On Certifying Code Generation

Jan Olaf Blech

blech@informatik.uni-kl.de

Technical Report
No. 366/07

November 2007

Computer Science Department
University of Kaiserslautern

Abstract. Guaranteeing correctness of compilation is a major precondition for correct software. Code generation can be one of the most error-prone tasks in a compiler. One way to achieve trusted compilation is certifying compilation. A certifying compiler generates for each run a proof that it has performed the compilation run correctly. The proof is checked in a separate theorem prover. If the theorem prover is content with the proof, one can be sure that the compiler produced correct code. This paper presents a certifying code generation phase for a compiler translating an intermediate language into assembler code. The time spent for checking the proofs is the bottleneck of certifying compilation. We exhibit an improved framework for certifying compilation and considerable advances to overcome this bottleneck. We compare our implementation featuring the Coq theorem prover to an older implementation. Our current implementation is feasible for medium to large sized programs.

1 Introduction

Today's software systems are developed using high-level model or programming languages, even in safety critical embedded systems. Since their runtime behavior is controlled by the compiled code the need for trusted compilation is more pressing than ever. Results achieved from static analyses and formal methods on the source code level have often to be considered worthless if the formalization chain from high-level formal methods to the machine-code level is not closed.

Two general approaches can be distinguished to bridge this gap. Thus establishing compilation correctness[1]. *Certified compilers* prove in a first step that the algorithms of the compiler define a correct translation for all given well-formed input programs (compiler algorithm correctness) and second that the algorithms are correctly implemented on a given machine (compiler implementation correctness). *Certifying compilers* (cp. Figure 1) provide a proof (called *certificate*) that a target program is a correct translation of a source program whenever such a translation is performed. It is important to notice that these proofs do not make a statement about a compiler algorithm or its implementation, but only about the relation of two programs. Compared to compiler certification, the technique

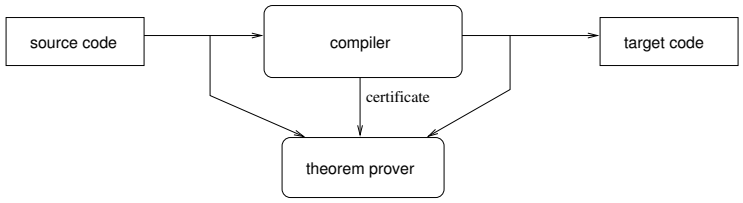

Fig. 1. Certifying Compiler

of compilers certifying their results has three main advantages. First, the issue of implementation correctness can be completely avoided. We do not have to trust the implementation of the compiler algorithms on a hardware system or prove it correct (cp. [4, 18, 6] on this problem). Second, similar to the proof carrying code approach ([13, 12, 1]), the technique provides a clear interface between compiler producer and user. In the certified compiler approach compiler users need access to the compiler correctness proof to assure themselves of the correctness. Thus, the compiler producer has to reveal the internal details of the compiler whereas the translation certificates can be independent of compiler implementation details. Furthermore this abstraction from implementation details frees us from reverifying the compiler once an aspect of implementation changes slightly. The disadvantages of the certifying compiler approach is that users have to check the certificates for each (critical) compilation. For larger programs this may be very

[1] We follow the notions given by Xavier Leroy in [10].

time consuming. Both the certifying and certified compiler methodologies can be applied independently to different phases of a compiler.

In this paper, we present a certifying compiler back-end translating an intermediate language into MIPS [16] code. Our original certifying compiler framework is described in [4, 6]. Based on this framework our certifying compiler back-end comprises the following features

- Machine-checkability and independence of logic: All certificates generated are machine-checkable using a theorem prover based on a formal general logic. This logic is independent of languages and techniques used in the translation. In this paper we use the Coq theorem prover [20] to specify our notion of compilation correctness and as a checker for certificates.
- Semantics of involved languages and their correspondence: We require an explicit formally specified semantics of intermediate language and MIPS code and an explicit criterion stating correctness of compilation.
- Certifying compiler: We are using a technique where a special well separated part of the compiler generates proof scripts as checkable certificates.
- User proved facts: The user of our compiler may provide facts he has proved on source code level. For example the user may provide the information that a variable used as an array index never exceeds the bounds of the array. This enables us to abandon bound checks when accessing a memory location the array is mapped to. This is an optional feature if the user does not want to provide facts (maybe because he does not trust his source code analysis) he does not have to.

This work ports and improves the certification framework introduced in [4] to the Coq theorem prover. Compared to the old implementation we have encountered a great reduction in the speed for conducting the correctness proofs and are now able to present a certifying compiler back-end that is able to handle realistically sized programs. Furthermore we have extended the involved languages to make them even more realistic and simplified the architecture of our certificate generation resulting in a clear separation between actual code generation and certificate generation.

Overview of the Paper

We discuss related work in Section 2. The intermediate language, the generated MIPS machine code as well as the compilation process is described in Section 3. Intermediate language and MIPS code are related with a notion of semantical correspondence in Section 4. We describe the process of proving correctness of a compiler run, its automation, and implementation using Coq in Section 5. In Section 6 we evaluate our work and a conclusion is drawn in Section 7.

2 Related Work

Apart from our own work [4, 18, 6] on certifying compilers the following approaches are most relevant to this paper.

In the translation validation approach [17, 22, 23] the compiler is regarded as a black box with atmost minor instrumentation. For each compiler run, source and target program are passed to a separate checking unit comprising an analyzer generating proofs. These proofs are checked with a proof checker. A translation validation approach and implementation for the GNU C compiler is described in [14]. Like in translation validation we regard correctness for each single compiler run. The analyzer generating the proofs corresponds to our certificate generator. In contrast to translation validation our approach is based on a general higher-order proof assistant as proof checker and explicitly formalized semantics. Furtheron we use more information to generate the proof scripts from the compiler.

Credible compilation [19] is an approach for certifying compilers. Credible compilation largely uses instrumentation of the compiler to generate proof scripts. Like translation validation and in contrast to our work credible compilation is not based on a explicitly formalized semantics.

Proof carrying code [13] is a framework for guaranteeing that certain requirements or properties of a compiled program are met, e.g. type safety or the absence of stack overflows. While these are necessary conditions that have to be fulfilled in a correctly compiled program we require in our work a comprehensive notion of compilation correctness. In [11], Necula and Lee described a certifying compiler for their approach guaranteeing that target programs are type and memory safe. The clear separation between the compilation infrastructure and the checkable ceritificate is realized in our approach as well.

A large body of research has been done on certified compilers. Here, we can only give an overview of the different areas of work. In [10], the algorithms for a sophisticated multi-phase compiler back-end are proved correct within the Coq theorem prover. To achieve a trusted implementation of the algorithm, it is exported directly from the theorem prover to program code. A similar approach based on Isabelle/HOL is presented in [8]. The verification of an optimization algorithm is described in [2]; it uses an explicit simulation proof scheme for showing semantical equivalence. In an important step in the direction of automating the generation of correct program translation procedures is explained in [9]. A specification language is described for writing program transformations and their soundness properties. The properties are verified by an automatic theorem prover. Techniques and formalisms for compiler result checkers, decomposition of compilers, notions of semantical equivalence of source and target program as well as stack properties were developed in the Verifix project [7, 5, 21].

3 Intermediate Language and MIPS code

In this section we sketch syntax and semantics of our intermediate and MIPS language. Both intermediate and MIPS semantics are defined in a small-step operational way. Hence definitions of syntax are done using abstract datatypes. States are encoded as tuples and transition rules as state transition functions.

3.1 The Intermediate Language

An excerpt of the definition of the intermediate language's syntax is depicted in Figure 2. The language comprises arithmetic expressions, (array-)variable assignments, (un)conditional branches, a print statement for output, and (potentially recursive) procedure call and return statements. Procedures are lists of statements. Programs consist of one or more procedures. Intermediate language statements may comprise operands appearing on the left (*loperand*) or right side of an assignment. Such operands comprise local (with respect to a procedure) as well as global variables. Variables are identified with integers (Z). N denotes natural numbers.

```
Inductive operand : Type :=
  | CONST : Z -> operand | VAR : Z -> operand | LOCVAR : Z -> operand
  | ARRAYC : (Z * Z) -> operand | ARRAYV : (Z * Z) -> operand.

Inductive loperand : Type :=
  | LVAR : Z -> loperand | LLOCVAR : Z -> loperand
  | LARRAYC : (Z * Z) -> loperand | LARRAYV : (Z * Z) -> loperand.

Inductive ilstatement : Type :=
  | ILPLUS : (loperand * operand * operand) -> ilstatement
  | ILBRANCH1 : (operand  * N) -> ilstatement
  | ILPRINT : operand -> ilstatement
  | ILCALL2 : (loperand * N * operand * operand) -> ilstatement
  | ILRET1 : operand -> ilstatement
  ...
```

Fig. 2. Intermediate Language Syntax (excerpt)

The Coq definition of a state in the intermediate language is show in Figure 3. It consists of five components: a flag of termination indicating whether the current procedure has terminated, called another procedure or encountered an error state. Furthermore the output occurred so far during the execution of the program, a mapping from global variables (including arrays) to values, a stack for local variables (including call arguments) and program counters as well as a program counter indicating the next statement to be executed are encoded. The semantics is defined via a state transition function *ilnext* taking one state and an intermediate language procedure mapping them to the succeeding state.

3.2 The MIPS Language

Our formalized set of MIPS instructions comprises basic arithmetic operations, shift operations, branch instructions. In addition instructions for basic output,

```
Record ilstate : Set := mkilstate
{termstate : N; output : list Z; varvals : (Z * Z) − > Z;
 locvarsstack : list ((Z − > Z) * N); pc : N }.
```

Fig. 3. Intermediate Language State

procedure calls and return from a procedure are provided. It should be noted that some formalized instructions such as instructions for procedure calls are not genuine MIPS instructions. They consist of several real instructions but are handled as one atomic instruction throughout this paper for simplicity reasons. They encapsulate a predefined sequence of MIPS instructions doing work such as storing call arguments in predefined spaces on the stack. As in the intermediate language code for procedures is stored as lists of instructions. The definition of

```
Record ilstate : Set := mkilstate
{tltermstate : N; tloutput : list Z; regs : Z − > Z;
 mem : Z − > Z; tlpc : N }.
```

Fig. 4. MIPS Code State Definition

a MIPS machine's state is shown in Figure 4. As in the intermediate language it consists of a flag indicating termination or other special occurrences and a list of so far accumulated output. Instead of variable to value mappings it consists of registers and memory to value mappings. A program counter is part of the MIPS state, too.

The state transition function encapsulating the semantics is called *tlnext*. Our semantics also needs a state transition function executing several instructions at a time taking a state, a procedure definition, and the number of states to be executed: *tlnextn*.

3.3 The Code Generation Algorithm

Our code generation phase comprises four steps. Apart from generating code some analysis information for generating the correctness proofs are emitted. In a first step memory locations are determined for local and global variables. Memory locations for local variables are assigned relatively to a special fixed register serving as stack pointer. In the second step register allocation is performed. Some values may be kept at some program points in registers. Nevertheless our current implementation requires that there is still one memory location for each variable. One result of these two steps is a mapping from intermediate language variables to registers and memory addresses (*variable mapping*).

In the next step the intermediate language program is processed sequentially and for each statement one or more MIPS instructions are generated. In our current implementation this generation is done via standard compiler textbook algorithms. Hence some simple optimizations are applied to each instruction code sequence representing an intermediate language statement. Apart from the generated code a byproduct of this phase is a relation of intermediate language and MIPS code program points that correspond to each other: the *program counter relation*.

In a last pass through the MIPS program jump targets are resolved with the help of this *program counter relation*. Both the *variable mapping* and the *program counter relation* serve as hints for our certificate generation. The whole compiler is implemented using the ML programming language.

We have introduced an intermediate language and our formalization of the MIPS processor instructions in this section as well as the principal code generation. Integers are formalized in Coq using a possibly non limited bit-wise representation. This can be limited to 32 or 64 bits depending on the actual MIPS processor the code is compiled for. For verification purposes integer arithmetics is required to be the same in intermediate language and MIPS language in our current implementation. Strings are not explicitly handled in our intermediate language and MIPS code. It is however possible to encode strings as integer arrays.

The involved intermediate language was chosen for its closeness to source code resulting in sequential processing of statements and good readability (see e.g. [3] for approaches to defining and reasoning about semantics of a more sophisticated intermediate language). The MIPS processor was chosen because of its simple architecture, wide area of usage, and the availability of a simulator. Further language features such as intricate arithmetic operations can be added easily into our compiler. However the focus of this paper is on demonstrating the applicability of the certifying compiler approach particularly solving the time problems arising with checking the certificates.

4 Correctness of Compilation: Semantic Correspondence

To verify that a transformation has been conducted correctly one needs to formalize a notion of correctness. The original and transformed program shall semantical correspond to each other.

For our compiler we regard two programs as semantical corresponding if they produce the same output traces. For the conduction of correctness proofs however, it is much more useful to use a more restricted criterion that implies the equality of observable traces.

In this work we break the task of verifying the compilation of a complete program down to the verification of its procedures. Hence we regard the correctness of independently compiled procedures. To guarantee semantical correspondence of output traces we require the compiled procedures to generate the same output traces. Furthermore the target code procedure may only write to the memory

heap (global variables in the intermediate language) or to its own stack frame (local variables in the intermediate language). Parameters during procedure calls have to be passed at distinct locations on the stack as are return values from procedure calls. We require each procedure invoked within a procedure to be correctly compiled according to these criteria. Global variables of different procedures from the same program have to be mapped to the same memory locations. The main procedure is treated like any other procedure in our methodology. With these requirements on compilation of procedures we guarantee correctness for the compilation of a complete program.

Formally we require both intermediate language and MIPS program to be in a (weak) simulation:

- The initial states have to have corresponding values for variables and memory locations.
- For two corresponding intermediate and MIPS states, if there is a next intermediate operation, there has to be one or more MIPS instructions and the execution of these operations has to denote the same output, and calculate the same corresponding values i.e. the succeeding states are in the simulation relation again. During the execution of such a step no violation of stack or other properties may occur.

```
Lemma simulation:
 user_provided_facts (optional) ->
   statecomp s0_il s0_tl Vars MemMap PCRel  ∧
   forall s_il s_tl,
      statecomp
        s_il s_tl  Vars MemMap PCRel
         ->
      statecomp
        (ilnext s_il ilprog)
        (tlnextn (steplength s_tl PCRel) s_tl tlprog )
        Vars MemMap PCRel.
```

Fig. 5. Simulation Criterion

Figure 5 shows our *simulation criterion* comprising the requirements for correctness of procedure compilation formalized in Coq (slightly simplified). As mentioned in the introduction when proving it correct optionally facts provided by the user of the certifying compiler may be used. One can see the requirements on the initial states *s0_il* and *s0_tl* as formalized in the second line of the Lemma as well as the simulation step quantifying over all possible state *s_il* and

s_tl in intermediate language and MIPS code. The *statecomp* predicate encapsulates the requirements on states as defined by the simulation relation. It is parametrized with a set of variables (*Vars*) whose values shall correspond to the values stored at certain memory (or register) locations on the MIPS machine, the *variable mapping* encoded using a function *MemMap* (cp.3.3), and *PCRel* the formalization of the *program counter relation*. Note that the correctness of our certificate checks does not depend on these compiler provided information. If wrong parameters are provided to *statecomp* the overall proof check will not succeed since derivation of output equivalence will not be possible!

Discussion
The methodology presented in this paper allows for a verification that transforms an intermediate language operation into one or more MIPS instructions. For our code generation phase such a $(1 : n)$ relation is sufficient and simplifies the prove process. However in other compiler phases other criteria have to be used (cp. [6]). In this paper we do not regard stack overflows, but simple assume that they do not occur. It is possible to abandon this general assumption and provide facts to the verification process proved on source code level. These might be stating e.g. that only a certain stack depth occurs during the execution of a program. This procurement is similar to dealing with array index bounds verification.

The question on when to regard programs as correctly transformed lacks a simple answer (cp. Section 2) . Different notions may be adequate for different purposes e.g. a failure of a target program due to resource limitations might be an acceptable behaviour for some software aimed at running on a large range of different computers. It is however unacceptable for most cases in embedded systems. With (weak) simulation allowing us to encapsulate the requirements of the simulation relation within a predicate like *statecomp* we believe that our general approach is flexible enough to be adapted to all criteria commonly used for correctness of compilation.

5 Proving Correctness of Compilation

In this section we describe our methodology to prove a compilation run correct. We sketch a general correctness proof first. Secondly we emphasize on the certificates our compiler generates. Moreover we describe the pieces of software that have to be written for certificate generation.

5.1 Proof Sketch

To prove a code generation run correct we have to show that each intermediate language procedure and its compiled MIPS counterpart fulfill the *simulation criterion* presented in Figure 5.

First we prove that the initial states of both programs are in the simulation relation fulfill the *statecomp* predicate, respectively. For showing that for each two states fulfilling the *statecomp* predicate the succeeding states are in the relation again we make a case distinction on the intermediate languages program

counter. To fulfill *statecomp* it must point to some intermediate language statement. Furthermore the MIPS program counter has to point to a corresponding MIPS program point and the *program counter relation* has to indicate the exact number of corresponding MIPS instructions. We make a case distinction on all possible intermediate language statements. Hence we split intermediate language and MIPS code into corresponding slices which have to semantical correspond to each other. For each corresponding pair of slices we prove in Coq a separate lemma that they compute equivalent values, store them at equivalent locations, reach equivalent program points, call equivalent procedures with equivalent parameters, return equivalent values or produce equivalent outputs. Of course a typical MIPS program may compute a lot of intermediate values that do not appear in the intermediate language. We handle this by requiring only values of variables appearing in the intermediate language procedure and the appropriate memory locations to correspond to each other.

To prove such a single step correct we require a number of prerequisites. Various properties concerning the mapping from variables to memory have to be ensured in a *first phase*.

The step lemmata realizing the case distinction on the intermediate languages program points are done in a *second phase*. Finally it is all put together in a *third phase* proving the *simulation criterion* (cp. Figure 5).

This case distinction on program points of the given programs is the key to proving the equivalence of intermediate language program and MIPS program. It should be noted that proving such a step correct is not a direct execution of certain instructions in certain states since the variables/registers/memory values in such states are not fixed. It is the deduction of an abstract successor state from another abstract state with the rules defining the semantics as introduced in Section 3. Hence this procurement lifts the dynamic nature of trace based semantics to a static view enhancing the possibility to reason about possibly infinite state systems in a theorem prover.

5.2 Generating and Proving Certificates

Figure 6 shows our certifying code generation infrastructure. The actual code generation takes an intermediate language procedure and produces MIPS code. Furthermore as pointed out in Section 3.3 a set of variables used in the intermediate language, a *variable mapping* mapping variables to memory locations and a *program counter relation* is emitted. These are subsumed to *info* in the Figure. It should be noted that when performing complicated optimizations in a compiler phase it is very helpful to emit optimization relevant information such as analysis results among the other *info* items. Coq representations of intermediate language and MIPS code are created for the compiled procedure. Based on these information the certificate generator generates the proof scripts proving the semantical correspondance between intermediate language and MIPS code. Finally the theorem prover is invoked to process the proof scripts. Thus conducting the correctness of compilation. Facts proved on source code level may

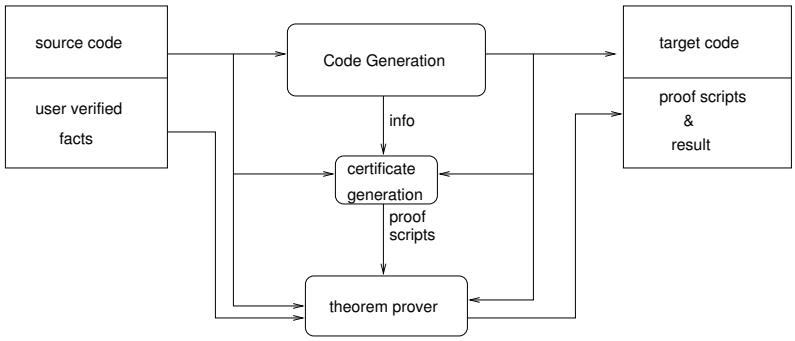

Fig. 6. Overview of Our Certifying Code Generation

be used for this process. As with proof carrying code one might imagine scenarios in which it is advantageous to keep the proof script so that other people using the program can be convinced that they have indeed a correctly compiled procedure with respect to a piece of source code.

The certificate generator emits several proof scripts that depend on each other. As described in Section 5.2 the processing of these scripts by the theorem prover is structured in three phases as is their generation: mapping function properties in a *first phase*. In a *second phase* lemmata proving the correctness of symbolic execution steps. The *third phase* verifies our *simulation criterion*.

5.3 Proving the Mapping Function Properties

Crucial to our proofs is the fact that the *variable mapping* is injective: If we change a variable and a corresponding memory cell no other variable's memory cell is effected. In the *first phase* this proof of injectivity is done in an *inductive* way. This means: we prove that a mapping with one variable mapped to a memory or register location is injective. With adding additional variables we prove that the mapping comprising the additional *variable to new memory or register location* is still injective. To do this in a simple way we use a memory address counter. All prior variable's memory locations are below this counter. Hence, if we assign a new memory location and it is equal or above this counter the resulting mapping will be injective again. For local variables the proof is done with locations relative to a stack pointer. This proof is combined with proving additional characteristics of the *variable mapping*. For example it is vital for the verification of operations involving dynamic array accesses that the following holds:

the address of $a[i]$ *is the address of* $a[0] + 4 * i$ (4 is the integer width)

Mapping function properties only have to be recomputed if the layout of the *variable mapping* changes. Furthermore it is possible although not yet implemented to partially reuse the proofs for old variable mappings when additional variables are added and the mapping for the old variables does not change.

```
Lemma step14
 forall varvals regs mem outp locvarstack,
  ... assumptions/facts ... − >
  statecomp
     (mkilstate 0 outp varvals locvarstack 14)
     (mktlstate 0 outp regs mem 64)
     Vars MemMap pcrel
     − >
  statecomp
     (ilnext (mkilstate 0 outp varvals locvarstack 14) ilprog)
     (tlnextn (mktlstate 0 outp regs mem 64) tlprog 3)
     Vars MemMap pcrel.
```

Fig. 7. Lemma for one Symbolic Execution Step

The proof of injectivity is by far the most time consuming part of the proofs conducted in the *first phase*. With an adequate encoding of the mapping function other requirements are almost trivial to prove.

5.4 Proving Symbolic Execution Steps

The *second phase* realizes the case distinction on all possible intermediate language statements. For each corresponding pairs of intermediate language statement and MIPS instructions we generate and check a separate lemma that the requirements of the *statecomp* (cp. Section 4) predicate are preserved during the state transition. Before unfolding *statecomp* and checking that its requirements are fulfilled we compute a symbolic representation of the states to be reached via the current execution step. A single symbolic representation of these states in Coq is crucial for easing the complexity of the proof scripts.

A typical lemma formalizing the correctness of one symbolic execution step is shown in Figure 7 As described in Section 3 *evalstatement* and *evalNinstructions* are state transition functions. It differs from the *simulation criterion* in the way that program points and the number of steps to be execued in the MIPS code are initialized with concrete values.

Figure 8 shows the corresponding code pieces that are proved to fulfill the simulation requirements. A global variable [42] is increased by one. 268500992 is the address it is mapped to. Checking the scripts generated in the *second phase* can be carried out in parallel since no step lemma depends on another.

5.5 Proving the Simulation Criterion

In the *third phase* we prove that the *simulation criterion* from Figure 5 is fulfilled. The correspondance of initial states can be done by simply unfolding the *statecomp* predicate.

Intermediate Language Statement	
ILPLUS (LVAR 42,VAR 42,CONST 1)	[42]:=[42] + 1
MIPS Code	
LOAD 8 268500992	register 8 := value_at (268500992)
ADD 8 8 1	register 8 := register 8 + 1
STORE 8 268500992	addr_at (268500992) := register 8

Fig. 8. Corresponding Code Pieces

The generated script for the simulation step makes a case distinction on all possible program points of the intermediate language procedure. Each execution step from such a program point is proved correct by applying the appropriate lemma from the *second phase*.

We showed that the proof can be split up in three phases. While the *first phase* proves a global property holding for the complete program the *second phase* proves independent lemmata for each intermediate language statement. The *third phase* finally proves our *simulation criterion* correct. Apart from that we use lemmata proved independently of concrete programs to speed operations up.

6 Evaluation of our Work

In this section we evaluate our certifying code generation phase. We focus on the generated proofs and especially the time it takes to check the proofs. In a previous work [4] it turned out that this is by far the bottleneck of our certifying compilers.

The table shows the time[2] it takes to prove the code generation of different programs correct. It shows the number of variables occuring in the program (counting array elements as single variables). The length of the original intermediate program (IL length) as well as the length of the generated MIPS code (TL length). In the last three columns the time it takes to check the proofs for the three different phases is shown.

[2] Experiments conducted on Intel Core 2 Duo machine with 2.16 GHz using one core and Coq Version 8.1.

program	no. variables	IL length	TL length	phase1	phase2	phase3
sort1	1008	16	67	3m 17s	5s	2s
sort1a	2008	16	67	10m 28s	5s	2s
sort1b	3008	16	67	21m 29s	5s	2s
sort1b	4008	16	67	36m 15s	5s	2s
sort1c	5008	16	67	54m 50s	5s	2s
arith1	16	177	705	2s	38s	29s
arith2	18	353	1409	2s	1m 53s	1m 51s
arrays1	2030	520	2059	10m 34s	4m 25s	4m 21s
arrays2	2030	1030	4107	10m 34s	14m 47s	9m 32s

The *sort* procedures sort arrays from 1000 (*sort1*) up to 5000 (*sort1c*) elements. The *arith* procedures mostly contain arithmetic operations while the *arrays* procedures perform operations on differently sized arrays. With procedures reaching several hundred lines of code the time it takes to check the proofs is increasing faster than linear. This is due to the larger data structures which have to be handled during the proof process. Accesses to these structures grow linear with code size however since the structures themself are growing linear we end up with a time that is growing quadratic. A similar argument holds for the *variable mapping* in the *first phase*. Compared to the Isabelle/HOL [15] (2005) implementation of [4] we are able to handle much larger programs. Verification times from several hours up to several days where typical for programs between 100 and 200 lines of code and up to 200 variables. We also proclaimed quadratic time behaviour in *phase2* since each proof for a single step lemma would grow linear with the size of the program due to the look up of statements, instructions, variable, memory and program counter correspondences from list like data structures. These look up operations where carried out in the Isabelle/HOL theorem prover mostly by unfolding the definition of a look up function and matching axioms describing the semantics of such a function against the definition and the data structure containing the data to be looked up. In Coq we are able to execute look up function definitions directly in the Coq environment. Hence the look up operations which where the bottleneck in our Isabelle implementation are not critical in our Coq implementation any more. Furthermore the trusted computing base is not enlarged.

Compared to the time it takes to check the proofs the time the proof generator takes to generate them and the compiler takes to generate the code is negligible. The proof generator size is with a few hundred lines of ML code comparable to its Isabelle/HOL counterpart in [4].

Our implementation and its performance evaluation demonstrates that certifying code generations is practicable for realistic compiler back-ends. As mentioned in Section 5.2 the time to conduct *phase1* can be significantly reduced by preproving common memory layouts. Time reduction is even easier to achieve for *phase2* since each lemma can be conducted in parallel. We believe that *phase3* could highly benefit from the use of proof terms that may enable us to abandon a large number of unifications done in this phase. This solution however might require a larger certificate generator. With these improvements it should be pos-

sible to conduct proofs even for procedures with significantly more than 10000 lines of code or variables in acceptable time.

7 Conclusion and Future Work

In this paper we have presented a methodology as well as an implementation of a certifying code generation phase. We did extend the code generation phase by a certificate generator producing Coq correctness proofs (certificates) for each compiler run. These are proved correct in the Coq system giving us the guarantee that the compiler has worked correctly. Our correctness criterion is independently of concrete transformations formalized in a higher order logic. In previous work we have shown that checking the certificates is the bottleneck in the certifying compiler approach. We did a great effort on reducing the speed for certificate checking by switching to the Coq theorem prover. It allows us to conduct time critical operations in a native way without enlarging the trusted computing base. Furthermore we have extended the involved languages and were able to further simplify our certification architecture. In our current implementation only minimal instrumentation of the compiler is required for our code generation phase. Therewith we have demonstrated the feasibility of the certifying compilation approach for the code generation phase of compilers.

A goal for the near future is to investigate in how far Coqs ability to generate and handle proof terms can make the proofs even more fast. Further goals comprise language extensions such as pointers and improvement of the other compiler phases.

Acknowledgement
The author would like to thank Arnd Poetzsch-Heffter for many valuable suggestions and comments on this paper.

References

1. A. W. Appel. Foundational proof-carrying code. In *LICS*, 2001.
2. J. O. Blech, L. Gesellensetter, and S. Glesner. Formal verification of dead code elimination in Isabelle/HOL. In *Software Engineering and Formal Methods*, pages 200–209. IEEE, IEEE Computer Society Press, September 2005.
3. J. O. Blech, S. Glesner, J. Leitner, and S. Mülling. A comparison between two formal correctness proofs in Isabelle/HOL. In *Proc. COCV Workshop , ETAPS 2005*, ENTCS, April 2005.
4. J. O. Blech and A. Poetzsch-Heffter. A Certifying Code Generation Phase. In *Proc. COCV Workshop, ETAPS 2007*, ENTCS, March 2007.
5. B. Buth, K.-H. Buth, M. Fränzle, B. von Karger, Y. Lakhnech, H. Langmaack, and M. Müller-Olm. Provably correct compiler development and implementation. In *Proc. CC '92*, volume 641 of LNCS, Springer-Verlag, 1992.
6. M. J. Gawkowski, J. O. Blech, and A. Poetzsch-Heffter. Certifying Compilers based on Formal Translation Contracts. Technical Report 355-06, Technische Universität Kaiserslautern, November 2006.

7. G. Goos and W. Zimmermann. Verification of compilers. In Bernhard Steffen and Ernst Rüdiger Olderog, editors, *Correct System Design*, volume 1710 of LNCS, Springer-Verlag, November 1999.
8. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.
9. S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *proc. POPL'05*, pages 364–377, ACM Press, 2005.
10. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *proc. POPL'06*, pages 42–54, ACM Press, 2006.
11. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *proc. PLDI'98*, pages 333–344, ACM Press, 1998.
12. G. C. Necula. Proof-carrying code. In *proc. POPL'97*, ACM Press, January 1997.
13. G. C. Necula. *Compiling with Proofs*. PhD thesis, 1998.
14. G. C. Necula. Translation validation for an optimizing compiler. In *proc. PLDI'00*, pages 83–95, ACM Press, 2000.
15. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of LNCS, Springer-Verlag, 2002.
16. D. A. Patterson and J. L. Hennessy. *Computer organization and design (2nd ed.): the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.
17. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *proc. TACAS*, volume 1284 of LNCS, 151+, Springer-Verlag, 1998.
18. A. Poetzsch-Heffter and M. J. Gawkowski. Towards proof generating compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):37–51, 2005.
19. M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.
20. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1* http://coq.inria.fr.
21. W. Zimmermann. On the Correctness of Transformations in Compiler Back-Ends. volume 4313 of LNCS, Springer-Verlag, 2006.
22. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A translation validator for optimizing compilers. In *proc. COCV Workshop*, ETAPS 2002, ENTCS, April 2002.
23. L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and Run-Time Validation of Loop Transformations. *Formal Methods in System Design*, 27(3):335–360, Springer-Verlag, 2005.