$ and $\varphi^{\mathsf{TL}}_{\mathsf{TL}'}$. We formulate $\mathcal{C}$ such that it will relate target programs π_{TL} to well-formed source programs π_{SL} at most, *i.e.*, for which source semantics $[\![\pi_{\mathsf{SL}}]\!]_{\mathsf{SL}}$ are defined. We construct τ_{SL} such that, if applied to well-formed source programs in SL', τ_{SL}, or more precisely $[\![\tau_{\mathsf{SL}}]\!]_{\mathsf{SL}}$, 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, $[\![\,\tau_{\mathsf{SL}}\,]\!]_{\mathsf{SL}}$ 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 $\mathscr{C}$, page 28).

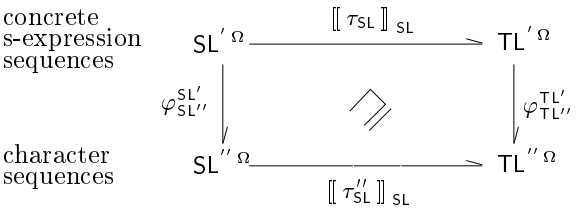


Figure 22. A correct compiler τ''_{SL} mapping character sequences in SL" to character sequences in TL"

We will also need a version τ''_{SL} which works on machine representations of s-expressions, for instance on character sequences (strings, cf. Figure 22), which form a particular subset of all s-expression sequences. Then, $\varphi^{\mathsf{SL}'}_{\mathsf{SL}''} = \varphi^{\mathsf{TL}'}_{\mathsf{TL}''}$ 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 and returns a list of s-expressions. Analogously the output is programmed using character output. $\tau''_{\mathsf{SL}''}$ denotes a string version representing τ''_{SL}, *i.e.*, $(\tau''_{\mathsf{SL}}, \tau''_{\mathsf{SL}''}) \in \varphi^{\mathsf{SL}}_{\mathsf{SL}''}$.

Let us now (unrealistically) assume for a moment, that τ_0, $\mathbf{M}_0$'s (the workstation's) Lisp-compiler, has been correctly developed and correctly implemented in TL_0. Then a threefold bootstrapping (Figure 23) of τ_{SL} (often of a byte-sequence or string represen-

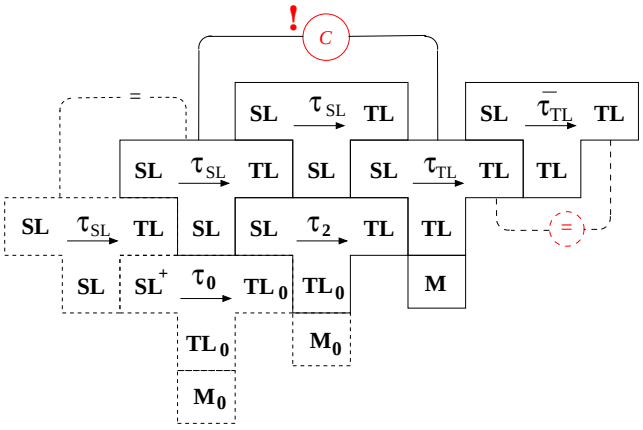


Figure 23. Bootstrapping the initial compiler. A sufficient variant of Wirth's strong compiler test (*a-posteriori-syntactical code inspection*) guarantees fully correct implementation

tation) eventually generates $\overline{\tau}_{\mathsf{TL}}$, provided that the three compiler executions of τ_0, τ_2, and τ_{TL} successfully terminate with regular outputs τ_2, τ_{TL}, and $\overline{\tau}_{\mathsf{TL}}$, respectively. Since τ_{SL} and τ_0 are assumed correct, and whenever the data abstractions ${\rho_i}^{\mathsf{SL}\,-1}_{\mathsf{TL}}$, ${\rho_o}^{\mathsf{SL}\,-1}_{\mathsf{TL}}$, ${\rho_i}^{\mathsf{SL}^+\,-1}_{\mathsf{TL}_0}$, and ${\rho_o}^{\mathsf{SL}^+\,-1}_{\mathsf{TL}_0}$ are single-valued functions, we have the following bootstrapping theorem [Lan97b, Goe99, Goe00c]:

Theorem 8.1 (Bootstrapping theorem). *The compilers τ_2, τ_{TL}, and $\overline{\tau}_{\mathsf{TL}}$ are correct, even correctly implemented on hardware processor $\mathbf{M}_0$ resp. $\mathbf{M}$. They are all correct refinements of $\mathcal{C}$ and τ_{SL}.*

In our case ${\rho_i}_{\mathsf{TL}}^{\mathsf{SL}\,-1}$, ${\rho_i}_{\mathsf{TL}_0}^{\mathsf{SL}^+\,-1}$, ${\rho_o}_{\mathsf{TL}}^{\mathsf{SL}\,-1}$ and ${\rho_o}_{\mathsf{TL}_0}^{\mathsf{SL}^+\,-1}$ are single-valued functions; actually, they are just the same as $\varphi_{\mathsf{SL}''}^{\mathsf{SL}'\,-1}$, $\varphi_{\mathsf{SL}''+}^{\mathsf{SL}'^+\,-1}$, $\varphi_{\mathsf{TL}''}^{\mathsf{TL}'\,-1}$, and $\varphi_{\mathsf{TL}_0''}^{\mathsf{TL}_0'\,-1}$, respectively.

We prove Theorem 8.1 by successive application of a bootstrapping lemma: Let τ_0 and τ_1[4] be two correct compilers and let us apply τ_0 to τ_1:

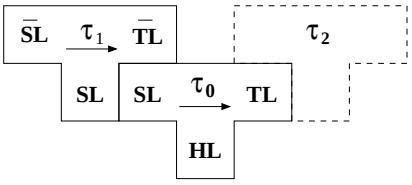


Figure 24. Bootstrapping an initial compiler implementation τ_2

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 25 for τ_0, and Figure 26 for τ_1).

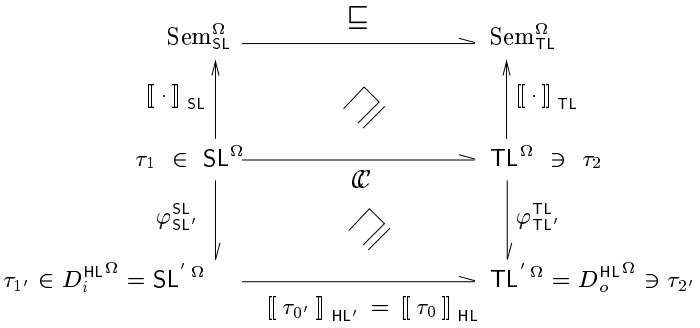


Figure 25. Extended view at Figure 24 (part 1 for τ_0)

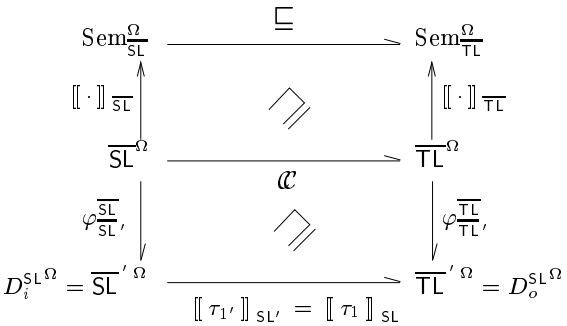


Figure 26. Extended view at Figure 24 (part 2 for τ_1)

Moreover, τ_0 is correct, and the result of applying τ_0 to $\tau_{1'}$ is the regular HL-datum $\tau_{2'} \in D_o^{\mathsf{HL}\Omega} = \mathsf{TL}'^{\Omega}$. Thus, $[\![\, \tau_{1'} \,]\!]_{\mathsf{SL}'} \sqsubseteq [\![\, \tau_{2'} \,]\!]_{\mathsf{TL}'}$, which means $\rho_i \,;\, [\![\, \tau_{2'} \,]\!]_{\mathsf{TL}'} \sqsupseteq [\![\, \tau_{1'} \,]\!]_{\mathsf{SL}'} \,;\, \rho_o$. Hence, also the diagram in Figure 27 is commutative and, thus, vertical composition of the diagrams in Figure 26 and in Figure 27 finally yields the following bootstrapping lemma:

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

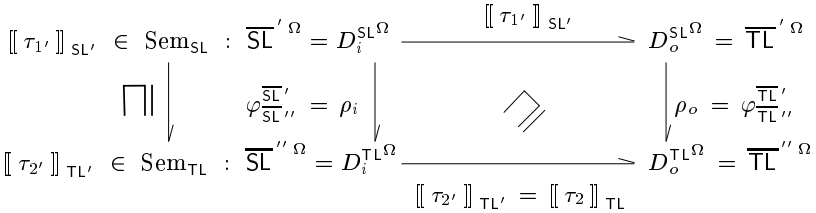


Figure 27. Commutative diagram corresponding to $[\![\,\tau_{1'}\,]\!]_{\mathsf{SL}'} \sqsubseteq [\![\,\tau_{2'}\,]\!]_{\mathsf{TL}'}$

Lemma 8.1 (Bootstrapping lemma). *If* $(\rho_i{}^{-1}; [\![\cdot]\!]_{\overline{\mathsf{SL}}})$ *and* $(\rho_o{}^{-1}; [\![\cdot]\!]_{\overline{\mathsf{TL}}})$ *are reasonable semantics functions (e.g., if* $\rho_i{}^{-1}$, $\rho_o{}^{-1}$ *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 concretions* $\overline{\mathsf{SL}}''$ *and* $\overline{\mathsf{TL}}''$ *of the programming languages* $\overline{\mathsf{SL}}$, $\overline{\mathsf{SL}}'$ *and* $\overline{\mathsf{TL}}$, $\overline{\mathsf{TL}}'$, *and in case of regular termination of* $\tau_{0'}$ *applied to* $\tau_{1'}$, $[\![\,\tau_{2'}\,]\!]_{\mathsf{TL}'} = [\![\,\tau_2\,]\!]_{\mathsf{TL}}$ *is a correct implementation of* $[\![\,\tau_{1'}\,]\!]_{\mathsf{SL}'} = [\![\,\tau_1\,]\!]_{\mathsf{SL}}$ *and by vertical composition also of* $\mathfrak{C}$. *Thus,* $\tau_{2'}$ *and* τ_2 *are correct compiler programs with the T-diagram shown in Figure 28.*

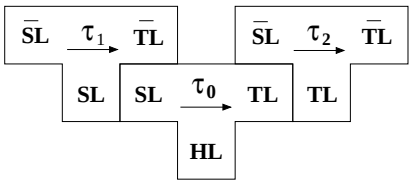


Figure 28. If τ_0 and τ_1 are correct compilers, then so is τ_2

Let us start a minor detour and come back to the discussion in section 5.3.3: 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 $\pi_{\mathsf{SL}'}$ in SL', even to a well-formed compiler like $\tau_{1'}$, might run into an acceptable error in $A_{\mathsf{TL}'} = A_o^{\mathsf{HL}}$, 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 $\overline{\mathsf{SL}}$ to $\overline{\mathsf{TL}}$-compiler. In other words: Since we cannot expect $\tau_{0'}$ to yield regular results for all (arbitrarily large) source program respresentations, $\tau_{0'}$ 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 (cf. section 4).

Thus, the expected error behaviors of $\overline{\mathsf{SL}}$-programs and in particular of $\tau_{1'}$ and hence also of $\tau_{2'}$, on the one hand, might be of quite a different nature than, on the other hand, of SL' and hence in particular of $\tau_{0'}$. $\overline{\mathsf{SL}}$ might even be a process programming language such that we expect $\overline{\mathsf{SL}}$-programs and their implementations in $\overline{\mathsf{TL}}$ to be regularly (totally) correct. If $\tau_{1'}$ preserves regular (total) program correctness, then so will $\tau_{2'}$, *i.e.*, the result of applying $\tau_{0'}$ to $\tau_{1'}$ (cf. section 5.3.3).

That is to say: A partial correctness preserving compiler $\tau_{0'}$ may well generate a total correctness preserving compiler executable $\tau_{2'}$ from a corresponding total correctness preserving compiler source program $\tau_{1'}$. 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 [Lan97b, Goe99, Goe00c]). Whereas the proof of the bootstrapping theorem (8.1) 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 8.2 (Strong compiler test theorem). *The compilers τ_{TL} and $\overline{\tau}_{\mathsf{TL}}$ 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 $\tau_{\mathsf{SL}'}$ of (the well-formed abstract s-expression) τ_{SL}. $\tau_{\mathsf{SL}'}$ is also a representation 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 $\tau_{\mathsf{SL}'}$ have the same semantics $[\![\,\tau_{\mathsf{SL}}\,]\!]_{\mathsf{SL}} = [\![\,\tau_{\mathsf{SL}}\,]\!]_{\mathsf{SL}^+} = [\![\,\tau_{\mathsf{SL}'}\,]\!]_{\mathsf{SL}'} = [\![\,\tau_{\mathsf{SL}'}\,]\!]_{\mathsf{SL}^{+\prime}}$.

Let $\tau_{2'}$ be the concrete result of successfully applying τ_0 to $\tau_{\mathsf{SL}'}$ on host machine M_0. Due to the proof of the bootstrapping lemma we have the following commutative diagram:

$$\begin{array}{ccc}
\mathsf{SL}'^{\,\Omega} = D_i^{\mathsf{SL}\,\Omega} = D_i^{\mathsf{SL}^+\,\Omega} & \xrightarrow{\quad [\![\,\tau_{\mathsf{SL}'}\,]\!]_{\mathsf{SL}^{+\prime}} \quad} & D_o^{\mathsf{SL}^+\,\Omega} = D_o^{\mathsf{SL}\,\Omega} = \mathsf{TL}'^{\,\Omega} \\
\varphi_{\mathsf{SL}''}^{\mathsf{SL}'} = \rho_{i\,\mathsf{TL}_0}^{\mathsf{SL}^+} \Big\downarrow & \swarrow\!\!\!\swarrow & \Big\downarrow \rho_{o\,\mathsf{TL}_0}^{\mathsf{SL}^+} = \varphi_{\mathsf{TL}''}^{\mathsf{TL}'} \\
\tau_{\mathsf{SL}'} \in \mathsf{SL}''^{\,\Omega} = D_i^{\mathsf{TL}_0\,\Omega} & \xrightarrow[[\![\,\tau_{2'}\,]\!]_{\mathsf{TL}_0'} = [\![\,\tau_2\,]\!]_{\mathsf{TL}_0}]{} & D_o^{\mathsf{TL}_0\,\Omega} = \mathsf{TL}''^{\,\Omega} \ni \tau_{\mathsf{TL}''}
\end{array}$$

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 $\rho_{i\,\mathsf{TL}_0}^{\mathsf{SL}^+}$ and $\rho_{o\,\mathsf{TL}_0}^{\mathsf{SL}^+}$ are single valued functions.

Let $\tau_{\mathsf{TL}''}$ be the concrete successful result of compiling $\tau_{\mathsf{SL}'}$ by $\tau_{2'}$ again on the host machine M_0. Again, due to the proof of the bootstrapping lemma we have

$$\begin{array}{ccc}
\mathsf{SL}'^{\,\Omega} = D_i^{\mathsf{SL}\,\Omega} & \xrightarrow{\quad [\![\,\tau_{\mathsf{SL}'}\,]\!]_{\mathsf{SL}'} \quad} & D_o^{\mathsf{SL}\,\Omega} = \mathsf{TL}'^{\,\Omega} \\
\varphi_{\mathsf{SL}''}^{\mathsf{SL}'} = \rho_{i\,\mathsf{TL}}^{\mathsf{SL}} \Big\downarrow & \swarrow\!\!\!\swarrow & \Big\downarrow \rho_{o\,\mathsf{TL}}^{\mathsf{SL}} = \varphi_{\mathsf{TL}''}^{\mathsf{TL}'} \\
\tau_{\mathsf{SL}''} \in \mathsf{SL}''^{\,\Omega} = D_i^{\mathsf{TL}\,\Omega} & \xrightarrow[[\![\,\tau_{\mathsf{TL}''}\,]\!]_{\mathsf{TL}''} = [\![\,\tau_{\mathsf{TL}}\,]\!]_{\mathsf{TL}}]{} & D_o^{\mathsf{TL}\,\Omega} = \mathsf{TL}''^{\,\Omega} \ni \overline{\tau}_{\mathsf{TL}''}
\end{array}$$

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

□

We can not prove equality of $\tau_{\mathsf{TL}''}$ and $\overline{\tau}_{\mathsf{TL}''}$. Equality does in general not hold, because the code of $\tau_{\mathsf{TL}''}$ and of $\overline{\tau}_{\mathsf{TL}''}$ has been generated by two different code generation

mechanisms of τ_2' on M0 and of $\tau_{\mathsf{TL}'}$ on M, *i.e.*, on two different machines. Code generation of $\tau_{\mathsf{TL}''}$ is influenced by runtime-system code which τ_0, *e.g.*, the existing SL^+ = ANSI-Common Lisp compiler running on M0, generates as part of τ_2', whereas code generation of $\overline{\tau}_{\mathsf{TL}''}$ is influenced by our (the τ_{SL} runtime-system generated as part of $\tau_{\mathsf{TL}''}$ by τ_2'.

Let us for instance assume $\tau_{\mathsf{TL}''}$ and $\overline{\tau}_{\mathsf{TL}''}$ to be character sequence (string) representations of τ_{TL} resp. $\overline{\tau}_{\mathsf{TL}}$. Since there are different correct string representations of one and the same s-expression, the M0-print routine of τ_2' might print $\tau_{\mathsf{TL}''}$ differently to how $\tau_{\mathsf{TL}'}$ prints $\overline{\tau}_{\mathsf{TL}''}$, although both printed represenations are correct with respect to the identical s-expressions $\tau_{\mathsf{TL}'}$ resp. $\overline{\tau}_{\mathsf{TL}'}$.

However, we could add a further bootstrapping step comparing the string results of τ_{TL} and $\overline{\tau}_{\mathsf{TL}}$. Since we would then use the same print routine, and because our programs are determinstic, we would finally get equal string representations $\overline{\tau}_{\mathsf{TL}''} = \overline{\overline{\tau}}_{\mathsf{TL}''}$, if the bootstrap succeeds. For that we exploit that $\tau_{\mathsf{TL}''}, \tau_{\mathsf{TL}''}, \overline{\tau}_{\mathsf{TL}}, \overline{\tau}_{\mathsf{TL}''}$ 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 $\tau_{\mathsf{TL}''}$ resp. $\overline{\tau}_{\mathsf{TL}''}$ of $\tau_{\mathsf{TL}} = \overline{\tau}_{\mathsf{TL}}$. 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 M0 to $\tau''_{\mathsf{SL}''}$. Recall that the latter is a version of our compiler source program τ_{SL} which does not print s-expressions (by using the runtime-system of the implementing compiler) but comprises its own printing routine and hence generates character sequences. In that case, both programs deterministically compute the same sequence of characters and we would have $\tau''_{\mathsf{TL}''} = \overline{\tau}''_{\mathsf{TL}''}$, although the two strings are computed on different machines:

$$\tau''_{\mathsf{TL}'} = [\![\, \tau''_{\mathsf{SL}''} \,]\!]_{\mathsf{SL}^{+\prime}} (\varphi^{\mathsf{SL}'^{-1}}_{\mathsf{SL}''} (\tau''_{\mathsf{SL}''})) = [\![\, \tau''_{\mathsf{SL}''} \,]\!]_{\mathsf{SL}'} (\varphi^{\mathsf{SL}'^{-1}}_{\mathsf{SL}''} (\tau''_{\mathsf{SL}''})) = \overline{\tau}''_{\mathsf{TL}'}$$

due to determinism of τ''_{SL} resp. $\tau''_{\mathsf{SL}''}$. 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 M0 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 it precisely.

9 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 ($\tau_{\mathsf{TL}} = \overline{\tau}_{\mathsf{TL}}$) 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 [Goe99, Goe00a, Goe00b] even in this case: Although $\overline{\tau}_{\mathsf{TL}}$ 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 $\overline{\tau}_{\mathsf{TL}}$ (and also in τ_2).

In [Goe00a, Goe00b], we prove this fact mechanically using M. Kaufmann's and J Moore's logic and theorem prover ACL2 [KM94]. 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).

10 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. $\tau''_{\mathsf{SL}''}$ on machine 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 23. 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 $\tau''_{\mathsf{SL}}\, \mathcal{C}\, \tau''_{\mathsf{TL}}$ holds, *i.e.*, let τ''_{SL}, τ''_{TL} fulfil the compiling specification which was verified in step 1 (Figure 21). Hence, τ''_{SL}'s and τ''_{TL}'s semantics are related by $\sqsubseteq$.

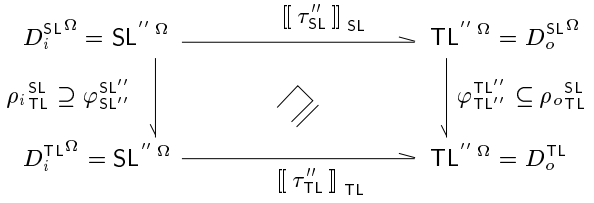


Figure 29. Correct implementation of correct compiler source programs. The extended semantics, defined on $D_i^{\mathsf{SL}\Omega}$ resp. $D_i^{\mathsf{TL}\Omega}$, carries an unacceptable error outcome in U_i^{SL} resp. U_i^{TL} indicating the cases where inputs do not represent well-formed SL- or TL-programs

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

used in SL and TL.

Since ${\rho_i^{\mathsf{SL}}}^{-1} = {\rho_o^{\mathsf{SL}}}^{-1}$ is single-valued, representations of well-formed programs have unique semantics in $\mathrm{Sem}_{\mathsf{SL}}$ resp. $\mathrm{Sem}_{\mathsf{TL}}$. Due to vertical composability (cf. section 4.4, Theorem 4.2), τ''_{TL} is a correct SL" to TL"- compiler correctly implemented in TL. Figure 29 accomplishes Figure 21 and 22 (page 39) and yields Figure 2 (page 18) 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 $[\![\cdot]\!]_{\mathsf{L}} \in \mathrm{Sem}_{\mathsf{L}}$.

Compiling specifications $\mathcal{C}$ 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 10.1 (Semantics to Syntax Reduction). *If s is a string, if $\tau''_{\mathsf{TL}} = (\varphi^{\mathsf{TL}}_{\mathsf{TL}''}; \varphi^{\mathsf{TL}''}_{\mathsf{TL}''})^{-1}(s)$ and if syntactical checking of $\tau''_{\mathsf{SL}}\, \mathcal{C}\, \tau''_{\mathsf{TL}}$ 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 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 7.

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 [BLR89], trusted compilation [Lan97a, GH98b, PSS98, HGG+99, GZG99, CGP+97], and systems verification [GGZ98, PT99, BG01] in general.

10.1 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 *circuli vitiosi*. 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 circuli vitiosi. Since source code verification may succeed, and since manual machine code verification hardly ever will, we strongly believe that providing trustworthy compiler executables is the most promising sound basis.

Verifix has introduced [GH98b, Hof98] 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. $\mathcal{C}^{L_i}_{L_{i+1}}$ 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 10.2 below. Source, intermediate and target languages are:

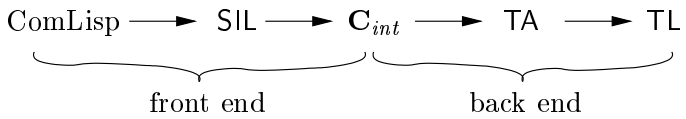
1. High level source language is ComLisp = SL. Programs consist of non-nested mutually recursive function procedures with call-by-value parameter passing. Variables are simple, and data are Lisp-s-expressions. Denotational, operational copy rule resp. stack semantics are well-known [LS87, NN92, MO90, Goe97].
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 procedure and operator parameters. Operational stack semantics is straight forward and easily comparable to the operational SL-semantics.
3. C-like intermediate language $\mathbf{C}_{int}$ 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 $\mathbf{C}_{int}$ 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 [GH98c]. 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 undefinednesses. This is guaranteed by compiling specification verification (step 1).

10.2 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 [GH98b, GH98c, GH98a]. 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.

10.2.1 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 transformation 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[5]. 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. $\mathcal{CL}_{def}[\![\,(\texttt{DEFUN}\ p\ (p_1\ \ldots\ p_k)\ f_1\ \ldots f_m)\,]\!]_{\gamma}$ $\supseteq_{\text{def}}$ (`DEFUN` p $\mathcal{CL}_{prog}[\![\,f_1\ \ldots f_m\,]\!]_{\rho,\gamma,k}$ (`_COPY` n 0))

   where $\rho(p_i) = i-1$ for each $i = 1,\ldots,k$

2. $\mathcal{CL}_{prog}[\![\,f_1\ \ldots f_m\,]\!]_{\rho,\gamma,k}$ $\supseteq_{\text{def}}$ $\mathcal{CL}_{form}[\![\,f_1\,]\!]_{\rho,\gamma,k}$ ... $\mathcal{CL}_{form}[\![\,f_m\,]\!]_{\rho,\gamma,k}$

   where $m \geq 1$

3. $\mathcal{CL}_{form}[\![\,(p\ f_1\ \ldots\ f_n)\,]\!]_{\rho,\gamma,k}$ $\supseteq_{\text{def}}$ $\mathcal{CL}_{form}[\![\,f_1\,]\!]_{\rho,\gamma,k}$ ... $\mathcal{CL}_{form}[\![\,f_n\,]\!]_{\rho,\gamma,k+n-1}$ $(p\ k)$

4. $\mathcal{CL}_{form}[\![\,c\,]\!]_{\rho,\gamma,k}$ $\supseteq_{\text{def}}$ (`_COPY` c k)

   where c is a constant integer, character, string or symbol `NIL` or `T`

5. $\mathcal{CL}_{form}[\![\,v\,]\!]_{\rho,\gamma,k}$ $\supseteq_{\text{def}}$ (`_COPY` $\rho(v)$ k)

   where v is a local variable or formal parameter with $\rho(v)$ defined $\in \mathbb{N}_0$

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

We present the compiling specification rules in the style of a conditional term rewrite system (for details see [GH98c, Hof98]). 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. $\texttt{<program>}_{\mathsf{SIL}}$, $\texttt{<declarations>}_{\mathsf{SIL}}$, $\texttt{<form>}_{\mathsf{SIL}}$. Ground terms are augmented by rewrite variables[6] and unary rewrite operators like $\mathcal{CL}_{def}[\![\cdot]\!]_{\gamma}$, $\mathcal{CL}_{prog}[\![\cdot]\!]_{\rho,\gamma,k}$ or $\mathcal{CL}_{form}[\![\cdot]\!]_{\rho,\gamma,k}$ 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.

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 $\mathbf{C}_{int}$:

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

1. $\mathcal{CS}_{def}[\![\,(\texttt{DEFUN}\ p\ f_1\ \ldots f_m)\,]\!]_\xi$ $\supseteq_{\text{def}}$ $(\texttt{DEFUN}\ p\ (s)$
   $\mathcal{CS}_{form}[\![\,f_1\,]\!]_\xi$
   $\ldots$
   $\mathcal{CS}_{form}[\![\,f_m\,]\!]_\xi$

   where s is the maximal stack frame length needed by $f_1, \ldots, f_m$
2. $\mathcal{CS}_{form}[\![\,(\texttt{\_COPY}\ i\ j)\,]\!]_\xi$ $\supseteq_{\text{def}}$ $(\texttt{\_SETLOCAL}\ (\texttt{\_LOCAL}\ 2i)\ 2j)$
   $(\texttt{\_SETLOCAL}\ (\texttt{\_LOCAL}\ 2i+1)\ 2j+1)$
3. $\mathcal{CS}_{form}[\![\,(p\ i)\,]\!]_\xi$ $\supseteq_{\text{def}}$ $(p\ 2i)$
4. $\mathcal{CS}_{form}[\![\,(\texttt{\_COPYC}\ n\ i)\,]\!]_\xi$ $\supseteq_{\text{def}}$ $(\texttt{\_SETLOCAL}\ \tau\ 2i)$
   $(\texttt{\_SETLOCAL}\ n\ 2i+1)$

   where τ is the number tag 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.

10.2.2 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 $\mathbf{C}_{int}$-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 $\mathbf{C}_{int}$-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 from $\mathbf{C}_{int}$ to TA are the following:

1. $\mathcal{CC}_{def}[\![\,(\texttt{DEFUN}\ f\ (\boxed{\sigma})\ s_1\ \ldots s_n)\,]\!]_{\varphi}$ $\supseteq_{\text{def}}$ (_DEFCODE f $\psi(f)$
   ($entrycode(\boxed{\sigma})$)
   ($\mathcal{CC}_{stmt}[\![\,s_1\,]\!]_{\varphi,\sigma}$
   ...
   $\mathcal{CC}_{stmt}[\![\,s_n\,]\!]_{\varphi,\sigma}$)
   ($exitcode$)

   where $\varphi = \langle\psi, |stack_i|, |heap_i|\rangle$
   and ψ is a subroutine numbering

2. $\mathcal{CC}_{stmt}[\![\,(f\ i)\,]\!]_{\varphi,\sigma}$ $\supseteq_{\text{def}}$ LDC i LDL $start$ LDNL $\psi(f)$ OPR 6
   where $0 \leq i < \sigma$

3. $\mathcal{CC}_{stmt}[\![\,(\texttt{\_SETLOCAL}\ e\ i)\,]\!]_{\varphi,\sigma}$ $\supseteq_{\text{def}}$ $\mathcal{CC}_{expr}[\![\,e\,]\!]_{\varphi,\sigma}$ LDL $base$ STNL i
   where $0 \leq i < \sigma$

4. $\mathcal{CC}_{expr}[\![\,(\texttt{\_LOCAL}\ i)\,]\!]_{\varphi,\sigma}$ $\supseteq_{\text{def}}$ LDL $base$ LDNL i
   where $0 \leq i < \sigma$

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. $|stack_i|$ and $|heap_i|$ 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 have been transformed correctly to instruction byte sequences (cf. Figure 30). 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
  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:

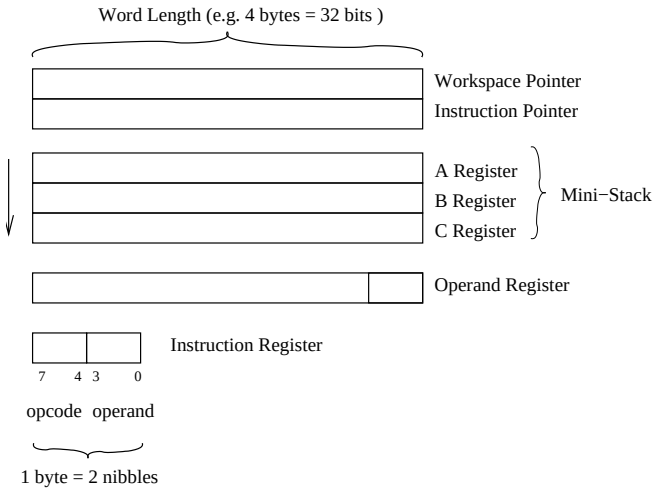
1. $\mathcal{CA}_{def}[\![\,(\texttt{\_DEFCODE}\ f\ i\ b)\,]\!] \quad \supseteq_{\text{def}} \quad (i_{\#x}\ \texttt{z}\ |c|_{\#x}\ c)$
   where $c = \mathcal{CA}_{body}[\![\,b\,]\!]$ and $i_{\#x}, |c|_{\#x}$
   denote hexadecimal representations of $i, |c|$
2. $\mathcal{CA}_{body}[\![\,op_1\ e_1\ \ldots\ (\ldots)\ \ldots\ (\ldots\ op_n\ e_n)\,]\!]$
   $\supseteq_{\text{def}} \quad \mathcal{CA}_{opr}[\![\,op_1\ e_1\,]\!]\ \ldots\ \mathcal{CA}_{opr}[\![\,op_n\ e_n\,]\!]$
3. $\mathcal{CA}_{opr}[\![\,op\ e\,]\!] \quad \supseteq_{\text{def}} \quad prefix(assemble\_op(op), e)$

In the second rule, $\mathcal{CA}_{body}$ 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 30, 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 $16 * 1 + 12 = 28$.

10.2.3 The Complete Proof Structure

This ends the more detailed look into the characteristics of our technique 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.



Direct Operation Codes

0x	0000	j	jump
1x	0001	ldlp	load local pointer
2x	0010	pfix	prefix
3x	0011	ldnl	load non local
4x	0100	ldc	load constant
5x	0101	ldnlp	load non local pointer
6x	0110	nfix	negative prefix
7x	0111	ldl	load local
8x	1000	adc	add constant
9x	1001	call	call
Ax	1010	cj	conditional jump
Bx	1011	ajw	adjust workspace
Cx	1100	eqc	equal to constant
Dx	1101	stl	store local
Ex	1110	stnl	store non local
Fx	1111	opr	operate

Figure 30. 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.

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 M0 for both front-end and both back-end phases.

The following large diagram (Figure 31) shows all four subcompiling specifications $\mathcal{C}_{L_i}^{L_{i-1}}$, all four hand-written subcompiler implementations $\tau''_{i,\mathsf{SL}} = \tau''_{i,L_1}$ 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

$$\tau''_{i,L_{j-1}} \; \mathcal{C}_{L_j}^{L_{j-1}} \; \tau''_{i,L_j} \tag{13}$$

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 10.1.
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 $\mathcal{C}_{\mathsf{TL}}^{\mathsf{TA}}$ 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 $\mathbf{C}_{int}$ 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 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 M and let M read $\tau''_{3,\mathsf{TA}''}$. If M terminates successfully (regularly), then due to Bootstrapping Theorem 8.1 the output is a correct compiler $\tau'''_{3,\mathsf{TL}}$ (actually

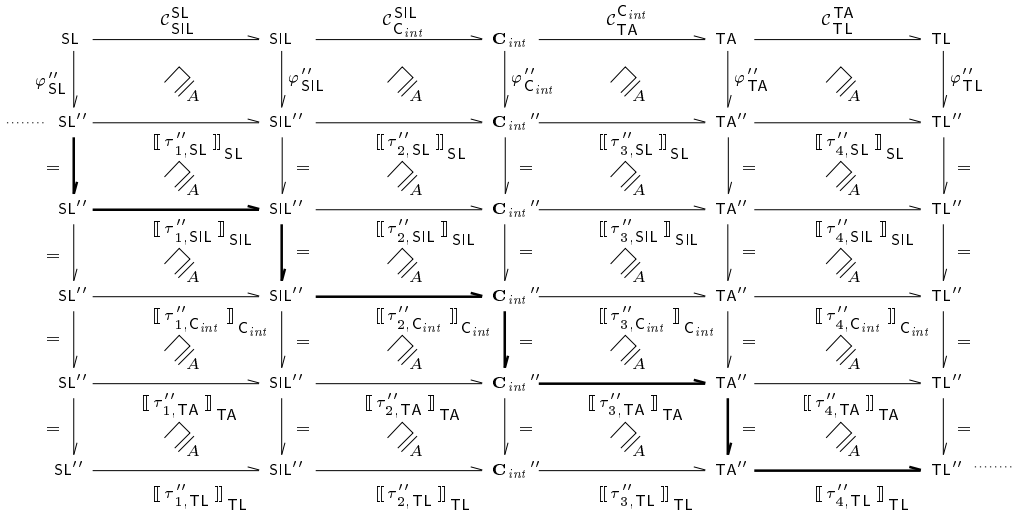


Figure 31. Special bootstrapping with four compiler phases. $\varphi''_{\mathsf{L}} =_{\text{def}} (\varphi^{\mathsf{L}}_{\mathsf{L}'} \,;\, \varphi^{\mathsf{L}'}_{\mathsf{L}''})$.

$\tau'''_{3,\mathsf{TL}''}$) as well, not necessarily identical to $\tau'''_{3,\mathsf{TL}}$, but according to $\mathcal{C}^{\mathsf{TA}}_{\mathsf{TL}}$.

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 $\mathbf{C}_{int}$" to TL" due to composability of commutative diagrams (Theorem 4.3). If we proceed, this process will finally generate the desired correct compiler executable from SL" to TL".

$$\tau'''_{\mathsf{TL}''} \quad = \quad \tau'''_{1,\mathsf{TL}''} \,;\, \tau'''_{2,\mathsf{TL}''} \,;\, \tau'''_{3,\mathsf{TL}''} \,;\, \tau''_{4,\mathsf{TL}''} \tag{14}$$

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 $\tau''_{i,\mathsf{SL}}$ 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 $\tau''_{4,\mathsf{TL}}$ resp. $\tau''_{3,\mathsf{TA}}$ 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 $\tau''_{0,\mathsf{SL}}$. 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[7], 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] Unfortunately, it seems again not to be sufficient to compare the initially loaded and the generated code to be identical (cf. section 7).

11 Conclusions

At the end of our exposé 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 solves 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 (sections 1.3,1.4,1.5 and 2).

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 (section 1.4).

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 or 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 6).

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, [JBFB01]). 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 bootstrapping 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 6).

But unverified compilers are well outside Wittgenstein's domain and they bear real risks (as shown by Ken Thompson [Tho84] and later by ourselves [Goe99, Goe00a, Goe00b]). 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 7). 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 (sections 4 and 8 to 10). 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 10).

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 8). 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 10).

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 (sections 4 and 5). 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.

References

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

[Bau78] F. L. Bauer. Program development by stepwise transformations – the project CIP. In F. L. Bauer and M. Broy, editors, *Program Construction*, volume 69 of *Lecture Notes in Computer Science*, pages 237–266. Springer Verlag, 1978.

[BG01] R. Bartsch and W. Goerigk. Mechanical a-posteriori Verification of Results: A Case Study for a Safety Critical AI System (Abstract). In *AAAI Workshop on Model Based Validation of Intelligence MBVI'2001*, Stanford, CA, U.S.A., March 2001.

[BLR89] 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.

[BR92] E. Börger and D. Rosenzweig. The WAM-definition and Compiler Correctness. Technical Report TR-14/92, Dip. di informatica, Univ. Pisa, Italy, 1992.

[Bro92] M. Broy. Experiences with software specification and verification using LP, the larch proof assistant. Technical Report 93, Digital Systems Research Center, Palo Alto, July 1992.

[BS98] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *23rd International Symposium on Mathematical Foundations of Computer Science*, LNCS. Springer, 1998.

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

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

[CGP+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.

[CM86] 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.

[Cur93] P. Curzon. Deriving Correctness Properties of Compiled Code. *Formal Methods in System Design*, 3:83–115, August 1993.

[Cur94] P. Curzon. The Verified Compilation of Vista Programs. Internal Report, Computer Laboratory, University of Cambridge, January 1994.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Dol00] Axel Dold. *Formal Software Development using Generic Development Steps.* Logos-Verlag, Berlin, 2000. Dissertation, Universität Ulm.

[Fag86] M. E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, SE-12(7):744 – 751, 1986.

[Fla92] A. D. Flatau. *A verified implementation of an applicative language with dynamic storage allocation.* PhD thesis, University of Texas at Austin, 1992.

[GG91] S.J. Garland and J.V. Guttag. A guide to lp, the larch prover. Technical Report SRC Report 82, Digital systems Research 'Center, 1991.

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

[GGZ98] W. Goerigk, Th. Gaul, and W. Zimmermann. Correct Programs without Proof? On Checker-Based Program Verification. In *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computing Science, Malente, 1998. Springer Verlag.

[GH96] W. Goerigk and U. Hoffmann. The Compiler Implementation Language ComLisp. Technical Report Verifix/CAU/1.7, CAU Kiel, June 1996.

[GH98a] W. Goerigk and U. Hoffmann. Compiling ComLisp to Executable Machine Code: Compiler Construction. Technical Report Nr. 9812, Institut für Informatik, CAU, October 1998.

[GH98b] 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 *Lecture Notes in Computer Science*, pages 122 – 136, 1998.

[GH98c] 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.

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

[GL01a] W. Goerigk and H. Langmaack. Compiler Implementation Verification and Trojan Horses. In D. Bainov, editor, *Proc. 9th International Colloquium on Numerical Analysis and Computer Science with Applications*, Plovdiv, Bulgaria, 2001.

[GL01b] W. Goerigk and H. Langmaack. Compiler Implementation Verification and Trojan Horses (Extended Draft Version). Technical report, Institut für Informatik, CAU, 2001. Available under `http://www.informatik.uni--kiel.de/~wg/Berichte/Plovdiv.ps.gz`.

[Goe96a] 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.

[Goe96b] W. Goerke. Über Fehlerbewußtseinskultur. Mündlicher Diskussionsbeitrag, Informatik-Kolloquium, IBM, Böblingen, 1996. Unpublished, 1996.

[Goe97] W. Goerigk. A Denotational Semantics for ComLisp and SIL. Technical Report Verifix/CAU/2.8, CAU Kiel, December 1997.

[Goe99] W. Goerigk. On Trojan Horses in Compiler Implementations. In F. Saglietti and W. Goerigk, editors, *Proc. des Workshops* Sicherheit und Zuverlässigkeit softwarebasierter Systeme, ISTec Report ISTec-A-367, ISBN 3-00-004872-3, Garching, August 1999.

[Goe00a] 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.

[Goe00b] 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.

[Goe00c] W. Goerigk. Trusted Program Execution. Habilitation thesis. Techn. Faculty, Christian-Albrechts-Universität zu Kiel, May 2000. To be published.

[Gur91] Y. Gurevich. Evolving Algebras; A Tutorial Introduction. *Bulletin EATCS*, 43:264–284, 1991.

[Gur95] Y. Gurevich. Evolving Algebras: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[GZG99] 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.

[HGG+99] 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.

[HJS93] C. A. R. Hoare, He Jifeng, and A. Sampaio. Normal Form Approach to Compiler Design. *Acta Informatica*, 30:701–739, 1993.

[HKB93] B. Hoffmann and B. Krieg-Brückner. *Program Development by Specification and Transformation.* Springer, Berlin, 1993.

[Hof98] U. Hoffmann. *Compiler Implementation Verification through Rigorous Syntactical Code Inspection.* PhD thesis, Technische Fakultät der Christian-Albrechts-Universität zu Kiel, Kiel, 1998.

[JBFB01] 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.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM, 2nd ed.* Prentice Hall, New York, London, 1990.

[KM94] M. Kaufmann and J S. Moore. Design Goals of ACL2. Technical Report 101, Computational Logic, Inc., August 1994.

[Lan73] H. Langmaack. On Correct Procedure Parameter Transmission in Higher Programming Languages. *Acta Informatica*, 2(2):110–142, 1973.

[Lan97a] 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*, 1337:321–335, 1997.

[Lan97b] H. Langmaack. Softwareengineering zur Zertifizierung von Systemen: Spezifikations-, Implementierungs-, Übersetzerkorrektheit. *Informationstechnik und Technische Informatik it+ti*, 39(3):41–47, 1997.

[Lan97c] H. Langmaack. The ProCoS Approach to Correct Systems. *Real Time Systems*, 13:251–273, 1997.

[Lap94] J. C. Laprie, editor. *Dependability: basic concepts and terminology.* Springer Verlag for IFIP WG 10.4, 1994.

[LS87] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification (Second edition).* John Wiley and Sons, New York, N.Y., 1987.

[MO90] Markus Müller-Olm. Korrektheit einer Übersetzung der Sprache rekursiver Funktionsdefinitionen erster Ordnung in eine einfache imperative Sprache. Master's thesis, CAU Kiel, 1990.

[MO97] M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1997.

[Moo88] J S. Moore. Piton: A verified assembly level language. Technical Report 22, Comp. Logic Inc, Austin, Texas, 1988.

[Moo96] J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Kluwer Academic Press, Dordrecht, The Netherlands, 1996.

[MOW00] M. Müller-Olm and A. Wolf. On the Translation of Procedures to Finite Machines. In G. Smolka, editor, *Programming Languages and Systems. Proceedings of ESOP 2000*, volume 1782 of *LNCS*, pages 290–304, Berlin, March 2000.

[MP67] J. McCarthy and J. A. Painter. Correctness of a compiler for arithmetical expressions. In J.T. Schwartz, editor, *Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.

[MS76] R. Milne and Ch. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976.

[NN92] H. R. Nielson and F. Nielson. *Semantics with Applications : A Formal Introduction*. John Wiley & Sons, Chichester, 1992.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings 11th International Conference on Automated Deduction CADE*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, October 1992. Springer-Verlag.

[Par90] H. Partsch. *Specification and Transformation of Programs*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[PLG88] Eduardo Pelegrí-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application of BURS theory. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 15th Annual ACM Symposium on Principels of Programming Languages (POPL '88)*, pages 294–308, San Diego, CA, USA, January 1988. ACM Press.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[Pof95] E. Pofahl. Methods Used for Inspecting Safety Relevant Software. In W. J. Cullyer, W. A. Halang and B. J. Krämer (eds.): High Integrity Programmable Electronic Systems, Dagstuhl-Sem.-Rep. 107, p 13, 1995.

[Pol81] W. Polak. Compiler specification and verification. In J. Hartmanis G. Goos, editor, *Lecture Notes in Computer Science*, number 124 in LNCS. Springer-Verlag, 1981.

[PSS98] 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.

[PT99] A. Pnueli and P. Traverso, editors. *Proceedings of the FLoC'99 International Workshop on "Runtime Result Verification"*, Trento, Italy, 1999.

[Sam93] A. Sampaio. *An Algebraic Approach to Compiler Design.* PhD thesis, Oxford University Computing Laboratory, Programming Research Group, October 1993. Technical Monograph PRG–110, Oxford University Computing Laboratory.

[Sco70] D. S. Scott. Outline of a Mathematical Theory of Computation. In *Proceedings of the 4th Annual Princeton Conference on Information Sciences and Systems*, pages 169–176, Princeton, 1970.

[Spi92] J. M. Spivey. *The Z Notation.* Prentice-Hall International Series in Computer Science. Prentice Hall, 1992.

[SS71] D. Scott and Ch. Strachey. Toward a mathematical semantics for computer languages. pages 19–46. April 1971.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA., London, 1977.

[SV96] G. Schweizer and M. Voss. Systems engineering and infrastructures for open computer based systems. *Lecture Notes in Computer Science*, 1030:317–??, 1996.

[Tho84] 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.

[Wir77] N. Wirth. *Compilerbau, eine Einführung.* B.G. Teubner, Stuttgart, 1977.

[Wol01] A. Wolf. *Weakest Relative Precondition Semantics - Balancing Approved Theory and Realistic Translation Verification.* PhD thesis, Technische Fakultät der Christian-Albrechts-Universität, Report No. 2013, Kiel, February 2001.

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

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

List of Figures