Certifying Code Generation with Coq

Jan Olaf Blech[1]

University of Kaiserslautern, Germany

Benjamin Grégoire[2]

INRIA Sophia Antipolis, France

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 present considerable advances to overcome this bottleneck. Our framework comprises a checker – an executable program that is formalized within a theorem prover to increase the speed of distinct sub tasks of certificate checking. We prove our checker correct and thus are able to use it instead of traditional proving techniques within our theorem prover environment. We compare our implementation featuring the Coq theorem prover to an older implementation. Our current implementation is feasible for medium to large sized programs.

Keywords: Translation Validation, Certifying Compilation, Theorem Proving, Coq

1 Introduction

Today's software systems are developed using high-level model or programming languages, even for 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 [3].

[1] Email: blech@informatik.uni-kl.de

[2] Email: Benjamin.Gregoire@sophia.inria.fr

[3] We follow the notions given by Xavier Leroy in [14].

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

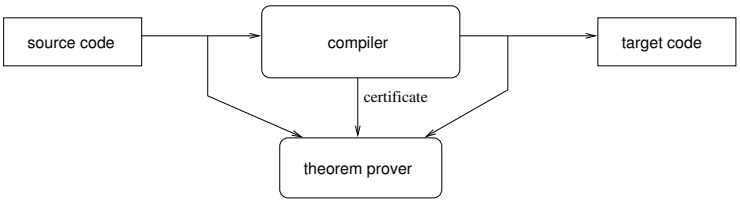

Fig. 1. Certifying Compiler

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 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. [6,22,8] on this problem).
- Second, similar to the proof carrying code approach ([17,16,1]), 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 large programs this may be very 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 [20] code. Our original certifying compiler framework is described in [6,8]. 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 [24] to specify our notion of compilation correctness and for checking the generated 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 automatically generates proof scripts as checkable certificates.
- User proved facts: The users of our compiler may provide facts they have proved on source code level. For example the users may provide the information that a variable used as an array index never exceeds the bounds of the array. We can conclude that a register or memory cell that corresponds to this variable in the MIPS code does not exceed these bounds, too. If this register or memory cell is used to index the memory corresponding to this array we can safely abandon bound checks. This is an optional feature if the users do not want to provide facts (maybe because they do not trust their source code analysis) they do not have to.

This work ports and improves the certification framework introduced in [6] to the Coq theorem prover. Compared to the old implementation we have encountered a great reduction of the run-time for conducting the correctness proofs and are now presenting a certifying compiler back-end that is able to handle realistically sized programs. Some of the speed improvement is achieved by using a checker – a feature that did not appear in our earlier implementation. This is a predicate formalized in an executable way within the theorem prover. Since we did prove our checker equivalent to a classical, non-executable specification we use it instead of the non-executable specification in our generated proof scripts. Furthermore, we have extended the involved languages to make them more detailed. Our intermediate language consists of arithmetic expressions, (array-)variable assignments, (un)conditional branches, a print statement, and (potentially recursive) procedure call and return statements. Our MIPS language comprises basic arithmetic operations, shift operations, and branch instructions. Instructions for handling outputs and procedure calls are provided. We 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 and the checker 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 [6,22,8] on certifying compilers the following approaches are most relevant to this paper.

In the translation validation approach [21,2,27] the compiler is regarded as a black box with at most 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 [18]. 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 checking unit and explicitly formalized semantics. Further on we use more information to generate the proof scripts from the compiler.

However, a translation validation checker (called validator) has been formally verified in [25]. Here – like in our work – correctness is based on a formalized semantics, too. The validator is used for verifying instruction scheduling. It is generated out of a verified Coq specification.

Credible compilation [23] 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 [17] 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 [15] a compiler generating certificates for the proof carrying code approach that guarantees that target programs are type and memory safe is described. The clear separation between the compilation infrastructure and the checkable certificate 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 [14], 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 [11]. The verification of an optimization algorithm is described in [4]; 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 [13]. A specification language is described for writing program transformations and their soundness properties. The properties are verified by an automatic theorem prover.

Important 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 [9,10,26] and in the ProCoS project [7]. The development of a formally verified compiler for a C subset is part of the Verisoft project focussing on pervasive formal verification of computer systems [12].

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.

```
operand ::=
  CONST Z | VAR Z | LOCVAR Z |
  ARRAYC (Z × Z) | ARRAYV (Z × Z)

loperand ::=
  LVAR Z | LLOCVAR Z |
  LARRAYC (Z × Z) | LARRAYV (Z × Z)

ilstatement ::=
  ILPLUS (loperand × operand × operand) |
  ILBRANCH1 (operand × N) |
  ILPRINT operand |
  ILCALL2 (loperand × N × operand × operand) |
  ILRET1 operand |
  ...
```

Fig. 2. Intermediate Language Syntax (excerpt)

```
ilstate :
(termstate : N, output : list Z, varvals : (Z × Z) ⇒ Z, locvarsstack : list ((Z ⇒ Z) × N), pc : N)
```

Fig. 3. Intermediate Language State

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.

The definition of a state in the intermediate language is show in Figure 3. It is a tuple consisting 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 is represented as a list of values. The next component is a mapping from global variables (including arrays) to values. The fourth component comprises a stack – formalized as a list – for local variables (including call arguments) and program counters. The latter ones serve as return addresses. Finally there is a program counter indicating the next statement to be executed. 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, and branch instructions. In addition instructions for basic output, 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

```
tlstate :
(tltermstate : N, tloutput : list Z, regs : Z ⇒ Z,mem : Z ⇒ Z, tlpc : N)
```

Fig. 4. MIPS Code State Definition

arguments in predefined spaces on the stack. As in the intermediate language code for procedures is stored as lists of instructions. The definition of 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. [5] 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.

We regard two programs as semantical corresponding if they generate the same output values in the same order. I.e. 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 the intermediate language program and the MIPS program to be in a (weak) simulation relation:

- 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.

Figure 5 shows our *simulation criterion* comprising the requirements for correctness

```
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

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 states *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 which is encoded using a function *MemMap* (cp. Section 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 proof process. However in other compiler phases other criteria have to be used (cp. [8]).

In this paper we do not regard stack overflows, but simply 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 of 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 including their use of checkers. 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.

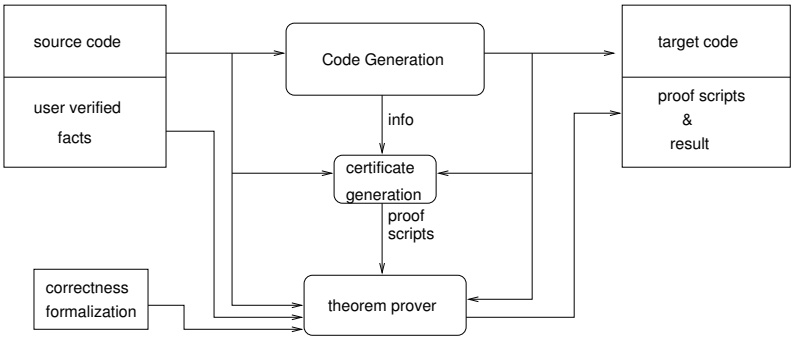

Fig. 6. Overview of Our Certifying Code Generation

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 correspondence between intermediate language and MIPS code. Finally the theorem prover is invoked to process the proof scripts. The theorem prover does this by using the formalized semantics and notions of correctness as well as derived lemmata and formalizations such as our checker. Thus it conducts the correctness of compilation. Facts proved on source code level may 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 altered. For local variables the proof is done with locations relative to a stack pointer. Apart from the injectivity proof additional characteristics of the variable mapping are proved in the first phase, too.

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)

```
check_inj' MemMap l max =
        match l with
        | nil => true
        | cons x l =>
        let im := MemMap x in
                if (is_greater im max) then check_inj' l im
                else false
        end.

check_inj MemMap l =
        match l with
        | nil => true
        | cons x l => check_inj' MemMap l (MemMap x)
        end.
```

Fig. 7. Injectivity Checker

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.

5.4 A Fast Injectivity Proof using a Checker

In the current implementation the injectivity proof is done by using a checker.

The general idea behind the technique described in this subsection is the fact that functions formalized in an executable way in Coq may be evaluated very fast. Other steps involving unification or rewriting of terms are considerably slower. Thus we want to keep as much executable as possible. Instead of verifying a proof goal with respect to some declarative correctness notion directly using traditional higher order theorem proving techniques we implement a predicate in Coq that computes whether the proof goal holds. The predicate has to be proved correct with respect to the original declarative correctness notion. This is done once and for all. Thus we can now reuse the executable predicate instead of the declarative correctness notion in any proof script produced by our certifying code generation phase.

A simple checker for an injectivity proof is shown in Figure 7. Two functions used for a fast computation of the injectivity of a variable mapping are presented. Both take a function realizing the variable mapping and a list of variables as inputs. The first function takes an additional max argument as input. It checks whether all variables in the list of memory addresses that are sufficiently larger than the address the previous element is mapped to. The second function is initially called and sets the value of max to the address of the first variable in the list. To use this function in our proofs we have proved that whenever a variable mapping fulfills this check_inj predicate with respect to a list of variables than it is indeed injective with respect to the list.

The algorithm realized by the check_inj predicate is the same in an earlier, non-checker based approach. However, traditional theorem proving formalizes each recursive application of the checker as at least one distinct lemma. To derive the lemma that if injectivity holds for one recursive application it will hold for the next if the addresses are mapped appropriate several unification steps are necessary.

The max does represent a single address value. It can be used with or without respect to an offset register. The is_greater is implemented appropriate. In our

```
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. 8. Lemma for one Symbolic Execution Step

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. 9. Corresponding Code Pieces

implementation it is also used to relate registers and memory addresses to each other.

We have proved our implemented injectivity checker correct in Coq with respect to the original injectivity specification.

5.5 Proving Symbolic Execution Steps

The second phase realizes the case distinction on all possible intermediate language statements. For each corresponding pair 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 8. 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 executed in the MIPS code are initialized with concrete values.

Figure 9 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.

5.6 Proving the Simulation Criterion

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

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 using 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 [6] it turned out that this is by far the bottleneck of our certifying compilers.

The table shows the time[4] it takes to prove the code generation of different programs correct. It shows the number of variables occurring 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.

program	no. variables	IL length	TL length	phase1	phase2	phase3
sort1	1008	16	67	3s	5s	1s
sort1a	2008	16	67	6s	5s	1s
sort1b	3008	16	67	10s	5s	1s
sort1c	4008	16	67	15s	5s	1s
sort1d	5008	16	67	22s	5s	1s
arith1	16	177	705	1s	38s	13s
arith2	18	353	1409	1s	1m 53s	46s
arrays1	2030	520	2059	5s	4m 25s	1m 34s
arrays2	2030	1030	4107	5s	14m 47s	6m 24s

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 themselves are growing linear we end up with a time that is growing quadratic. Compared to the Isabelle/HOL [19] (2005) implementation of [6] 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 the second phase 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

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

and program counter correspondences from list like data structures. These look up operations were 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. The use of the checker has speeded the first phase up by a factor slightly larger than 100 compared to a Coq implementation without a checker. Due to an artifact in the Coq implementation the time the third phase takes grows larger than linear with the size of the program code. In the table at hand we used explicit proof terms for conducting the third phase. We do have however a version with linear growing time in the third phase which is slightly slower for the last line in the table [5].

Furthermore, the trusted computing base is not enlarged by using Coq instead of Isabelle and adding a checker to the Coq implementation.

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 [6]. The use of the checker simplifies proof script generation for the first phase compared to a non-checker implementation. Proof script generation for the second phase is done by making a case distinction on the syntactical structure of the involved statement and operations. Each case is treated independently. It is easy to maintain and extend.

Our implementation and its performance evaluation demonstrates that certifying code generations is practicable for realistic compiler back-ends. One may not be willing to use certifying compilation for every compiler run, but at least for compiling production releases it can be applied. As mentioned in Section 5.2 the time to conduct the proofs from the first phase can be reduced by preproving common memory layouts. Time reduction is even easier to achieve for the second phase since each lemma can be conducted in parallel.

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 achieved serious results on reducing the speed for certificate checking by switching to the Coq theorem prover and making use of the checker. Coq and checkers allow us to conduct time critical operations in a native way without enlarging the trusted computing base. Furthermore, we have extended the involved

[5] A Coq implementation without checkers and explicit proof term use is described in [3].

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.

We are currently working on extending the checker approach to the second phase of our certifying code generation. The goal is to formalize and prove correct a fast checker that proves the symbolic execution steps. Moreover, in the case that our checker can not prove a step correct it shall return a list of conditions that need to be verified. The checked MIPS code may be regarded as a correct translation of the corresponding piece of intermediate code if these conditions are fulfilled. We believe that this feature makes the development of the checker in case of future language extensions more easy. It also allows for some interaction with the users in case that they provide some preproved facts. Further goals for the near future comprise language extensions such as pointers and improvement of the other compiler phases.

Acknowledgement
The authors 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 '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society, 2001.

[2] C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A Translation Validator for Optimizing Compilers. In *CAV 2005*, volume 3576 of *Lecture notes in computer science*, pages 291–295. Springer, 2005.

[3] J. O. Blech. On certifying code generation. Technical Report 366/07, University of Kaiserslautern, November 2007.

[4] 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.

[5] 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.

[6] J. O. Blech and A. Poetzsch-Heffter. A Certifying Code Generation Phase. In *Proc. COCV Workshop, ETAPS 2007*, ENTCS, March 2007.

[7] 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.

[8] M. J. Gawkowski, J. O. Blech, and A. Poetzsch-Heffter. Certifying Compilers based on Formal Translation Contracts. Technical Report 355-06, University of Kaiserslautern, November 2006.

[9] S. Glesner, G. Goos, and W. Zimmermann. Verifix: Konstruktion und Architektur verifizierender Uebersetzer(Verifix: Construction and Architecture of Verifying Compilers). *it- Information Technology*, 46(5):265–276, 2004.

[10] 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.

[11] 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.

[12] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *Software Engineering and Formal Methods*. IEEE, IEEE Computer Society Press, September 2005.

[13] 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.

[14] 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.

[15] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *proc. PLDI'98*, pages 333–344, ACM Press, 1998.

[16] G. C. Necula. Proof-carrying code. In *proc. POPL'97*, ACM Press, January 1997.

[17] G. C. Necula. *Compiling with Proofs*. PhD thesis, 1998.

[18] G. C. Necula. Translation validation for an optimizing compiler. In *proc. PLDI'00*, pages 83–95, ACM Press, 2000.

[19] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of LNCS, Springer-Verlag, 2002.

[20] 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.

[21] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *proc. TACAS*, volume 1284 of LNCS, 151+, Springer-Verlag, 1998.

[22] A. Poetzsch-Heffter and M. J. Gawkowski. Towards proof generating compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):37–51, 2005.

[23] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[24] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1* http://coq.inria.fr.

[25] J.-B. Tristan and X. Leroy. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *POPL '08: Conference record of the 35rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2008. ACM Press.

[26] W. Zimmermann. On the Correctness of Transformations in Compiler Back-Ends. volume 4313 of LNCS, Springer-Verlag, 2006.

[27] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, March 2003.