What Level of Mathematical Reasoning can Computer Science Demand of a Software Implementer? ¹

Auf Deutsch: Welche Art mathematischer Argumentation darf die Informatikwissenschaft einem Softwareimplementierer auf jeden Fall zumuten?

Hans Langmaack

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
E-Mail: hl@informatik.uni-kiel.de

Abstract

The article starts out from the observation that software engineering splits in two large activity areas: Software specification with its verification and software implementation with its verification. To find answers to the question in the title the article studies a practical systems software engineering area where theory is better developed than compared to other areas: Compiler construction. Our answer is a conclusion from work in the DFG-project Verifix, U.Karlsruhe, U.Kiel, U.Ulm, 1995-2003. One very complex cooperational task has been construction of a so called initial correct compiler for a realistic high level programming (and compiler writing) language correctly implemented and executed on a real life host processor. The interface between compiling specification and compiler implementation is given by algebraic-style, conditional formula transformation or program term rewriting rules which the specifier figures out and must prove correct w. r. t. source program and target processor semantics and data and states representations. Intensive cooperation of compiling specifiers and compiler implementers has revealed that the implementer’s mathematical reasoning is algebraic reasoning of moderate depth. The specifier overtakes semantical issues and does induction proofs, a field of much more intricate mathematical reasoning.

Keywords: Compiler, specification, implementation, correctness, code inspection, term-rewriting, algebraic reasoning

¹ The investigations in this article were supported by the Deutsche Forschungsgemeinschaft DFG under Go 323/3-1,2,3, He 2411/2-1,2,3, La 426/15-1,2,3, La 426/16-1.
1 Motivation of the question

In 1968 at the Garmisch NATO-conference, F. L. Bauer created the notion of software engineering. Since the 1970s, schools like VDM [45], CIP (an algebraic school) [2,64], Z [70] or ASM [4] split software engineering in two large areas: Software specification and software implementation. Mathematical reasoning about software is so called software verification and splits in two disciplines: Software specification verification and software implementation verification. The first discipline means for transformational software: Mathematical proof of correctness of a specifying mapping from abstract input to abstract output data w.r.t. characterizing problem oriented properties. The second discipline means: Correctness proof of a translation of the specifying mapping to an implementing program of a correctly executable programming language. In case verification is done in one step the specifying mapping can be seen as given by the semantics of the implementing program. Questions: Do software specification and implementation verification require different levels, kinds and abilities of mathematical reasoning? Are there different levels of mathematical intricacy? What special kind of mathematics does software implementation verification require?

Because the area of software engineering is wide it seems to be wise firstly to restrict to software which is closer known to informaticians and software engineers: Systems software. So we shield application software the essential properties of which are defined not only by computer scientists but, above all, by application engineers and scientists. Second restriction: Among systems software there is one area which up to now has a theory better developed than other systems software areas have: Compiler construction. So in this lecture we investigate compiling specification verification (shorter: compiling verification) versus compiler implementation verification as has been taken over from the EU-funded project “Provably Correct Systems - ProCoS”, directed by D.Bjørner and C.A.R.Hoare, 1989/95 [3,50] and done in more extension in the DFG-funded project ”Verification of Compiler Specifications, Implementations and Generation Techniques (Verifying Compilers) – Verifix“, directed by G.Goos, F.W. von Henke and H.Langmaack, 1995/2003 [19,49,20,26,35,12,21]. The authors of [10] were probably the first who came up with differentiating between these two compiler verification areas.

Compiler constructors are allowed to work under two relieving assumptions [34]:

(i) Source level application programs are assumed to be correct w.r.t. their characterizing properties. Application programmers are responsible here, compiler constructors need only consider bare semantics of well-formed
source programs and not any further intentions of the application side (unless the source programs are systems programs used in compiler construction).

(ii) Hardware, i.e. target and host processors, are assumed to work correctly as described in their instruction manuals. Hardware engineers are responsible here, compiler constructors need only consider bare semantics of processor languages irrespectively of any errors in hardware construction. Further justifications of this assumption are: Hardware engineers have a much more rigorous culture in failures treatment than software engineers and hardware shows better resistancy against malevolent attacks [53].

Due to the hardware correctness assumption above (2.) every well-formed binary coded processor program is correctly executable. A higher level language program is correctly executable if there is a corresponding compiler (or interpreter) program correctly implemented as a well-formed binary coded processor program.

In spite of these two relieving assumptions there are virtually no executable industrial-commercial compilers available for which both, specifications and implementations, have been rigorously verified in full extension. See long lists of compiler bugs [21,5,44].

We have to accept that up to now rigorous compiler verifications have been performed mainly in academic institutions or in companies close to academic circles and not subjected to hard industrial competition. So we ought to find out what qualifications industrial software engineers and compiler constructors must possess who are able to do verification as well.

Compiler construction plus rigorous verification demand both a good systems software engineering qualification and a proper mastership in theoretical informatics and program semantics which in some sense are special areas of mathematics. It is by no means contradictory that an industrial software engineering area requires a great deal of theoretical considerations. Every good industrial practice is based upon theory, mathematical theory in our situation. L. Boltzmann pronounced that no idea is more practical than a good theory. A. Einstein even stated that only theory can say what is an appropriate experiment.

What can we learn from compiler construction and verification activities in order to explore which kinds and styles of mathematics are relevant? Firstly, we try to find out problems by differentiating between specification and implementation of compilers. Later for more decisive answers, we look at details in compiler verification projects [66,57,58,34,15,52]. A preliminary version of the present article appeared as [51].
2 Problems in adequate notions of correct implementation of program semantics and of programs

When we see the phrase “compiler verification” in literature then most authors only mean “compiling specification verification”. Furtheron, results on compiling specification verification must be handled with care, their assumptions are often unrealistic so that there are serious difficulties to subject the results to industrial requirements:

The difficulties stem e.g. from problematic choices of too idealistic target processors with infinite memories of cells where each one is capable to store any source language datum \[66\]. In theoretical informatics there is an unfortunate tendency to hide the rôle of representations and abstractions (retrievings) between abstract and concrete data and programs. For software engineers in industry (and academic informaticians as well) it is often quite hazardous to incorporate missing representations and abstractions in results on specification verifications.

The difficulties originate also from questionable choices of notions of correct implementation of a source program (semantics) by a target program (semantics). Namely different application areas and programmers have different view points and interests: A programmer of safety critical processes expects preservation of total program correctness as his notion of correct implementation \[8,38,56\]. This means in equivalent words: If a source program terminates successfully then so does the target program with equal resp. corresponding source and target results. The majority of programmers in “usual” information processing neither look so much for guaranteed successful program termination nor for total program correctness. “Usual” programmers are very satisfied with partial program correctness and its preservation \[6,47\]. This type of preservation can be formulated equivalently: If a target program terminates successfully then so does the source program with equal resp. corresponding source and target results.

A compiler constructor favours to consider himself as a “usual” programmer. Let his compiler, written in a specification or high level host language, be implemented on a host processor. Since this processor and also the target processor have only finite resources it is unreasonable to demand or expect that every well-formed source program can be successfully translated and every generated target program can be correctly implemented on the target processor. Nevertheless, compiler constructors must take in consideration further and different notions of implementation correctness which depend on the application fields of the source languages which are to be compiled to code of real life target processors.
Optimizing compilers (better: target code improving compilers) deserve a decent deliberation concerning errors in source and target code. Total program correctness does not guarantee any pleasant program property in case of at least one non-successful computation starting with inputs in the associated precondition domain. All errors as there are operation errors like division by zero, other execution undeniabilities, array bounds violations, infinite computations by loops or recursive procedures, number or memory overflows and other target processor error reports are considered to be unacceptable, chaotic, inexcusable, to be avoided by program users.

On the other hand, in case of partial program correctness, all errors are considered acceptable, excusable, unavoidable. In case of an error a program user simply tries a different computation, perhaps with a different input, and hopes for success. But the situation changes as soon as we optimize a compiler which preserves partial correctness. Errors come into play which a user must avoid by input data restrictions. A most prominent optimization in practice is elimination of array bounds tests in generated target code, a special dead code elimination. The new compiler does no longer preserve partial program correctness. Target code computations may terminate successfully whereas there is no corresponding source program computation which ends successfully with the same resp. a corresponding result. So the target processor’s computed result has nothing in common with the source program’s semantics.

If we want to avoid to be cheated by the optimizing compiler then we must declare array bounds violation in the source program semantics as an unacceptable error which the user must avoid by appropriate restriction to so called admissible inputs. So we have a generalized kind of program correctness, so called relative program correctness, and of implementation correctness in the sense of preservation of relative program correctness [59,60,76]. User manuals of optimizing compilers, especially commercial ones, must warn the user very clearly that he has to restrict his inputs to admissible ones. [62] harshly criticizes the actual state of the art in real life compiler manuals. An implemented executable optimizing compiler in general does not and cannot say which inputs may cause computations with unacceptable errors. The user himself has to avoid them. In future verifying compilers in the sense of C.A.R.Hoare [40] might be able to warn or even prevent the user to start translated programs with inadmissible inputs.

Let us formalize the ideas. The notion of correct implementation $\sqsubseteq$ deals with program semantics $f_s$, $f_t$, data representations $\rho_i, \rho_o$ and data (state) do-
mains $D_i^s$, $D_o^s$, $D_i^t$, $D_o^t$ in a commutative diagram

\[
\begin{align*}
D_i^s & \xrightarrow{f_i} D_o^s \\
\rho_i & \mapsto \rho_o \\
D_i^t & \xrightarrow{f_t} D_o^t
\end{align*}
\]

where $\mapsto$ or $\sqsubseteq$ indicate partially defined multivalued functions (relations). Due to [59,60,76] data domains $D$ include disjoint error sets

\[D = D^{\text{reg}} \cup A \cup U = D^{\text{reg}} \cup \Omega = D^{\text{acc}} \cup U.\]

We have regular data $D^{\text{reg}}$, acceptable errors $A$, unacceptable errors $U$ and acceptable data $D^{\text{acc}}$. Semantics and representations are required to be strongly error strict, i.e. total on $\Omega$, error strict and, beyond that, unacceptable error strict.

**Definition 2.1** (Correct implementation in the sense of commutativity or simulation) $f_t$ correctly implements $f_s$ ($f_s \sqsubseteq f_t$ for short) iff for all admissible inputs $d \in D^{\text{acc}}_i$

\[f_i(\rho_i(d) \cap D^{\text{acc}}_i) \subseteq \rho_o(f_s(d)) \cap D^{\text{acc}}_o\]

where admissibility of $d$ means $f_s(d) \subseteq D^{\text{acc}}_o$.

Nothing is said about inadmissible $d$ and how $f_t$ deals with inadmissible target input data. The user is warned furtheron to apply $f_t$ to target input data outside $\rho_i$-represented admissible inputs.

**Definition 2.2** (Relative program semantics correctness) $f$ is relatively correct w.r.t. pre- and postcondition $\Phi \subseteq D_i$, $\Psi \subseteq D_o$ ($<\Phi > f <\Psi >$ for short) iff

\[f(\Phi \cap D^{\text{acc}}_i) \subseteq \Psi \cap D^{\text{acc}}_o.\]

If all errors, especially divergence or infinite, non-terminating computation, are acceptable (or a little more general: if resulting unacceptable errors come only from unacceptable input errors) then we have partial program semantics correctness. If all errors and undefinednesses are considered to be unacceptable errors (or again a little more general: if resulting acceptable errors come only from acceptable input errors and all latter ones result in acceptable errors) then we have total correctness.
Theorem 2.3 (Correct implementation in the sense of preservation of relative program semantics correctness) \( f_t \) correctly implements \( f_s \) iff

\[
< \Phi > f_s < \Psi > \text{ implies } < \rho_i(\Phi) > f_t < \rho_o(\Psi) >
\]

for all \( \Phi \subseteq D^s_i, \Psi \subseteq D^t_o \).

It is obvious what correct implementation in the sense of preservation of partial resp. total program semantics correctness is. The following theorem is important for compilation in passes:

Theorem 2.4 Correct implementation is transitive horizontally and vertically, i.e. commutative diagrams can be composed in both directions.

Transferring the notions of relative correctness and correct implementation from program semantics \( f \) to well-formed programs \( \pi \) of a programming language \( L \) is simple: \( \pi \) is called relatively correct iff its uniquely associated semantics \( [\pi]_L \in Sem_L \) is so. And \( \pi_t \) correctly implements \( \pi_s \) iff \([\pi_t]_{TL}\) does so for \([\pi_s]_{SL}\), i.e. \([\pi_s]_{SL} \sqsubseteq [\pi_t]_{TL}\). The semantics space \( Sem_L \) is defined by

\[
\{ f : D_i \rightarrow D_o \}
\]

where \( D_i \) and \( D_o \) are the input and output data domains associated to \( L \). Semantics spaces like \( Sem_{SL}, Sem_{TL} \) may be considered as second order data domains, and an implementation relation \( Sem_{SL} \sqsubseteq Sem_{TL} \) (parameterized by \( \rho_i \) and \( \rho_o \)) as a corresponding program semantics.

Total program correctness implies partial one and often is much harder to prove. On the other side: Implementation correctnesses in the sense of preservation of total resp. partial program correctness are logically independent. Nevertheless, it is in a way intuitive that a proof of partial correctness preservation is harder than a proof of total correctness preservation. Are there mathematical justifications of this interesting observation? We find answers in [55,59,60,76]: The authors assume source languages to be furnished with denotational predicate transformer semantics, target languages with operational step semantics. The decisive difference is evoked by the premis whether divergence is considered to be an unacceptable error (total program semantics correctness) resp. to be an acceptable one (partial program semantics correctness). The inductive proof of total correctness preservation utilizes just a series of algebraic laws for assembly instructions semantics. But proof of preservation of partial correctness has, above that, to exploit a fixpoint characterization of the operational target semantics. The authors of [15] with their mechanical PVS-based proofs have the following answer to the question above:
For a proof of total correctness preservation rule induction suffices whereas partial correctness preservation requires a more intricate measure induction.

We should keep in mind that there is a twofold genesis of unacceptable errors and their associated inadmissible inputs of a well-formed source program which is correctly implemented and executed by a target program. In case of an inadmissible input there is either a target program computation which contradicts the source program semantics as expressed in Def. 2.1; or the source program semantics applied to the input does not satisfy the user’s intentions. So the reasons for unacceptable errors are weaknesses 1. of the implementation technique or 2. of the source program w. r. t. its application.

Admissible inputs $d$ of $f_s$ split also in two characteristic subdomains. The first one is that domain of $d$-s with

\[ f_t(\rho_i(d) \cap D_{regt_i}) \subseteq \rho_o(f_s(d)) \cap D_{regt_o} \]

where all target computations of regular inputs lead to regular results. For the other admissible inputs representation $\rho_o$, which is allowed to be multivalued, makes use of the programmer’s generosity that he is ready to accept so called acceptable target errors beside to accept regular results.

3 Demands of official software certification boards how to generate highest quality programs and software

In 1989/94 the German Bundesamt für Sicherheit in der Informationstechnik BSI published official IT-certification prescriptions for IT-safety and -security [77,78,9]². BSI introduced eight quality levels of trustworthiness from Q0 (a given program is unsufficiently checked) up to Q7 (a high level language version of a program is formally verified w.r.t. characterizing properties, the proof and the program’s implementation on a processor are performed by officially admitted tools). Officially admitted means: An official validation test suite is passed successfully. Because BSI felt that this kind of admission does not meet proved correctness [13] BSI added the following more rigorous code checking requirement for compiler tools:

“The transformations from source to target code executed by the admitted compiler program must be a-posteriori checkable (inspectable, in German: nachvollziehbar)” [17].

² There are similar activities in other countries.
It is amazing that a higher quality level Q8 (a program and its implemen-
tation on a processor are proved correct; i.o.w. a program is formally verified as
for Q7, the proof and the program’s implementation are performed by proved
correct and correctly implemented tools) is outside the scope of thinking of
an IT-certification board although renowned computer scientists have stressed
the eminent importance of program correctness (including prover and compiler
tool correctness) since a long time [73,18,61,39,13].

Nevertheless, it is BSI’s IT-certification prescription for BSI’s highest qual-
ity level Q7 which has suggested to our ProCoS- and Verifix-projects [50],
[19,49] the employment of checking or inspecting (by program or by hand) of
generated programs or codes in methods how to develop correct realistic compi-
lers, correctly implemented even down in binary real world host processor
code. Although BSI does not think of proved correct and correctly imple-
mented (only officially admitted) compilers one of our investigation results is

BSI’s prescription and recipe Q7 is applicable also to

compilers (which so is becoming a bootstrapping tech-
nique [75]) and suffices to create realistic Q8-compilers

The reason why BSI did not venture to demand a more rigorous tool ad-
mittance becomes aware when recipe Q7 is applied to verification (upper com-
mutativity) of a SL to TL-compiler τ₁ written in a high level host language
HL and to inspection (lower commutativity) of compiler τ₂ implemented as
an executable binary host machine program (code) HML:

```
Sem_{SL} \sqsubseteq Sem_{TL}
\\[\llbracket \_ \rrbracket_{SL} \uparrow \quad \uparrow \llbracket \_ \rrbracket_{TL}\\]
\quad SL \sqsubseteq TL
abstract well-formed programs

\varphi_{SL'}^{SL} \sqsubseteq \llbracket \varphi_{TL'}^{TL} \rrbracket
\quad SL' \quad [\tau_{HLL}]\quad TL'
concrete well-formed programs
and other HL-data

\varphi_{SL''}^{SL} \sqsubseteq \llbracket \varphi_{TL''}^{TL} \rrbracket
\quad SL'' \quad [\tau_{HML}]\quad TL''
concrete well-formed programs
and other HML-data
```
The $\varphi$-s are program representations and, in the lower diagram, identical to the representations $\rho$ of input and output data of programming languages like $HL$ and $HML$.

If BSI had required that the $HL$ to $HML$-bootstrapping compiler tool $\tau_0$ is not just tested but proved correct and correctly executable then, on the one hand, inspection of $\tau_2$ is unnecessary, but on the other hand, a further compiler (namely $\tau_0$) with the same high qualities is required as we are intending to prove them for the so called initial $\tau_2$. By BSI’s intellectual withdrawal to Q7 BSI avoids the logical difficulties of this circular regression.

But BSI’s recipe to create executable so called highest quality Q7-programs demands to do code checking for every translated program again and again. As that is nasty and time consuming [65] one might think to develop programs to perform the code checkings, but there is always remaining a certain quantum of code checking which must be done manually [68,49].

In the eyes of informatics science certification boards are obliged to introduce the higher software quality level Q8 because existence of Q8-compilers makes (low level) code checking unnecessary (outside compiler building) and because the Verifix-project has demonstrated that realistic proved correct and correctly implemented initial compilers can be constructed which translate high level systems programming (compiler writing) languages on real world processors [34,41].

4 Formalized differentiation between compiling specification and compiler implementation

BSI’s recipe Q7 is not easy to apply. This is because code inspection (lower commutativity, i.e. correct implementation of $\tau_2$) has no chance if the implementer has no knowledge of the $\tau_0$-constructor’s algorithmic compiling specification $C_0$ of abstract $HL$- to abstract $HML$-language features plus his representations of abstract source and target data and states by concrete ones. Bare knowledge of the natural compiling specification $\mathcal{C}_0$ [34] based only on $HL$’s and $HML$’s semantics is not helping the inspector. BSI’s so called official admittance of $\tau_0$ must include a readable algorithmic compiling specification which the specifier is obliged to care for a correctness proof. Otherwise inspection of $\tau_2$ is in vain.

We have an analogous requirement to verify compiler $\tau_1$. Whether verification is done by hand or by support of a mechanical prover the verifier has to know or to define an algorithmic compiling specification

$$C_1 : SL \rightarrow TL$$
and firstly to verify $C_1$ (i.e. $C_1$ correctly implements the implementation relation $Sem_{SL} \subseteq Sem_{TL}$, i.e.w. for all well-formed $\pi_s$ with

$$\pi_s C_1 \pi_t$$

$\pi_t$ is also well-formed with

$$[\pi_s]_{SL} \subseteq [\pi_t]_{TL}$$

and secondly to prove that $\tau_1$ implements $C_1$ correctly:

$$C_1 \subseteq [\tau_1]_{HL}$$

(i.e. if $\tau_1$ applied to an acceptable representation $\pi'_s \in \varphi_{SL'}(\pi_s) \subseteq SL'$ of an abstract well-formed $\pi_s$ terms with $\pi'_t$ then there is a

$$\pi_t \in C_1(\pi_s) \subseteq TL$$

with

$$\pi'_t \in \varphi_{TL'}(\pi_t) \subseteq TL'$$

and $\pi'_t$ is an acceptable $HL$-output datum. If $\pi'_t$ is not an acceptable error but a regular datum, then $\pi'_t$ is well-formed with

$$[[\pi'_t]_{TL'} = [[\pi_t]_{TL}$$

Most often a compiler constructor’s algorithmic compiling specification is a calculus of inductive conditional term rewriting or algebraic formula transformation rules which defines a source language $SL$ to target language $TL$ compiling relation $C_1$ (or $C_0$, a multivalued function), see [16,43,25], [15].

In order to economize the construction of a correct initial compiler N.Wirth recommended, already in the 1970s [75], to identify source and host language $SL = HL$ and target and host machine $TM = HM$, $TL = HML$ and the compiling specifications $C_1 = C_0$. So we understand the notion compiling specification to be definition of a (preferably algorithmic) $C_1$ with its verification of

---

3 The natural compiling specification $\mathcal{C}_1 = [[SL]; \subseteq; [TL]^{-1}$ is trivially correct.

4 So elements of programming languages like $SL$ or $SL'$ which are not well-formed (have no semantics) are considered as inadmissible input data of translations like $C_1$ or $[\tau_1]_{HL}$. These are thought to assign an unacceptable error in the output data domain to every inadmissible input datum.
The notion Compiler implementation naturally splits in construction of a compiler version $\tau_1$ written in high level language and of a version $\tau_2$ coded in low (machine) level language with associated verifications of

$$\text{Sem}_{SL} \subseteq \text{Sem}_{TL}$$

$$[\!\! SL \uparrow \& | \uparrow TL \!\!]$$

$$SL \triangleleft TL$$

The notion Compiler implementation naturally splits in construction of a compiler version $\tau_1$ written in high level language and of a version $\tau_2$ coded in low (machine) level language with associated verifications of

$$SL \triangleleft TL$$

$$\varphi_{SL} \uparrow \& | \uparrow TL$$

$$[\!\! \tau_1 \!\!]_{SL} \quad SL'$$

$$\varphi_{SL'} \uparrow \& | \uparrow TL'$$

$$[\!\! \tau_2 \!\!]_{TL} \quad TL''$$

$$\varphi_{SL''} \uparrow \& | \uparrow TL''$$

The notion of a correct compiler does not require that it is furnished with a complete syntax and static semantics checker (in spite of its practical importance), the programmer is allowed to offer only well-formed source programs to the compiler. If we have a verified compiler of a superlanguage $SL_0$ of $SL$ then we need not change the compiler code in order to get a proved correct $SL$-compiler, we need only a (mental) alteration of the code’s semantics by assigning a larger set of inadmissible source programs.

5 Realistic correct initial compiler implementation

Prime goal of this article is to figure out the level of mathematical reasoning which the compiler implementer, as opposed to the compiling specifier, defined at the end of chapter 4, is expected to master. Therefore we can assume that the compiling specifier has already properly done his work, all mathematical specification verification inclusive (specification correctness assumption). A clear operational and an equivalent denotational semantics of $SL$ help a lot to do rigorous specification verification. For better trustworthiness a mechanical theorem prover should be employed, e.g. Acl2, PVS, HOL or KIF [57,7,58,31,32,63,15,12,11,69], as much as possible with proof documents in the mathematician’s short style. The availability of a verified algebraic compiling specification $C_1$ makes high and low level compiler implementation verification a purely syntactic-algebraic reasoning activity; the implementer needs no longer look at semantical aspects of involved languages [34].
Formal methods projects have developed correct refinement or implementation rules for specifications and programs [45,23,2,64], [42,70,1,14]. But we rarely see complete well documented derivations of realistic compiler programs written down in binary real life host processor code and correctly executable by this processor or one of its family.

5.1 Realistic correct high level initial compiler implementation

The compiler writing and source language $SL$ should be in close neighbourhood to the chosen specification language. The latter is determined by the mechanical theorem prover in case such a prover is employed. Especially, $SL$ should have good programming facilities which allow to write down $SL$-program pieces which at execution time manipulate source and target program terms as internal and external data of $SL$. Ideally, abstract program terms of the specification language should be internal data of the source language as well. The $SL$-facilities should allow immediate evidence to the reader that only such program terms are resulting which the specification’s conditional term rewriting rules permit [27,14]. In a way, [57,58] uses $SL$ itself as the specification language, namely Boyer-Moore- or Acl2-Lisp. Consequence: If a computation of the well-formed $SL$-written compiler program $\tau_1$ starts with a well-formed source program $\pi_s$ and finishes with a regular result then this is a well-formed target program $\pi_t$ which can be generated by the specification calculus as well; i.e. $\tau_1$ is correctly implemented.

We demonstrate construction of a piece of compiler $\tau_1$ due to an associated $C_1$-rule from project Verifix where $SL$ has been defined to be ComLisp [24], a sublanguage of ANSI-Common Lisp [71], for several good reasons. A ComLisp-program is essentially a list of non-nested function declarations with call by value (strict) parameter transmissions. ComLisp has just one data type: s(symbolic)-expressions which are abstract binary trees with a standard character string representation. Abstract resp. concrete ComLisp-programs are special abstract resp. concrete s-expressions.

ComLisp performs dynamic typing which a compiler can handle more easily than static typing and type inference as in Standard ML [54]. This fact has excluded a sublanguage of ML as a candidate for $SL$. Especially, every undefined application of a ComLisp-operator leads to signalling of an acceptable error at every implementation level. This enables the desired preservation of partial program correctness.

Also C [46] is no candidate as a superlanguage of $SL$. C is not only too low level (and nevertheless very distant from binary processor code), C’s semantics of arithmetical operators depends on the arithmetic of those processors on which translated C-programs are executed. This hinders reliable portability
of and reasoning about C-programs.

$C_1$’s term rewriting rule for ComLisp’s standard operator and user defined function calls is [25,15]:

$$CL_{\text{form}}[(p\ f_1\ \ldots\ f_n)]_{\rho\gamma k} = CL_{\text{form}}[f_1]_{\rho\gamma k}$$

$$\vdots$$

$$CL_{\text{form}}[f_n]_{\rho\gamma k+n-1}$$

$$(p\ k)$$

Local and global variable environments $\rho$ and $\gamma$ and addresses $k$ relative to stack frames are parameters of calculus operators.

The compiler implementer implements this rule as a part of a function declaration in $\tau_1$ [27]:

(defun CLform (form lenv genv k)
  (cond
    ((consp form)
     (let* ((key (car form))
               (args (cdr form))
               (n (list-length args)))
       (cond
         ;; treatment of form as
         ;; ComLisp-keyword applications
         ;
         (T ;; treatment of form as a standard operator
          ;; or user defined function application
          (append
           (CLforms args lenv genv k)
           (list (list key k)))))))
    ;; treatment of form as
    ;; literals or variables
    ;
    )))

5.2 Realistic correct low level initial compiler implementation

Two problems arise:
(i) Where or how do we get hold of an executable compiler \(\tau_0\) which corresponds to BSI’s so called officially admitted compiler, i.e. which with a very high probability for just one computation (one compiling) from input \(\tau_1\) to output \(\tau_2\) adheres perfectly to the rules of compiling specification \(C_1 = C_0\) ?

(ii) How can we arrange an industrially feasible code inspection of \(\tau_2\) ?

Ad 1.: We should define \(SL\) as a syntactical and semantical sublanguage of an industrially used superlanguage \(SL_0\) which is implemented by an executable, industrial strength compiler \(\tau_{00}\) on a real life processor \(M_0\), e.g. Intel-Pentium.

\[
\begin{array}{ccc}
SL & \tau_1 & TL \\
\hline
SL & \tau_1 & TL | SL & \tau_0 & TL \\
| SL & \tau_0 & ML_0 | ML_0 & | ML_0
\end{array}
\]

Due to our understanding of the notion “compiler” \(\tau_{00}\) is an \(SL\)-compiler as well. Since \(SL\)-program \(\tau_1\) is well-formed execution of \(\tau_{00}\) will, in case of successful termination, end up with a result \(\tau_0\) which is, most probably, a correctly executable implementation of \(\tau_1\). Execution of \(\tau_0\) will, in case of successful termination, end up with a result \(\tau_2\) which is, most probably, also a correctly executable implementation of \(\tau_1\). As soon as we have done rigorous code inspection of the program pair \((\tau_1, \tau_2)\) with respect to the verified compiling specification rules \(C_1 = C_0\) we are definitely sure that \(\tau_2\) is correct as well. Should by accident \(\tau_{00}\) not have worked correctly and intruded an error in \(\tau_2\) rigorous code inspection of \((\tau_1, \tau_2)\) uncovers the error, see discussion about Trojan horses later. Code inspection of \((\tau_1, \tau_0)\) is not feasible if there is no algorithmic compiling specification \(C_{00}\) known for \(\tau_{00}\).

Ad 2.: Even syntactic algebraic checking might become unmanageable if the rules are too expansive and intertwined and the source/target documents are very long. Next section 5.3 offers a realistic method.

5.3 Multi pass compiling, good-natured compiling specification rules and diagonal checking method

Three modes of acting are central for realistic, practical manageability of syntactic-algebraic code inspection and so for low level compiler implementation verification:

\[\text{We use McKeeman’s T-diagrams as a shorthand for situations which should be more precisely depicted by commutative diagrams.}\]
(i) **Multi pass compiling:** In order to bring up rules which cause only modest expansions we split translation $C_1$ from $SL = \text{ComLisp}$ to $ML = \text{Transputer code } TC_0$ in five passes:

$$
\begin{align*}
\text{ComLisp} & \xrightarrow{CL} \text{SIL} \xrightarrow{CS} C^{int} \xrightarrow{CC} \text{TASM} \xrightarrow{CA} TC_1 \xrightarrow{CT} TC_0
\end{align*}
$$

Project Verifix has chosen the Transputer T400 with 1 MByte of main memory as target and host processor $M$. This choice was due to good experience in project ProCoS, T400 was available as a stand alone processor.

(a) SIL is a stack intermediate language with s-expressions as data, operational stack semantics and variable names replaced by small relative addresses. The underlying stack machine (also for the following $C^{int}$) has similarities to N.Wirth’s P(Pascal)-machine [74].

(b) $C^{int}$ is C-like, similar to Java’s virtual machine language JVL and to IL in [62] ([67] uses C itself as intermediate language), with two potentially infinite linear integer arrays for stack and heap.

(c) TASM is assembly language, here oriented towards Transputer, without symbolic addresses and with subroutines enumerated sequentially and called via sequence indices.

(d) $TC_1$ is external Transputer language in binary or hexadecimal notation of bytes (resp. words) of subroutines and of initial stack and heap. A small, 253 bytes long boot program is a representative of $CT$ and loads a well-formed $TC_1$-program as a well-formed $TC_0$-program. The latter simulates the former in a 1:1 manner if the former is compiled from a well-formed ComLisp program.

ComLisp, SIL and $C^{int}$ are machine independent. All intermediate languages have s-expression syntaxes as ComLisp has. It is worth mentioning that realistic practice (where rigorous verification is no primary aim) does compiling with similar intermediate languages for manageability reasons.

(ii) **Good-natured compiling specification rules:** We consider translation rules as subcases of inductive definitions of possibly multivalued, partially defined functions on s-expressions. A most important characteristic of good-natured rules is: Juxtaposition of source and result s-expressions allows the reader to recognize every derivation (rewriting) step and every associated location of rule application [43]. Even the very popular arithmetic violates this characteristic. Look at the rules of natural numbers

---

6 Code generators for i386, DECa, MC68000, C and Forth are available as well, but so far they depend on unverified software like assemblers or auxiliary compilers and lack verification.
addition and multiplication:

\[ \mathcal{A}[(o\ n)] = n \]
\[ \mathcal{A}[(+ 1\ n)\ m)] = (+ 1\ \mathcal{A}[(n\ m)]] \]
\[ \mathcal{M}[(o\ n)] = o \]
\[ \mathcal{M}[(+ 1\ n)\ m)] = \mathcal{A}[(m\ \mathcal{M}[(n\ m)]] \]

Pure addition \( \mathcal{A} \) is good-natured, but multiplication \( \mathcal{M} \) is not. E.g. in order to check result \( N \) of \( \mathcal{M}[(N\ M)] \) the checker in general has to do a whole new derivation and see whether its result is really equal to \( N \) [16]. Such a phenomenon makes manual result checking unfeasible.

On the other hand, W. Goerigk and U. Hoffmann [43,25,26] found out that all translation rules for \( C_1, CL, CS, CC, CA \) allow result checking without repeated derivations of result s-expressions. The authors figured out the following good-naturedness properties:

(a) No left or right hand nestings of calculus operator applications like \( CL_{form} [\cdots] \) in section 5.1.
(b) Compositionality, i.e. left hand is a calculus operator application, there are no multiple left hand variable occurrences, and right hand variables occur also on the left hand.
(c) No multiple right hand variable occurrences.
(d) Order of left hand variables occurrences is preserved on the right hand. Variable \( p \) and its disorder is not relevant because \( p \) stands for ComLisp-operators and -function names only and they are atomic (!) s-expressions.

These properties (a) to (d) have to take in consideration a possible condition which an application of an s-expression rewriting rule is subjected. Only those variables for s-expressions are relevant for feasible checkability which stand for possibly non-atomic (!) s-expressions. It is a massive help that all languages mentioned above have a subroutine concept and subroutines are translated to subroutines.

The second multiplication rule above violates (a),(c) and (d). Optimizing compilations make use of rules which do not satisfy the properties of good-naturedness. So constructors of initial compilers should abstain from optimizations and wait for later bootstrappings when low level compiler implementation verification is no longer necessary.

The compiling specification rules for all passes can be read in [25], [15], the rules’ implementations in \( SL = \) ComLisp in [27], [14] and their principal handling during code inspection in [43], [26,34].
(iii) **Diagonal checking method:** To abstain from one pass compilation and to split $C_1$ in passes (see 1.) does not only lead to less expanding compiling specification rules. There is a further big advantage when the passes are refined down to TC$_1$-code and checked (horizontal and vertical transitivity of implementation correctness): All checkings below the diagonal in the rectangular implementation diagram (see below) are redundant as a matter of fact. E.g. let pass $CA$ be implemented and checked down to TC$_1$, reading and printing routines included, and let pass $CC$ be done so only down to TASM. Then we can generate a correct TC$_1$-implementation $\tau_{53}$ of $CC$ by bootstrapping on the Transputer due to the hardware correctness assumption, chapter 1, 2.. Etc. $\tau_{52}$ for $CS$ and $\tau_{51}$ for $CL$. Most interesting: There is a very close connection between this checking redundancy and N.Wirth’s strong bootstrap test [48,49].

Furtheron: The printing routines can be improved [52] so that it is factually sufficing to implement the reading routines correctly in high level ComLisp $^7$ (and not down in binary TC$_1$) and to use the printing routines as correctly implemented result checkers. The remaining manual work is just to check given and printed s-expressions for equality.

In order to get a glimpse of feasibility of Verifix’s syntactic-algebraic checking method we show in Fig. 1 a complete code inspection documentation for a short ComLisp-function declaration example $f(x\ y)$ [34]:

Although checking is purely syntactical and the inspector needs no semantical insight, in principle, a few semantical remarks might help to understand the translations. The ComLisp-function body has two operator calls whose rewriting rule we have considered in section 5.1. $0, 1, 2, 3$ are local addresses in SIL (frame length 4), they correspond to $0, 2, 4, 6$ in C$^{int}$, TASM and TC$_1$ (frame length $8$). $51 = 33_{hex}$, $30 = 1e_{hex}$, $28 = 1c_{hex}$ are jump table indices of $f$, $*$ and $+$; $74 = 4a_{hex}$ bytes is $f$’s coded body length in TC$_1$.

```
(DEFUN F
  (* 0)
  (_COPYC 6 1)
  (+ 0))
```

would be a semantically correct compilation result of function declaration $f(x\ y)$ also, but is not as specified in $CL$ and so would be rejected by Verifix’s syntactical inspection.

---

$^7$ A reading routine is just a ComLisp-implementation of lexical-context-free analysis and transduction algorithms proved correct in formal language theory.
Beside several simple typewriting errors the mechanical prover PVS [15] has found one essential error which occurs at four places in [27]'s proposed algorithmical compiling specification. One occurrence is in the entry code \texttt{ENTCD} of operator and function bodies in TC\textsubscript{1}, namely the Transputer in-

Fig. 1. A Code Inspection Documentation

\begin{verbatim}
(defun f(x y) (+ (* x (_COPY 0 2) (_SETL (_L 0) 4) (_SETL (_L 1) 5)) y (_COPY 1 3) (_SETL (_L 2) 6) (_SETL (_L 3) 7)) (_,COPY 2 0)) (_SETL (_L 4) 0) (_SETL (_L 5) 1))
\end{verbatim}

\begin{verbatim}
TASM
(_DEFCODE F 51 (33 z 4a
ENTCD 8
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 GCALL 44 70 21 3e f6
LDC 3 LDL 3 STNL 6 43 73 e6
LDC 6 LDL 3 STNL 7 46 73 e7
LDC 4 LDL 0 LDNL 28 GCALL 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
EXTCD ) (75 60 5e d5 75 31 d3 75 30 f6 ))
\end{verbatim}

\begin{verbatim}
\end{verbatim}
The diagonal is indicated by boldface mapping symbols like \( \llbracket \text{id}^{\text{ext}} \rrbracket \) and language denotations like \( \text{SIL}' \). The compiling specifications are proved correct (diagram commutativities of the first row) in [15] and the high level compiler implementations (second row) in [27,14]. The complete code inspection protocol (a mathematical proof documentation for low level compiler implementation) is in [28,43], only the upper triangle is necessary. \( \tau_{31}, \tau_{41}, \tau_{42}, \tau_{51}, \tau_{52}, \tau_{53} \) need not be inspected, we are especially freed from most lowest level code inspections. Every occurrence of a primed language \( L' \) is input (resp. output) data domain \( D \) of a host programming language \( HL \). The regular data \( D^{\text{reg}} \) are (sequences of) character strings which are either representations of well-formed \( L \)-programs or representations of other (sequences of) \( s \)-expressions or further (sequences of) character strings. \( \text{id}^{\text{ext}} \) is the \( L \)-respective extended identity mapping

\[
\text{id}^{\text{ext}} = \text{id} \cup D^{\text{reg}} \times A \in D \rightarrow D,
\]

which expresses the compiler user’s readiness to accept every acceptable error report beside a regular (successfull) compilation result. The developed ComLisp-compilers \( \tau_1 \) and \( \tau_2 \) preserve partial program correctness if we re-
strict the compilation input data domains to well-formed source programs; the sequential composition of \( \tau_{11} \) to \( \tau_{14} \) is the compiler \( \tau_1 \), the sequential composition \( \tau_{51} \) to \( \tau_{54} \) is the finally desired correctly executable compiler \( \tau_2 \).

There is a trend for systems software to be required open source, enabling source code scrutiny for operating systems components, compilers and other tools and utilities. This will unveil a lot of bugs, but the open source idea depends crucially on trusted compilation. Therefore the authors of [72], [30,33,29] demonstrate how a hacker can corrupt the executable compiler \( \tau_{00} \) in section 5.2 so intelligently that the generated compiler \( \tau_2 \) passes Wirth’s strong bootstrap test successfully and is nevertheless an incorrect implementation of the verified (!) \( \tau_1 \), i.e. \( \tau_2 \) has got a Trojan horse from \( \tau_{00} \). \( \tau_2 \) translates just two SL-programs incorrectly, one application program \( \pi_1 \) and the compiler \( \tau_1 \).

BSI’s official compiler test suites applied to \( \tau_2 \) have no real chance to find the hacker’s specially chosen \( \pi_1 \) with its translated \( \pi_2 \) which reveals catastrophic results deviating from those of \( \pi_1 \). BSI has a trivial chance to find \( \tau_1 \), but BSI does no rigorous check of correctness of the translation result, which is \( \tau_2 \) due to the successfully passed bootstrap test. Here is BSI’s mental gap in BSI’s prescription and recipe which forces compiler users to do low level code inspection again and again. The neglected rigorous low level code inspection of \( (\tau_1, \tau_2) \) would have revealed that there is an error. The revelation comes up even in the upper implementation diagram triangle due to the successful bootstrap test.

Translation validation projects as in [67] or in [62] or also in Verifix [37,36,21] have a lot in common with Verifix’s initial compiler subproject. The interesting aspect is that the deviations in compiler research directions are not conflicting but complementary.

Every translation validator’s program checker for an existing compiler pass is to be verified and correctly programmed in a language, preferably the same high level host language (C is high level in this respect) of the compiler pass. So we get from one original problem to create a correctly executable compiler pass to a new problem to find a correctly executable compiler (pass) for a new language, the host language. This is a circular hen-egg problem which Verifix’s initial compiler is interrupting by doing (some, as few as possible) manual program checkings or program inspections [68,49].
6 Conclusion

Can we answer the question in our article’s title? We think so, at least in some respects. We have not looked at every possible today and future task of a software implementer. But we have looked at a very difficult and characteristic problem where there is an intensive and duely investigated cooperation between software specifier and software implementer in mathematical modeling, formalizing and rigorous proving: Realistic construction of a so called initial correct compiler for a realistic high level programming (and compiler writing) language correctly implemented on a real life host processor. Realistic construction means: As much as possible avoidance of programming in and inspection of lower level code. Unverified and unverified implemented auxiliary software, compilers and program checkers are allowed to be employed, but their employments must be done so carefully that the correctness of the finally implemented and executed initial compiler does not depend on any possible incorrectnesses of auxiliary software. Any further high level language compilers can be executably implemented by correct bootstrapping without any further programming in and inspection of lower level code.

What kind and level of mathematical reasoning can computer science expect from the software, especially the compiler implementer?

(i) He must understand calculi of conditional program term rewriting rules, which are just algebraic-syntactic formula transformation rules, and be able to do derivations (algebraic transformations).

(ii) He must be able to develop program pieces in a compiler writing language which at computation time result only in program terms which right hand sides of rules permit. In a functional language of Lisp- or ML-kind such developing is a very straight forward algebraic activity.

(iii) He must do rigorous syntactic a-posteriori code inspection of corresponding source and target code pieces which a possibly not quite correct compiler has generated. This inspection is a purely algebraic derivation activity due to term rewriting rules which the compiling specifier has proved correct. So the implementer needs not care for any semantics issues. In case the rules are good-natured as for the initial ComLisp to Transputer-compiler all derivation steps are recognizable inside juxtaposed source and target code. No extra derivation steps need be done.

Let us look at the software specifier, especially the compiling specifier. The kind of mathematics he has to master is much deeper than performing algebraic constructions, transformation and comparisons governed by algebraic rewriting rules. The specifier must be a versed expert in different forms of
denotational and operational semantics of programming languages and processors. Correctness of compiling specifications requires very subtle rule and measure induction proofs, an area of mathematical reasoning which many software engineers are no friends of.

The time needed for proofs by help of a mechanical theorem prover is in general not shorter than by hand. The prover can successfully prove decisive theorems only if the user knows strategies how to do proofs by hand. The proof effort with PVS for all passes of the initial ComLisp to Transputer-compiling specification took about 36 person months including the development of widely reusable verification techniques, of course. If proof would not be mechanized but just written down manually in a technical report, we believe that it is unlikely that it would even be touched again, e.g. when changes in compiling specifications, languages or target machine force for adjustments and reinvestigations [12].

Construction of the initial ComLisp to Transputer-compiler and implementation verification including experimental work and design of source and intermediate languages took about three years of intensive work, not counting the mechanical proof. The low level syntactical a-posteriori code inspection covers approx. 1000 pages of code inspection protocols (The initial compiler consists of 237 ComLisp-functions, many are short and therefore many protocol pages are not totally filled up with text). A complete check takes less than three person months of concentrated work, and additional trusted machine support might further increase confidence. The code inspection work load is significant, but well in the range of typical certification efforts [28,12].

Sometimes there are coming up critics that three months concentrated work of an inspector might be too long and boaring so that errors may slip through. But the software implementer who works here as an inspector should not forget that his work is absolutely necessary mathematical proof work and generated inspection protocols are proof documents which uncover every failure (also Trojan horses) which unverified auxiliary software (compilers) might have intruded. Project Verifix’s proceeding demonstrates that compiling specification verification takes much longer concentrated proof work, liberates the implementer from semantical issues and reduces the implementer’s mathematical reasoning to a moderate algebraic reasoning.

Acknowledgement

The author would like to thank his colleagues J. Knoop and W. Zimmermann for their kind invitation to give this lecture at the COCV 2005 workshop and A. Dold, T. Gaul, Sabine Glesner, W. Goerigk, G. Goos, A. Heberle, F.W. von
Henke, K. Höppner, U. Hoffmann, M. Müller-Olm, V. Vialard, A. Wolf and W. Zimmermann for their fruitful cooperation in the project Verifix. Heartily thanks to Annemarie Langmaack for typesetting of this article.

References


[44] Java Compiler Bug Database from Sun, approx. 90 bugs


[53] H. Maurer, Globalisierung und der Zusammenbruch der Computernetze. Informatik Spektrum 28(1), 24-33, 2005


