Compiler Correctness and Implementation Verification: The *Verifix* Approach

Wolfgang Goerigk* Axel Dold§ Thilo Gaul‡ Gerhard Goos‡
Andreas Heberle‡ Friedrich W. von Henke§ Ulrich Hoffmann* Hans Langmaack*
Holger Pfeifer§ Harald Ruess§ Wolf Zimmermann‡

Abstract

Compiler correctness is crucial to the software engineering of safety critical software. It depends on both the correctness of the compiling specification and the correctness of the compiler implementation. We will discuss compiler correctness for practically relevant source languages and target machines in order to find an adequate correctness notion for the compiling specification, i.e. for the mapping from source to target programs with respect to their standard semantics, which allows for proving both specification and implementation correctness. We will sketch our approach of proving the correctness of the compiler implementation as a binary machine program, using a special technique of bootstrapping and double checking the results. We will discuss mechanical proof support for both compiling verification and compiler implementation verification in order to make them feasible parts of the software engineering of correct compilers. *Verifix* is a joint project on *Correct Compilers* funded by the Deutsche Forschungsgemeinschaft (DFG).

Keywords: compiling verification, compiler implementation verification, computer based systems (CBS), correct compilers, safety critical software.

1 Introduction

In most cases tests are not sufficient to guarantee program correctness. It becomes more and more apparent that program verification is needed to meet high reliability requirements for safety critical software. Adequate software engineering methods, especially formal methods, should be used to support the correct program construction. Program verification methods are best known to work for mathematically clean high level programming languages. However, it is the binary machine code running on a concrete piece of hardware which we ultimately want and have to trust. Therefore, program correctness crucially depends on the correctness of the compiler used for implementation. The compiler, more precisely its machine code implementation, should be proved to carry over application program correctness from source to target programs, i.e. to preserve application program correctness.

Compiling correctness for sequential, imperative source languages is easier to formulate than the general problem of program correctness. We know the mathematical theory of source and target language semantics and the theory of partial and total program correctness in sequential, non-real-numerical application areas, and hence we are able to exactly define the relationship a compiler has to establish between source and target programs. A mathematically exact specification of the problem is available.

The definition of the compiling specification, mapping or relating source to target programs and the proof of its correctness with respect to semantics (*compiling verification*) are crucial tasks while constructing correct compilers. In section 2 we will develop a framework which allows to express and to study different compiler correctness notions. Practical verification techniques for realistic compilers and compiler implementations are needed. Even source languages are often defined operationally, based on an abstract execution model. In addition to the standard technique of structural induction on source programs also simulation or bisimulation proofs can be used to prove code generator specifications correct.

Like in ordinary program verification, however, it is again the binary machine code executable which we ultimately want to trust. In addition to compiling verification we have to prove the specification to be correctly implemented on the machine (*com-*

*Christian-Albrechts-University of Kiel, Preußerstr. 1-9, D-24105 Kiel, (wg@informatik.uni-kiel.d400.de)
‡University of Karlsruhe, Vincenz-Prießnitz-Str. 3, D-76128 Karlsruhe, (zimmer@ipd.info.uni-karlsruhe.de)
§University of Ulm, James-Franck-Ring, D-89069 Ulm, (ruess@informatik.uni-ulm.de)

piler implementation verification). In section 3 we will sketch our approach to the proof of compiler implementation correctness. We use a specialized bootstrapping technique and double-check the resulting code to be correctly generated as specified. A reasonable choice of intermediate program representations separates crucial compilation steps from each other. The implementation correctness proof is modularized into smaller parts. Compiling verification uses more or less the same intermediate steps anyway.

In section 4 we will discuss mechanical proof support for both compiling and compiler implementation verification. Since compiler verification usually produces a lot of tedious proof obligations, we need to incorporate mechanical support into our proof methods to make them practical. We must not depend on unverified theorem prover implementations, but of course proofs get much more trustworthy if they are additionally checked by machine, even if the automatism is not fully verified. We are also allowed to use theorem provers like PVS to find proofs, if the proof protocols at the end are completely under human control.

The *Verifix* project tackles techniques for the software engineering of correct compilers. Three research groups at Karlsruhe (G. Goos), Ulm (F.W. v.Henke) and Kiel (H. Langmaack) work together on compiling verification, compiler implementation verification and compiler generation verification for realistic sequential imperative source languages on real machines.

2 Compiling Correctness

As there are different notions of program correctness, the question arises whether we are able to define one single reasonable, comprehensive notion of compiler correctness. Of course, a compiler should transform source programs to semantically equivalent target code. However, this correctness notion is too strong for realistic compilers generating real machine code; machines have finite resource limitations. We will develop a framework which allows to express and to study advantages, drawbacks and relationships between different compiler correctness notions. The usual commutative diagram relating source and target program semantics to the compiling function (or relation) will be specialized for the case of state based sequential imperative languages.

This section is best characterized as a comprehensive presentation of preliminary work, results and ideas originated from the ProCoS [2] [1] work on compiling verification [8] [9] [17], especially from [11] and [14]. However, we will argue why for the special purpose of correct compiler construction a compiler correctness notion based on partial program correctness is sufficient.

Our definitions also work out if both source and target language are defined operationally on the basis of an abstract (or concrete) machine. In this case we can enrich the repertoire of proof techniques and use simulation or bisimulation proofs well known from computability theory in order to prove compiling correctness.

2.1 Correct Compilers

Compiler verification has a very long history starting in the mid-60's with the work of McCarthy and Painter [12]. A good overview of the research work performed on compiler verification and the results achieved since then is given by Jeffrey J. Joyce in [10]. McCarthy and Painter have established a standard mathematical paradigm which has been adopted by most of the literature on compiler correctness: abstract syntax of both source and target language, abstract mathematical definition of the compiling specification, abstract definition of an idealized target machine code; even the basic proof method, structural induction on the syntax of source programs, is common to most of the work presented so far.

For source languages SL and target languages TL a compiling specification C will be defined either mapping or relating source programs $p \in SL$ to target programs $C(p) \in TL$. Usually, the correctness of the compiling specification C is expressed by the commutativity of a diagram similar to the one shown in figure 1 below.

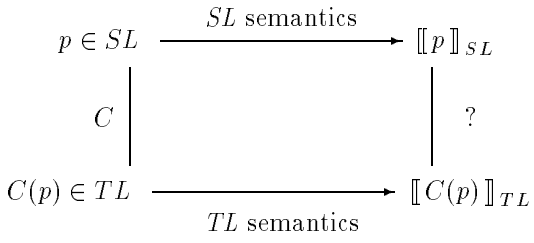

Figure 1: Compiling Correctness

But how to relate source and target program semantics to each other? For imperative languages we can

take a closer look (cf. figure 2) to the right hand side of the diagram above. In this case, programs mean state transformations. The source program semantics $[\![p]\!]_{SL}$ is a function mapping program states (mappings from program variables to values) $q \in Q_{SL}$ to program states. Denotational semantics for instance define programs to mean (continuous) mappings $[\![p]\!]_{SL} \in Q_{SL} \rightarrow Q_{SL}$. More recently, also operational (F. and H. Nielson), structural operational (G. Plotkin) and axiomatic approaches (C.A.R. Hoare, [8]) have been used in the area of compiling verification.

Machine program state transformation $[\![C(p)]\!]_{TL}$ can be defined very naturally starting from the operational definition of the machine configuration transformation of single instructions as defined in the machine manual.

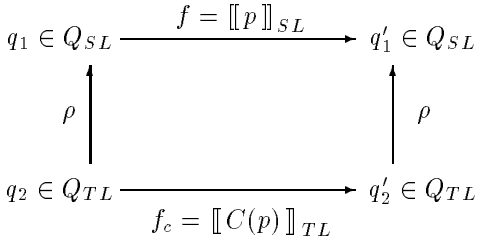

Figure 2: Compiling Correctness for Imperative Languages

A *retrieve* function ρ (*abstraction, or inverse representation*) relates both state spaces to each other. It abstracts program states from machine states, i.e. values from their concrete machine representations.

Different notions of weak or strong commutativity of this diagram now give rise to different notions of compiling correctness (cf. section 2.2). If we require

$$\rho \circ f_c \;=\; f \circ \rho\,,$$

the compiling specification C has to establish strong semantical equivalence of source and target program. It will preserve both partial and total correctness properties of the application source program (cf. section 2.2 below). Markus Müller-Olm [14] shows strong commutativity for a while-language P and a machine language MP of conditional jumps and assignment instructions, assuming ρ to be the identity map on $Q_P = Q_{MP}$.

Because of the finiteness of real machines, however, realistic correct compiler construction has to weaken the envisaged correctness notion. In general strong commutativity will be very hard to achieve. Regular program termination – successful termination without any runtime error occuring – of the target program cannot be guaranteed for arbitrary well-defined source programs. One solution could be to strengthen the source language semantics to reflect machine restrictions. This method, however, causes many additional problems in program verification and only pays off for instance in the field of safety critical embedded control, where irregular program termination could be disastrous, where total program correctness is required and has to be preserved. On the other hand, weakening the target code semantics to completely abstract from machine resource limitations at the end is undesirable either. Although very important for modularizing correctness proofs, and successfully adopted in the wide majority of the literature on compiler verification so far, complete abstraction from machine resource limitations rules out concrete hardware to be the target of correct program implementations.

If we have a closer look to different error situations in source and target programs, we can identify those runtime errors which are due to resource violations of the concrete finite target machine, e.g. memory or arithmetic overflow. The best we can achieve without considering additional knowledge about machine and implementation details is that C

- establishes semantical equivalence, but
- $C(p)$ may abort due to machine resource violation even for well-defined programs p.

For most concrete target machines M we are able to define an idealized version M^∞ with unrestricted resources. In this case we could additionally require $C(p)$ to be semantically equivalent to p on M^∞. This excludes unfair compilers like those constantly generating non terminating memory exhausting code or a division by zero. In most practical cases this also implies that for every well defined program p and every input there exists a (finite) target machine which is "large enough" for $C(p)$ to terminate regularly on that input.

2.2 Weaker Correctness Notions

Different fields of applications require different notions of program correctness. In many cases, partial correctness suffices. One typical example is the compiler itself: If it terminates regularly then the resulting code has to be the correct one. We need a rigorous mathematical proof for this fact. However, compiler verification would become much more

cumbersome or even impossible if we would insist upon regular termination of the implemented compiler for every source program. Although of course this property is a desirable and very important software quality aspect for compilers, it is not central for compiler correctness. We subsume this kind of total correctness of a compiler under quality aspects where a different level how to gain confidence could be satisfactory, e.g. compiler validation as already used in industry nowadays.

Figure 2 allows for the definition of different notions of compiler correctness. f_c is defined to be a correct implementation of f based on weak or strong commutativity of the diagram:

(a) **Preserving Partial Correctness:** $\rho \circ f_c \subseteq f \circ \rho$. Regular termination of the target program implies correct results w.r.t. source program semantics. This is the inverse of

(b) **Preserving Total Correctness:** $f \circ \rho \subseteq \rho \circ f_c$. For well-defined source programs correct target program results are guaranteed.

(c) **Preserving both Partial and Total Correctness:** $\rho \circ f_c = f \circ \rho$, i.e. both (a) and (b) hold. This notion of strong commutativity corresponds to (strong) semantical equivalence, sometimes called *bisimulation*.

(d) **Weak Commutativity:** Even weaker than (a) or (b), we require for every $q \in Q_{SL}$ $(\rho \circ f_c)(q) = (f \circ \rho)(q)$ only if both sides are well defined.

Although total program correctness is stronger than partial program correctness, it turns out that preserving total correctness does not imply preserving partial correctness nor vice versa.

The weak commutativity as defined in (d) actually is too weak. We cannot conclude the correctness of the result of a regularly terminating machine program execution without additionally proving the well-definedness of the corresponding source program with respect to the same input data. And, moreover, we also cannot conclude the regular termination of the target program from proving the well-definedness of the source program. The latter makes up the essential difference of (d) to the notion defined in (b). Since (b) in addition to (d) preserves regular termination behavior of the source program, (b) is very useful in the area of e.g. safety critical embedded control or reactive systems where irregular program termination or non-termination could be disastrous because of the absence of a safe error state. Hoare [8] and Sampaio [17] for instance use (b), i.e. the preservation of total program correctness. The programmer has to prove regular termination, which then is guaranteed to be preserved. In many cases, the compiler may produce more efficient target code, e.g. it may omit range or type checks. However, since the target program is allowed to terminate regularly producing incorrect results in cases where the source program is undefined, a user's proof of total program correctness is required if a correct compiler in the sense of (b) shall be used.

Since regular termination of the source program implies regular termination of the target program, (b) actually preserves total program correctness: Let f be totally correct with respect to P, Q and let us start f_c in state $s \in \rho^{-1}(P)$. Then $\rho(s) \in P$, $f(\rho(s))$ is defined and so is $\rho(f_c(s))$ with the same result. Since Q holds for $f(\rho(s))$ also $\rho^{-1}(Q)$ holds for $f_c(s)$.

The correctness notion in the sense of (a) allows target programs to be less defined than source programs, for instance because of limitations of machine resources. A target program may irregularly abort, even if the source program is well-defined. We consider that not harmful; above all, the target program is not allowed to deceive the user about the quality of results. If a regular result is given, it is guaranteed to be the correct one (or one of the correct ones in the case of nondeterminism or non-injectivity of ρ). This correctness notion is adequate if partial correctness of programs suffices. It does not stress the user to give cumbersome proofs of regular termination of application programs, if not required.

Since regular termination of the target program implies regular termination of the source program with the same result, partial program correctness is preserved (if ρ is total): Let f be partially correct with respect to P and Q and let $f_c(s_c)$ be defined such that $\rho^{-1}(P)$ holds for s_c. Then $\rho(f_c(s_c))$ is defined and equal to $f(\rho(s_c))$. Hence Q holds for $f(\rho(s_c))$ and $\rho^{-1}(Q)$ holds for $f_c(s_c)$.

2.3 Partial Correctness of Compilers

Let us now consider the compiler itself: The property that it preserves (partial and/or total) program correctness, is a partial correctness property of the compiler: If the source program is well formed, and if the compiler manages to generate a target program, then the target program shall be as correct as the source program. The same partial correctness property should hold for the compiler implementation, of course. Thus, preserving partial correctness (a) is sufficient for the compiler used to implement the compiler itself, even if the implemented compiler

shall be correct in one of the other senses. A correct compiler in the sense of (b) or (d) would not help unless we additionally prove regular termination properties for the compiler program. A lot of unnecessary proof work eventually would lead away from the central work necessary for establishing both compiling and compiler implementation correctness for a first compiler.

Moreover, in our initial approach to compiler implementation verification (cf. section 3), the compiler implementation will be bootstrapped with the compiler itself. Therefore it is convenient and sufficient to use (a) as the correctness notion for the compiling specification.

Preserving partial program correctness is not the ultimate compiler correctness notion. Our major goal is to make our techniques and methods practically usable for compiling and compiler implementation verification projects based on stronger or different compiler correctness notions. Compilers should be correct in a stronger sense, e.g. as defined at the end of section 2.1. Note, however, that for the construction of a first full correct compiler implementation we only need a rigorous mathematical proof of partial correctness preservation.

3 Implementation Verification

In order to prove full compiler correctness as rigorously as required to assure the correctness of the complete development process for safety critical software, we have to carefully verify both the compiling specification (cf. section 2) and the compiler implementation. After refining the compiling specification into a program formulated in high level compiler implementation language, the compiler program itself has to be transformed into a binary machine program. An implementation correctness proof is necessary.

This fact has first been severely stressed by J Moore [13]. Unfortunately, the literature on compiler verification gives no sufficient solution so far. No fully reliable realistic compiler implementation is available, since this agenda has not been completely worked out for any existing compiler or programming language implementation. Compilers and, hence, executed high level programs are not enough trustworthy nowadays. Instead, the correctness or reliability of safety critical software is approached by more or less complete semantical binary code inspection, partly using unverified de-compilation tools.

3.1 Bootstrapping

We have chosen source and implementation language to be an appropriate subset ComLisp of CommonLisp in order to achieve a first proved correct compiler implementation. Moreover, we will implement the compiler on its own target machine. A *specialized bootstrapping technique* can be used in order to generate the machine code implementation of the compiler using an unverified CommonLisp system. We rigorously double check the result to be correctly generated. The result has been generated according to our own proved correct compiling specification and hence we fully know what it should look like.

Figure 3 sketches our bootstrapping and double checking technique in the simplified case of only one intermediate language, e.g. C. Compiling verification and specification refinement yield verified compiler programs from ComLisp to C and from C to the machine language ML, written in ComLisp. The compiler from ComLisp to ML in ML will be generated in five steps:

1. We use an unverified CommonLisp system to run the compiler from ComLisp to C, compiling the compiler from C to ML to C. The result is obviously not fully verified (indicated by hatching the diagrams in figure 3). It depends on the unverified CommonLisp system. We double-check the result by hand in a mathematical style to be generated as specified. The mathematical correctness of this special test result then no longer depends on the CommonLisp-system.

2. The correct implementation of the compiler from C to ML in ML proceeds exactly the same. It is the initial verified compiler machine program on hardware (fat lined diagrams represent proved correct compiler programs).

3. At the front end, we get the verified compiler program from ComLisp to C written in C analogously.

4. We assume hardware to work correctly. Therefore we now can correctly bootstrap the compiler from ComLisp to C in ML on the machine. No further manual proof work is necessary.

5. The sequential composition of the two machine programs yields the desired proved correct compiler implementation (if compiling specification and its refinement to ComLisp have been proved correct beforehand).

If we generalize the setting to incorporate even more, say n, intermediate languages or program representations, figure 3 above will generalize to a

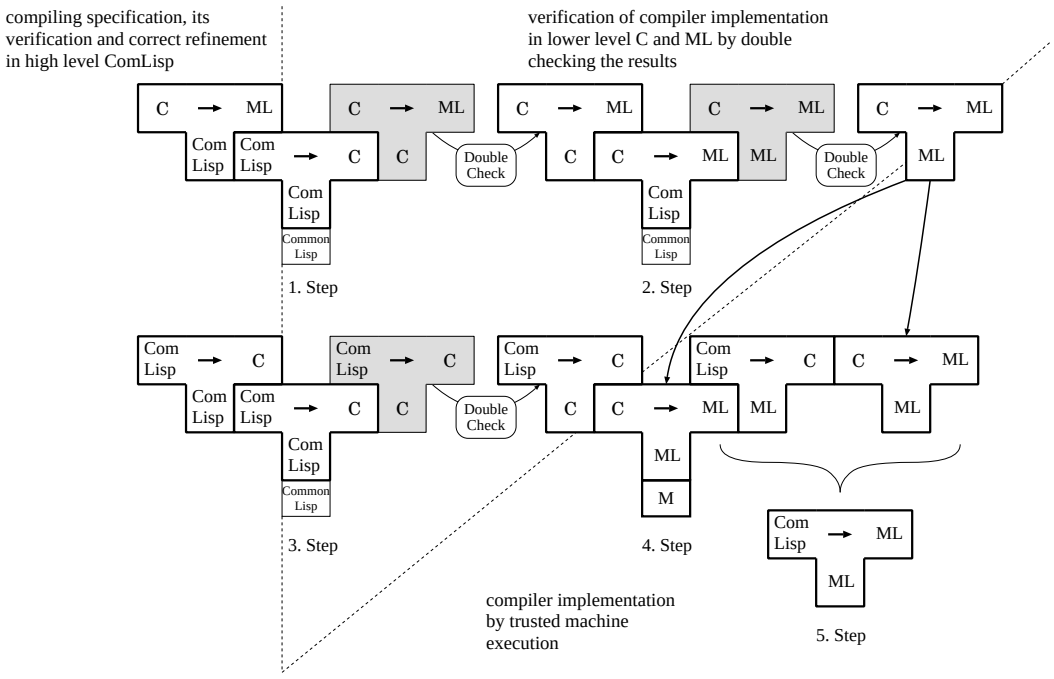

Figure 3: Implementation Verification Using Bootstrapping and Double-Checking the Results

$(n+1) \times (n+2)$ matrix of different compiler parts written down in different representations. It will immediately become clear, that manual double checking the results is necessary only in the upper left triangle of that diagram, whereas proved correct machine generation will generate the remaining part. Note that the compilers used for machine code generation are now both correctly specified and proved to be correctly implemented.

A reasonable choice of intermediate representations will modularize the manual double checks into smaller parts, and, even more important, it will separate crucial compilation steps from each other which also makes every single double checking proof step easier. Since it is especially annoying to double check lower level code and especially binary machine code, reasonable strategy should be to do small translation steps towards the end of compiling.

In a first attempt we have successfully run the bootstrap using three intermediate languages: The first one is an abstract high level stack intermediate language; functions (and procedures) with parameters are compiled to parameterless procedures. The second one is an abstract small subset of a strongly typed imperative language like C used as an abstract machine language. This language essentially is the target of data representation for dynamic Lisp data. Then, in order to flatten control structure into linear code with jumps, we use an assembler target language. For the last step, of course, an assembler program is constructed to finally generate binary machine code.

Our proceeding is an interesting application of a method of J. B. Goodenough and S. Gerhart [6], who proposed to prove a program property P as the consequence of finite testing results together with a substitute property P'.

Once having completely worked off this programme, including the compiling verification for the different compilation steps, the proved correct compiler for the basic implementation language ComLisp easily can be used in order to correctly bootstrap new or improved (e.g. optimizing) compiler implementations even for different hardware platforms. ComLisp can also be used to implement compiler construction tools like code generator or parser generators. No further implementation verification is needed.

4 Theorem Prover Support

In order to manage the large complexity of compiler correctness proof work, mechanical proof support is absolutely necessary. We use the specification and verification system PVS [16]. The higher-order specification language with a rich typing system, the set of tools for creating, analyzing, modifying and documenting theories and proofs, and the powerful inter-

active Gentzen-style theorem prover adequately support formalization and verification. PVS provides a set of elementary proof steps which can be combined into efficient proof strategies enabling more readable proofs, closer to those performed by hand. We will give some examples how PVS can adequately support compiling and compiler implementation verification:

If both source and target language are defined operationally (e.g. by means of evolving algebras) and formalized as abstract machines in the PVS language, proof strategies can carry out simulation proofs nearly automatically. They incorporate efficient rewritings, decision procedures and propositional simplifications by means of binary decision diagrams (BDDs). The verification process is divided into several refinement steps. Each step proves the correspondence between two abstract machine state traces w.r.t. a retrieve function ρ (cf. section 2.1). Usually only some *visible* machine states correspond to abstract (source) states, i.e. the machines run at different rates.

In the ProCoS approach the source language is embedded in a refinement algebra. The semantics of the target language is expressed by an interpreter written in the source language. Refinement laws are applied to show that the interpreted target code is a correct refinement (implementation) of the source code. We have developed PVS proof strategies which almost enable to carry out these proofs as done by hand, hiding several tedious PVS proof steps.

PVS can also provide support for reasoning about machine programs. Based on a formal operational machine model, we can prove properties like correctness assertions for the Transputer boot protocol, using symbolic execution techniques provided in proof strategies in a similar way as described above.

The goal of language and machine formalizations in PVS is to abstract from concrete languages and machines, to factor out common aspects and to identify language or architecture specific parts. This is directly supported by parameterized PVS theories. Parameters can be constrained by means of assumptions, theories can be instantiated by concrete machines; a proof is required that the assumptions are satisfied. This method also facilitates the augmentation of extra features of source and target languages. The generic parts of specifications and proofs can be reused.

To illustrate the method of generic specifications, we have developed an abstract scheme for verifying local optimizations on object code, and have proved correct a set of optimizations for different architectures using defined proof strategies [5].

The long term goal is to construct a library of reusable generic PVS theories for the development of correct compilers.

Related Work

In the so far largest project on the formal verification of compiling processes at Computational Logic Inc. (CLInc, Austin, Texas) the Boyer-Moore-prover is used to construct and verify a stack of components (CLInc stack) covering the compilation of the high level imperative language Micro Gypsy down to the hardware processor FM8502. This imperative language is first compiled to assembler code [18] and further to machine code [13]. Compiler and assembler are specified and verified with respect to source and target language semantics. In [13] J S. Moore formulated the necessity of also proving the implementation correct. However, even in the CLInc project this gap has not been closed so far. The Boyer-Moore-prover, both used to verify and to execute the compiler, has not completely been verified so far, neither as Lisp program nor in its binary form executed on the machine.

More recently, papers like [3] [4] or the VLisp project reports [15] [7] also express the necessity of proving the compiler implementation correct. In the VLisp project, however, this work has explicitly been left out, and Paul Curzon [4] argues that the direct theorem prover based execution of the compiling specification should be satisfactory to convince the user of implementation correctness. The implementation gets even more trustworthy, if, as an additional test, it has been generated independently using a different execution method like bootstrapping, and the two result are equal. Although of eminent practical use, however, in our opinion this argument leaves mathematical correctness of the implementation open. The correctness depends on the correctness of the theorem prover and its implementation.

Acknowledgements

We thank our colleagues in the ProCoS project, in particular Martin Fränzle, Burghard v. Karger and Markus Müller-Olm. Without the background of the ProCoS project work we would not be able to express both differences and relationships between different compiler correctness notions in the current form. We also thank J S. Moore of Computational Logic Inc. He gave the original motivation to more deeply think about the proof work necessary for a rigorous compiler implementation correctness proof.

References

[1] Dines Bjørner, Hans Langmaack, and C.A.R. Hoare, editors. *Provably Correct Systems – ProCoS, ESPRIT BRA 3104.* Dep. of Computer Science, Techn. Univ. of Denmark, 1993. Monograph, Final Deliverable.

[2] Dines Bjørner. *Final Report ProCoS - Provably Correct Systems, ESPRIT BRA 3104.* Dep. of Computer Science, Techn. Univ. of Denmark, 1991.

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

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

[5] Axel Dold, F.W. von Henke, H. Pfeifer, and H. Rueß. A Generic Specification for Verifying Peephole Optimizations. Ulmer Informatik-Berichte 95-14, Universität Ulm, 1995.

[6] J.B. Goodenough and S.L. Gerhart. Toward a Theory of Test Data Selection. *SIGPLAN Notices*, 10(6):493–510, June 1975.

[7] J. D. Guttman, L. G. Monk, J. D. Ramsdell, W. M. Farmer, and V. Swarup. A Guide to VLisp, A Verified Programming Language Implementation. Technical Report M92B091, The MITRE Corporation, Bedford, MA, September 1992.

[8] C. A. R. Hoare. Refinement algebra proves correctness of compiling specifications. In C.C. Morgan and J.C.P. Woodcock, editors, *3rd Refinement Workshop*, pages 33–48. Springer-Verlag, 1991.

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

[10] Jeffrey J. Joyce. Totally Verified Systems: Linking Verified Software to Verified Hardware. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *Lecture Notes in Computer Science*, 1990.

[11] Burghard v. Karger. Algebraic Compiler Verification. Internal report, Oxford University Computing Laboratory, October 1993.

[12] 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.

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

[14] Markus Müller-Olm. An Exercise in Compiler Verification. Internal report, CS Department, University of Kiel, 1995.

[15] Dino P. Oliva and Mitchell Wand. A Verified Compiler for Pure PreScheme. Technical Report NU-CCS-92-5, Northeastern University College of Computer Science, Northeastern University, February 1992.

[16] 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.

[17] Augusto 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.

[18] W.D. Young. A verified code generator for a subset of gypsy. Technical Report 33, Comp. Logic. Inc., Austin, Texas, 1988.

Verifix