Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 190 (2007) 65–82

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# A Certifying Code Generation Phase

Jan Olaf Blech[1] Arnd Poetzsch-Heffter[2]

*Computer Science Department*
*University of Kaiserslautern*
*Germany*

**Abstract**

Guaranteeing correctness of compilation is a vital 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 reports on the construction of a certifying code generation phase for a compiler. It is part of a larger project aimed at guaranteeing the correctness of a complete compiler. We emphasize on demonstrating the feasibility of the certifying compilation approach to code generation and focus on the implementation and practical issues. It turns out that the checking of the certificates is the actual bottleneck of certifying compilation. We present a proof schema to overcome this bottleneck. Hence we show the applicability of the certifying compilation approach for small sized programs processed by a compiler's code generation phase.

*Keywords:* Translation Validation, Certifying Compilation, Theorem Proving, Isabelle/HOL

## 1 Introduction

Most software systems are described in high-level model or programming languages. Their runtime behavior, however, is controlled by the compiled code. For software in critical systems, it is of great importance that static analyses and formal methods can be applied on the source code level, because this level is more abstract and better suited for such techniques. However, the analysis results can only be carried over to the machine code level, if we can establish the correctness of the compilation. Thus, compilation correctness is essential to close the formalization chain from high-level formal methods to the machine-code level.

Two general approaches can be distinguished to establish the correctness of a compiler [3]:

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

[2] Email: poetzsch@informatik.uni-kl.de

[3] We follow the notions given in [11] and slightly refine them based on a discussion at the Dagstuhl Seminar 05311 "Verifying Optimizing Compilers".



doi:10.1016/j.entcs.2007.09.008

- *Certified compiler*: 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*). We call a compiler for which machine checked proofs for both items are developed a *certified compiler (algorithm/implementation)*.
- *Certifying compiler*: 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 an algorithm or its implementation, but only about the relation of two programs. Different techniques have been developed to generate such proofs automatically (see Sect. 6).

Compared to compiler certification, the technique of compilers certifying their results has two advantages. First, the issue of implementation correctness can be completely avoided, that is, we do not have to trust the implementation of the compiler algorithms on a hardware system or prove it correct (cf. [19,5] on this problem). Second, similar to the proof carrying code approach ([14,13,1]), 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. The disadvantages of the certifying compiler approach is that users have to check the certificates for each (critical) compilation and this check might fail if the compiler has a bug.

We have constructed a certifying compiler, translating a C subset to MIPS [17] code. Our certifying compiler framework is described in [5]. It comprises the following features

- Machine-checkability and independence of logic: All specifications and proofs are machine-checkable based on a formal general logic, that is, a logic that is independent of languages and techniques used in the translation. We use Isabelle/HOL [16] as our specification and verification framework.
- Translation contract: We require an explicit *translation contract* formally specifying the semantics of source and target language and a translation correctness predicate.
- Certifying compiler: We are interested in a technique where the compiler generates proof scripts as checkable certificates.

This paper presents details about the construction and extension of the code generation phase as well as the checking of the certificates. We emphasize on the practical applicability of our approach. This means in particular to optimize the time it takes to check a certificate. The main technical contributions of this paper are:

- The presentation of our concrete certifying code generation phase. This includes

the extensions necessary to generate the certificates.

- The structuring and optimization of the compiler generated certificates to minimize the time it takes to conduct the proofs. For this goal, we distinguish between program independent parts, parts that have to be conducted once per program and parts that have to be done for each instruction in the program separately. In particular we present a solution to prove certificates more efficiently correct by proving the injectivity of a mapping function once for each program enabling us to abandon a complicated case distinction for each instruction in the program.
- Experimental results, experiences, effort assumptions, and technical propositions on how to run proofs more efficiently. (To the best of our knowledge, we are the first who developed a prototypical implementation of this approach.)

*Overview of the Paper*

We describe the intermediate language, the generated MIPS machine code and their relation in Section 2. The code generation algorithm and the certifying correctness proofs are described in Section 3. Section 4 describes the automation and performance enhancement of the certifying process. In Section 5, we evaluate our work. Related work is discussed in Section 6 and a conclusion is drawn in Section 7.

## 2 The Languages and their Semantical Equivalence

In this section we sketch syntax and semantics of our intermediate and MIPS language as well as semantical equivalence between them. This section builds on the work presented in [5,19]. Both intermediate and MIPS semantics are defined in a small-step operational way. Hence we give definitions of syntax as abstract datatypes, states as tuples and transition rules as nextstate functions. Two programs are regarded as semantically equivalent if they produce the same output traces.

### *2.1 The Intermediate Language*

The definition of the intermediate language's syntax is depicted in Figure 1. The language comprises (array-)variable assignments, expressions, conditional and unconditional branches, a print statement for output, and an exit statement. Programs are lists of statements.

The semantics of the intermediate language is shown in Figure 2. It is formalized as a state transition function *evalstatement* in Isabelle/HOL. To shorten the presentation, it is slightly simplified. A system state comprises three components. The first is a sequence of outputs that have occurred so far represented as a list. The # denotes the appending of an element to a list. The second one consists of a program counter indexing the current instruction of the program. The last one the *memorystate* is a mapping from variables to their values. Variables are represented as an integer tuple. The first component corresponds to the variable name. The second represents the index in case of an array or stays 0 in the case of a primitive

```
datatype operand =
        CONST int | VAR int | ARCONST int int | ARVAR int int


datatype expression =
        OPERAND operand |
        PLUS operand operand | MINUS operand operand | MULT operand operand |
        LT operand operand | LE operand operand


datatype statement =
        ASSIGN_V int expression | ASSIGN_AC int int expression | ASSIGN_AV int int expression |
        BRANCH expression int | GOTO int |
        PRINT int | EXIT
```

Fig. 1. Intermediate Language Syntax

```
evaloperand varvals (CONST c) = c
evaloperand varvals (VAR v) = varvals (v,0)
evaloperand varvals (ARCONST v i) = varvals (v,i)
evaloperand varvals (ARVAR v vi) = varvals (v,varvals(vi,0))


evalexpression varvals (OPERAND o1) = evaloperand varvals o1
evalexpression varvals (PLUS o1 o2) = evaloperand varvals o1 + evaloperand varvals o2
evalexpression varvals (MINUS o1 o2) = evaloperand varvals o1 - evaloperand varvals o2
evalexpression varvals (MULT o1 o2) = evaloperand varvals o1 * evaloperand varvals o2

evalexpression varvals (LT o1 o2) = { evaloperand varvals o1 < evaloperand varvals o2  then 1
                                    {                                                  else 0

evalexpression varvals (LE o1 o2) = { evaloperand varvals o1 ≤ evaloperand varvals o2  then 1
                                    {                                                  else 0

evalstatement (outp,pc,varvals) (ASSIGN_V v e1) =
                (outp,pc+1,varvals((v,0):=evalexpression varvals e1))
evalstatement (outp,pc,varvals) (ASSIGN_AC v i e1) =
                (outp,pc+1,varvals((v,i):=evalexpression varvals e1))
evalstatement (outp,pc,varvals) (ASSIGN_AV v vi e1) =
                (outp,pc+1,varvals((v,varvals vi):=evalexpression varvals e1))
evalstatement (outp,pc,varvals) (BRANCH e1 lab) =
                        { evalexpression varvals e1 = 1  then (outp,lab,varvals))
                        {                                else (outp,pc + 1,varvals)
evalstatement (outp,pc,varvals) (GOTO lab) =
                (outp,lab,varvals)
evalstatement (outp,pc,varvals) (PRINT v) =
                ((varvals v ) # outp,pc+1,varvals)
evalstatement (outp,pc,varvals) (EXIT) =
                (termination # outp,pc,varvals)
```

Fig. 2. Intermediate Language Semantics

variable. For simplicity we only regard infinite integers as types in the language definition. However, in an extended version we also handle booleans which did not complicate the semantics definitions and the proofs too much.

### *2.2 The MIPS Language*

The definition of the MIPS syntax can be found in Figure 3. As in the intermediate language programs are lists of instructions.

It should be noted that the PRINTINT and CHECKARRAYSIZE instructions are not genuine MIPS instructions. They consist of up to three real instructions but are handled as one atomic instruction throughout this paper for simplicity reasons.

```
datatype instruction =
    ADD int int int | ADDI int int int | SUB int int int | MLT int int int |
    SLT int int int | SLTI int int int | SLE int int int | SLEI int int int |
    STORE int int | LOAD int int |
    BGTZ int int | J int |
    PRINTINT int | CHECKARRAYSIZE int int | RETURN
```

Fig. 3. MIPS Syntax

```
evalinstruction (outp,pc,regs,mem) (ADD r1 r2 r3) = (outp,pc+1,regs(r1:= (regs r2) + (regs r3)),mems)
evalinstruction (outp,pc,regs,mem) (ADDI r1 r2 c) = (outp,pc+1,regs(r1:= (regs r2) + c),mems)
evalinstruction (outp,pc,regs,mem) (SUB r1 r2 r3) = (outp,pc+1,regs(r1:= (regs r2) - (regs r3)),mems)
evalinstruction (outp,pc,regs,mem) (MLT r1 r2 r3) = (outp,pc+1,regs(r1:= (regs r2) * (regs r3)),mems)
evalinstruction (outp,pc,regs,mem) (MLT r1 r2 r3) = (outp,pc+1,regs(r1:= (regs r2) * (regs r3)),mems)
evalinstruction  (outp,pc,regs,mems)  (SLT  r1  r2  r3 ) =
                  { regs r2 < regs r3  then (outp,pc + 1,regs(r1:=1),mems)
                  {                    else (outp,pc + 1,regs(r1:=0),mems)
evalinstruction  (outp,pc,regs,mems)  (SLTI  r1  r2  c ) = ...
evalinstruction  (outp,pc + 1,regs,mems)  (SLE  r1  r2  r3 ) = ...
evalinstruction  (outp,pc + 1,regs,mems)  (SLEI  r1  r2  c ) = ...
evalinstruction (outp,pc,regs,mem) (STORE r1 r2) = (outp,pc+1,regs,mems(regs(r2):= regs(r1)))
evalinstruction (outp,pc,regs,mem) (LOAD r1 r2) = (outp,pc+1,regs(r1 := mems (regs r1)),mems)

evalinstruction (outp, pc, regs, mems) (BGTZ r1 lab) = { 0 < regs r1  then (outp,lab,regs,mems)
                                                       {               else (outp,pc + 1,regs,mems)
evalinstruction (outp,pc,regs,mem) (J lab) = (outp,lab,regs,mems)
evalinstruction (outp,pc,regs,mem) (PRINTINT r1) = ((regs r1) # outp,lab,regs,mems)
evalinstruction  (outp,pc,regs,mems)  (CHECKARRAYSIZE  r1  c ) =
                  { 0 < regs r1  ∧  regs r1 < c  then (outp,pc+1,regs,mems)
                  {                              else (abnterm # outp,pc,regs,mems)
evalinstruction (outp,pc,regs,mem) (RETURN) = (term # outp,lab,regs,mems)
```

Fig. 4. MIPS Semantics

The (slightly simplified) MIPS semantics is shown in Figure 4. It is formalized as a state transition function, too. The MIPS state is a four tuple. Instead of the intermediate language's *memorystate* we have a (register → value) and (memory location → value) mapping. Registers and memory locations are simple integer values. The resolution of array variables to memory locations is an interesting task for compiler generated correctness proofs. Our semantics also needs a state transition function executing several instructions at a time: *evalNinstructions*. It makes use of the *evalinstruction* function and takes the number of instructions to be executed as well as a complete MIPS program as input.

The MIPS processor was chosen because of its simple architecture, its wide area of usage, and the availability of a simulator. Programs of the introduced subset of the MIPS language can be run within this simulator.

### *2.3 Semantical Equivalence*

In order to verify that a transformation has been conducted correctly one needs a notion of semantical equivalence. Lots of research has been done on the topic of

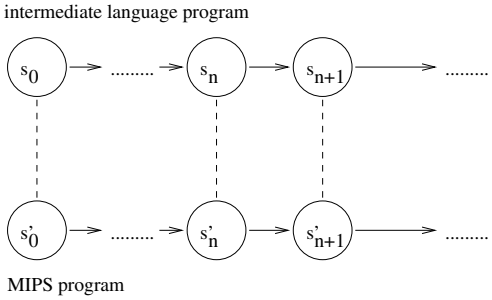


Fig. 5. Simulation Principle

semantical equivalence of programs (cf. Section 6). A very simple notion is to say that two programs are equivalent if they terminate in equivalent states or return the same result. Since we do not only want to compile programs that have to terminate, we need a notion that captures the non terminating cases adequately. Within our project we regard two programs as semantical equivalent if they have the same output traces. This is similar to the ideas presented in [22,2].

For the conduction of correctness proofs however, it is much more useful to use a more restricted criterion that implies the observable traces. An intermediate language and a MIPS program will have the same output traces if they calculate and output equivalent values during their execution. Hence in order to formalize this correspondence we need a relation between variable names from the intermediate language and memory/register locations from the MIPS code. We prove that variables and memory/register locations have corresponding values during each step of execution of intermediate and MIPS language. This will imply the equivalence of the output traces.

Formally we require both intermediate language and MIPS program to be in a (weak) (bi-)simulation [4] relation (cf. Figure 5):

- The initial states have to have corresponding values for variables and memory locations (= have to be equivalent).
- For two equivalent 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.

Figure 6 shows our concrete top level definition of the program equivalence criterion as formalized in Isabelle/HOL. The criterion is a predicate taking a source and target program and returning true or false. The criterion itself does not make any specific requirements on the simulation relation. PCREL relates program points in source and target program to each other. VARMAP is a *variable mapping* that maps variables to register/memory locations. STEPS provides the number of steps to be executed on the target language side. The predicate can only be proved by the theorem prover if the existentially quantified PCREL, STEPS, and VARMAP are

[4] Since we are only talking about deterministic systems bisimulation and simulation can be regarded as equivalent in our case.

```
consts
  IL_TL_prog_equiv :: " ILProgram => TLProgram => bool"
defs
IL_TL_prog_equiv_def:
  "IL_TL_prog_equiv ilprog tlprog ==
     EX STEPS PCREL VARMAP.
     (0,0) : PCREL &
     (ALL m.
        IL_TL_equiv IL_initial_state VARMAP PCREL TL_initial_state) &
     (ALL PCIL PCTL . (PCIL,PCTL) : PCREL --- >
       (correctstep ilprog tlprog PCIL PCTL (STEPS PCTL) VARMAP PCREL))"
```

Fig. 6. Program equivalence criterion

instantiated with their *correct* values. These are generally provided by the compiler and included in the proof scripts. If the compiler provides false values the proof will not succeed.

The simulation relation requirements can be identified in the definition. We have the initial requirement that the start states are in the relation (first two elements of the conjunction). The last element of the conjunction formalizes the step part of the simulation definition. It makes use of the predicate *correctstep* ensuring that for each corresponding intermediate language and MIPS states in the simulation relation the succeeding states will be in the relation again.

We have introduced an intermediate language and a certain subset of the MIPS processors instructions as well as their semantics in this section. We have also motivated and introduced a criterion for semantical equivalence. Note that our semantics formalism does not comprise integer arithmetics yet. This is an abstraction from the MIPS machine's finite integer representation. Hence errors occurring due to limited size of integer representations are not an issue of this paper. The focus of this paper is on the applicability of the certifying compiler approach not on semantical features of the involved languages (see e.g. [3] for approaches to defining and reasoning about semantics of a more sophisticated intermediate language). The program equivalence criterion used 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 $(1 : n)$ criterion is sufficient and simplifies the prove process. However in other compiler phases other criteria have to be used (cf. [5]).

## 3 The Code Generation Phase

In this section we describe the process of code generation that is subject to the verification process described in this paper. We also sketch a general strategy to prove that generated MIPS code of a program is semantical equivalent to its intermediate language representation.

### *3.1 The Code Generation Algorithm*

In the first step of our implemented code generator register allocation is performed. It is determined whether a variable's values shall be stored in a register or memory. In our implementation we use a very simple register allocation algorithm that maps the first 10 non-array variables in registers and all others (especially arrays) to memory. It should be noted that our technique can handle much more sophisticated register allocation schemas. In the next step we allocate memory locations for the non-register mapped variables. One result of these steps is a mapping from intermediate language variables to registers and memory addresses (*variable mapping*). This mapping is not only used during the compilation process but is also vital for conducting the proofs.

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

In a last pass through the MIPS program jump targets are resolved with the help of this program point correspondence relation. This relation is also very helpful for conduction of the correctness proof.
The whole compiler is implemented using the ML programming language.

### *3.2 Proving Correctness of Compilation*

In order to prove a codegeneration run correct we have to show that intermediate language program and the resulting MIPS program fulfill the correctness criterion presented in Figure 6. Hence we have to show that both programs simulate each other. They have to meet the requirements of the simulation relation from Section 2.3.

- We have to prove that the initial states of both programs are in the simulation relation. This is done by simply unfolding the equivalence criterion definitions.
- We have to prove that for each two equivalent ($\equiv$) states from intermediate and MIPS program $s_{\texttt{IL}}$ and $s_{\texttt{MIPS}}$ the succeeding states are equivalent again:

  $s_{\texttt{IL}} \equiv s_{\texttt{MIPS}} \Longrightarrow$
  $\texttt{evalstatement}\ s_{\texttt{IL}}\ (\texttt{picstat}\ s_{\texttt{IL}}\ \texttt{IL}) \equiv \texttt{evalNinstructions}\ s_{\texttt{MIPS}}\ (\texttt{corsteps}\ s_{\texttt{MIPS}}\ \texttt{IL}\ \texttt{MIPS})\ \texttt{MIPS}$

As described in Section 2 *evalstatement* and *evalNinstructions* are state transition functions. The *picstat* picks the appropriate statement out of the intermediate language program. *corsteps* gives the number of corresponding MIPS instructions to an intermediate language statement.

The problem with this item is, that we have very few information about the $s_{\texttt{IL}}$ and $s_{\texttt{MIPS}}$, but that they are in the simulation relation and that they are states of a certain program. If the $s_{\texttt{IL}}$, $s_{\texttt{MIPS}}$ are terminal states this follows directly from our definition of the semantics (cf. definitions of EXIT and RETURN semantics in

Section 2). If the program did not terminate there must be some statement that will be executed next in the intermediate language program. Hence we make a case distinction on all statements of the intermediate language program. The simulation relation relates this statement to machine code instructions in the MIPS program. We have to prove that the execution of the intermediate language statement results in an equivalent state to the state reached by the execution of the corresponding MIPS program points.

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 2. 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.

# 4 Automating the Certification Process

In this section we describe the proofs that our compiler generates to assure that code generation has been performed correctly. The conduction of the proofs is explained as well. We also describe the parts of the compiler that have to be modified for certificate generation. In the last part of this section we examine performance issues, i.e. the time the theorem prover takes to show that our generated proofs are correct. We show how restructuring of the proof principle makes the checking faster.

## *4.1 The Generation and Conduction of the Proofs*

In this subsection we describe the generation of the correctness proofs and their automatic conduction in Isabelle/HOL. Whenever a compiler (codegeneration) run has been completed we want to conduct a correctness proof. We invoke the theorem prover that gets different files. The files containing the translation contract (syntax, semantics definitions of the involved languages, and the criterion of semantical equivalence) are program independent and must not be generated by the compiler. Some other performance enhancing properties are preproved and non compiler generated as well. The concrete proof is generated by the compiler as well as the dumping of intermediate (IL) and MIPS language.

Figure 7 shows the compiler, codegeneration phase resp. as well as the theory files generated by the compiler and their dependencies (dashed lines).

The codegeneration phase as implemented in our compiler takes an intermediate language program and outputs MIPS assembly code. In addition, several theory files for use with the Isabelle theorem prover are created. These files are a collection of facts about the program and compilation process as well as subproofs. At the very beginning the original intermediate language program is converted to an Isabelle representation and written in a separate file. Likewise at the end the MIPS assembly

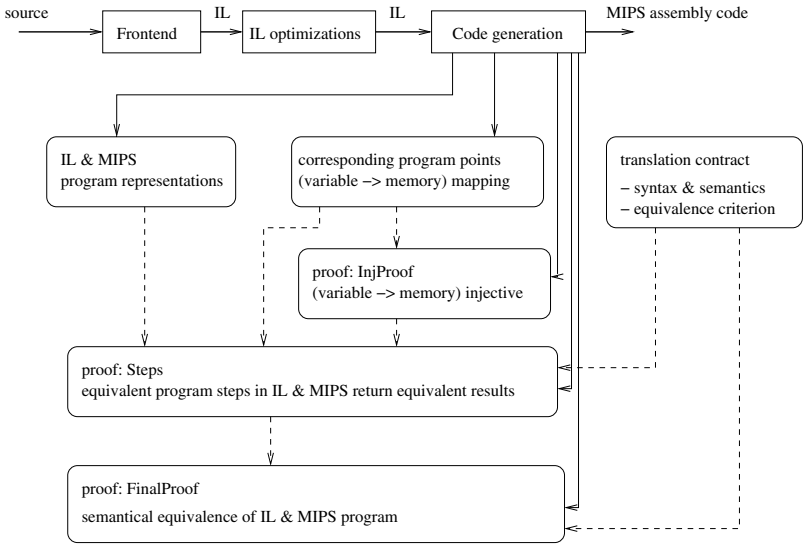


Fig. 7. The compiler and the generated theory files with dependencies

code is written to an Isabelle file as well.

There is also a file where the correspondence of program points in intermediate language and MIPS programs and the *variable mapping* are saved. This information is computed in the compiler anyway (cf. Section 3) and is simply written into Isabelle files. Note that it is *only* important for the conduction of the proofs. It is not needed for the semantical equivalence criterion itself. Finally InjProof.thy, Steps.thy and FinalProof.thy are files containing the correctness proof. They are computed by a special *proof generator* module of our certifying compiler that is independent of the rest of the compiler. However, this module gets intermediate language code and generated MIPS code as well as the relation of corresponding program points and the *variable mapping* as input. The generated files do rely in several additional non-compiler generated theory files that contain the translation contract. For performance reasons we have written a large collection of preproved lemmata that are non-compiler generated as well. This collection currently comprises 70 lemmata. Most of them perform relatively simple transformations like splitting a proof goal to several smaller proof goals or stating some generic equalities and implications. They encapsulate operations that frequently occur during the proving process. FinalProof.thy contains the topmost correctness criterion.

In order to prove semantical equivalence with respect to output traces of an intermediate language program and a MIPS program one has to prove that it fulfills the criterion for semantical equivalence described in Section 2.3.

In a first step the theorem prover proves that the initial states are in an equivalence relation. This is a simple task done directly in the Finalproof.thy file. The next task is to prove that for each pair of equivalent states the succeeding states are equivalent again. In practice this is done with a large case distinction on the locations one could be in a concrete program as described in Section 3.2. On these locations both intermediate language and MIPS programs are executed symboli-

cally, i.e. with abstract not concrete states. One shows that the resulting states will be in the equivalence relation again. We prove these steps with independent lemmata which are themselves proved correct in Steps.thy.

A lemma from Steps.thy proves for each corresponding program locations the execution of one step correct. This is done by looking at the operands and operations of both intermediate language and MIPS program. They have to correspond to each other. If the *variable mapping* is injective the remaining proof of semantical equivalence is quite easy and relatively fast to prove. A typical compiler generated step aimed for efficient usage by the automatic theorem prover Isabelle/HOL is depicted in Figure 8. The first part of the proof for this step is the actual correctness lemma formulated in quotation marks. We have to deduct from the equivalence of two states (*IL_TL_step_equiv*) the equivalence of two succeeding states. The second part consists of the proof of the lemma and is itself a kind of program to be interpreted by the theorem prover. Different tactics are applied to conduct the proof. The lemma states that an assignment to an array variable is correct. Hence the case distinction if the array index is within the array's bounds at the beginning of the proof script. The actual instructions and statements of MIPS and intermediate language are not displayed but occur only as references to a list of instructions/statements. In the next lines the program steps are symbolically evaluated i.e. the effect of the state transition on the original states is represented as a formula. In the next paragraph the defintion of state equivalence is unfolded (*IL_TL_step_equiv*) and different requirements are proved. The first one is the proof that the treated instructions correspond to each other. Afterwards the equivalence of the resulting memory/register values and variables values are proved. This is where the injectivity(*injtheorem*) of the *variable mapping* (*MapFun m*) comes into play.

Hence the compiler proves the injectivity of this mapping in a seperate file: InjProof.thy. Temporary values that occur during MIPS code execution are stored in several special registers where no variable is mapped to.

### *4.2 Performance Issues*

This section deals with performance issues of our certifying code generator and its Isabelle proofs. The time it takes to generate and conduct the proofs is crucial for the acceptance of the certifying compiler approach in industry and science. We present strategies that made the conduction of the proofs much faster than our first naive approaches. The time the proof generation within the compiler takes is negligible in the code generation phase. The actual code generator takes almost linear time, including the part that writes the Isabelle representation and proofs into files. However, the verification of the proofs within the Isabelle theorem prover takes a comparable large amount of time and it was a nontrivial task to reduce this time. This section presents our implemented solutions as well as estimations of their runtime complexity.

Throughout this section we will use $|P|$ as the number of program instructions/operations in either intermediate and MIPS program. $|V|$ is the number of

```
lemma step16: "[|
....
IL_TL_step_equiv (ac_IL,outp_IL,terms_IL,(16,varvals)) RelVarSet (MapFun m) PCrel (ac_TL,outp_TL,terms_TL,(44,memvals)) |]
==>
IL_TL_step_equiv (evalstatement (ac_IL,outp_IL,terms_IL,(16,varvals)) (ith ILprog 16) ) RelVarSet (MapFun m) PCrel (evalNinstructions (ac_TL,outp_TL,terms_TL,(44,memvals)) (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0)))))))) TLprog)"

        apply (case_tac "0 <= memvals -15 & memvals -15 < 10")

apply (subst evalstatementsplitter,subst ILprog_def,( ((rule ith1, simp(no_asm),simp(no_asm)))+)?,simp (no_asm),simp (no_asm))
apply (subst evalNinstrsplitter0 | (subst evalNinstrsplitter1,(subst TLprog_def,(((rule ith1, simp(no_asm),simp(no_asm)))+)?,simp(no_asm)), simp))+
prefer 2
apply (subst evalstatementsplitter,subst ILprog_def,( ((rule ith1, simp(no_asm),simp(no_asm)))+)?,simp (no_asm),simp (no_asm))
apply (subst evalNinstrsplitter0 | (subst evalNinstrsplitter1,(subst TLprog_def,(((rule ith1, simp(no_asm),simp(no_asm)))+)?,simp(no_asm)), simp,(subst if_not_P,simp )?,(simp only: Let_def)?))+

apply (simp add: IL_TL_step_equiv_def)
apply (rule s4ir | rule s3ir | rule s2ir | rule s1ir)
apply (simp (no_asm) only : One_nat_def PCrel_def set_t1 set_t1h1 set_t1h1' set_t1h0, simp)
apply simp ?
apply (rule injtheorem)
apply (simp (no_asm) only : One_nat_def RelVarSet_def set_t1 set_t1h1 set_t1h1' set_t1h0)
apply (rule rangetheorem', simp (no_asm)) +
apply (simp only: MapFun_def fun2at fun2at' fun2bt)
apply (rule opsplitterplus,(rule_tac memvals=memvals in varconc,simp+,simp (no_asm) only : RelVarSet_def set_t1 set_t1h1 set_t1h1' set_t1h0,simp (no_asm) only : MapFun_def fun2at fun2at' fun2bt)+)?
apply (rule_tac m=m and c=9 and MapFun=MapFun in dynarrayacc, simp, simp, simp, simp (no_asm) only : MapFun_def fun2at fun2at' fun2bt, rule w100, rule_tac memvals=memvals in varconc,simp+,simp (no_asm) only : RelVarSet_def set_t1 set_t1h1 set_t1h1' set_t1h0, simp (no_asm) only : MapFun_def fun2at fun2at' fun2bt,rule_tac memvals=memvals in varconc, simp +, rule_tac c=9 in dynarrayacc2, simp, simp, simp ,rule v100 )?
apply simp +
done
```

Fig. 8. Generated proof of one step

variables. We count each array element as a separate variable. $|A|$ is the number of arrays in a program. The following properties hold in our code generation framework:

- Since programs in Isabelle are represented as lists of instructions the lookup of an instruction takes $O(|P|)$ time.

- Lookup of a variable in a set of variables as well as the lookup of a register/memory location in a *variable mapping* takes $O(|V|)$ time. We instantiated the Isabelle simplifier tactic with some simple lookup lemmata/rules that prohibit other than linear processing of set and mapping function definitions to achieve this result.
- The lookup of an element in a program counter relation, i.e. the relation of corresponding program points in intermediate language and MIPS code, can be done in $O(|P|)$ time.
- As presented in Section 3.2 the verification of abstracted simulation steps is crucial to our correctness proof. Since the proof of a single step lemma in Steps.thy does need to make a constant number of lookups to get corresponding instructions as well as a constant number of lookups to get corresponding variable/memory/register locations one step in the Steps.thy file can be proved in $O(|P| + |V|)$ time. However a step can only be proved correct with this effort if additional requirements of the *variable mapping* have been proved in advance. It has to be ensured that only the variable/memory/register locations appearing as the instructions parameters are effected. This means that we have to require that if one writes to a MIPS memory location corresponding to an intermediate language variable no memory location corresponding to another memory location is changed. Thus we have to require that the *variable mapping* is injective. A second requirement puts restrictions on the alignment of array addresses. These tasks are done in a separate proof and only once per program.

### *4.3 Proving the Mapping Function Properties*

The proof of injectivity of the *variable mapping* between variables and memory/register locations is done in an *inductive* way. This means: we prove that a mapping with one variable and memory/register location is injective. With adding additional variables we prove that the mapping comprising the additional *variable to new memory location* is still injective. In order to do this in a simple way we use a memory counter. All prior variable's memory locations are below this memory counter. Hence, if we assign a new memory location and it is equal or above this counter the resulting mapping will be injective again. This proof is combined with a second one that states a property vital for the resolution of array addresses to memory locations.

A schema for proving the injectivity of the mapping function properties is described below.

- For each new *variable mapping* we proof a lemma:
  *using that there was no variable mapped to a memory location above a certain address before and the fact that the actual memory location is mapped to this certain address we prove that the mapping is still injective.*
  This takes O(1) time. We also proof that there is no memory location mapped to this (certain address + 4(integer width)) for use in the next step. This can be done in O(1) time, too.
- Since there are $|V|$ variables the complete injectivity proof for the mapping func-

tion takes $O(|V|)$ time

As mentioned above throughout the construction of the mapping we have to prove additional lemmata for use in the correctness proofs of the steps: 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)

This is proved for each array in the original program with the construction of the mapping function as well and requires at most one additional lemma ($O(1)$) for each array definition and each element in the mapping function. Therefore the whole process of proving injectivity and "arithmetic" correctness of array mapping may take up to $O(|V| \cdot |A|)$ time (with $|A|$ being the total number of arrays in the program). Since $|A|$ could be at worst $|V|$ one could argue that proving the properties of the mapping function might take quadratic time. This however only occurs in rather pathological cases, since few programs consist of arrays with only one element.

In a first approach we did not have an injectivity proof. Instead we did prove the equivalence of execution steps by making a case distinction over all used variables of a program and shown that for each one of them if the corresponding memory locations of the MIPS have the same values they will have the same values after this step, too.

Due to multiple lookup operations this process turned out to need time squared to the number of variables in a program for each program point. In our current version we dismiss of the case distinction because we know that the *variable mapping* is injective.

The verification in our current system of a complete code generation takes $O(|V| \cdot |A| + |P| \cdot (|P| + |V|))$ time. This is $O(|P|^2 + |V| \cdot |P|)$ for non pathological cases. Since the $|P|^2$ gets in because of simple lookup operations in a list representing the program and the $|V| \cdot |P|$ is lookup of the operands, we believe that this is close to optimal with using standard Isabelle/HOL datattypes. We believe however that this result could be improved by using more efficient datastructures or more efficient implementations of datastructure operations in the Isabelle internal parts.

We showed that the proof can be split up in parts that may be proved independently of concrete programs, once per program and once per instruction. We optimized our proof by proving a property (injectivity of the mapping function) once per program giving us the possibility to abandon a large case distinction that had to be conducted once per program instruction.

# 5 Evaluation of our Work

In this section we evaluate the implementation of our code generation phase. We focus on crucial parts of our implementation and show some statistics. We have implemented a complete compiler comprising a frontend, an intermediate representation with optimizations as well as a code generation phase including register allocation. The code generation phase presented in this paper is well integrated into

this compiler.

The table shows the time[5] it takes to prove the codegeneration of a program *fibo*[6] and two different sorting programs correct within the theorem prover Isabelle/HOL. It also shows the length of the original intermediate program (IL length) as well as the length of the generated MIPS code (TL length). The time to verify a program is linear to the size of the variables (counting array elements as single variables) and the length of the program.

| program | no. variables | IL length | TL length | time to prove correct |
|---|---|---|---|---|
| fibo (100 elements) | 112 | 13 | 55 | 762 s |
| fibo (200 elements) | 212 | 13 | 55 | 1518 s |
| sort | 46 | 58 | 261 | 6102 s |
| sort2 | 166 | 178 | 900 | 176424 s |

It can be seen that with doubling the amount of variables in the fibo program the verification of the code generation takes nearly twice as much time. The sorting algorithms take longer due to the processing of more and complicated instructions. Without the injectivity proof from section 4.3 all of these proofs would fail due to ressource limitations. However with intermediate language programs containing less than ten variables both approaches deliver comparable results.

The verification times presented here may be reduced significantly without changing the proofs if Isabelle's representation of datastructures and their operations are implemented and interpreted in a more native way within the ML environment Isabelle builds on. This is especially true for larger programs that suffer from the high time complexity due to inefficient lookups. This however might lead to a larger Trusted Code Base. Compared to the time it takes to conduct the proofs the time the compiler takes to generate them is negligible.

The original codegeneration phase comprises 334 lines of ML code. The proof generator has 864 lines of additional code. Note that both proof generator and compiler do not belong to the trusted computing base (TCB). A more sophisticated code generation phase could easily grow to more than 10000 lines of code. However, the proof generator part would stay almost the same.

Our academic prototype shows that certifying code generations is in general feasible for realistic compilers. It turned out that the time it takes to conduct a correctness proof in Isabelle/HOL is crucial. Most of the time complexity gets in because of linear lookup operations in Isabelle datatypes. The advent of more efficient datatypes in Isabelle/HOL can decrease the time and time complexity to conduct the correctness proofs significantly.

## 6 Related Work

Credible compilation [20,21] is an approach for certifying compilers similar to the one used in this paper. Credible compilation is aimed at compiler generated proof

[5] experiments conducted on a Sun UltraSPARC III with 900 MHz

[6] a program that computes the first 100, 200 fibonacci numbers resp. and writes them into an array

scripts, too. In contrast to it our approach is based on a general higher-order proof assistant and distinct formalized semantics.

Proof carrying code [14] 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. In [12], 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 appears 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 [11], the algorithms for a sophisticated multi-phase compiler back end are proved correct within the Coq theorem prover. In order 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 [9]. The verification of an optimization algorithm is described in [2]; it uses a simulation proof for showing semantical equivalence. In an important step in the direction of automating the generation of correct program translation procedures is explained in [10]. There, a specification language is described for writing program transformations and their soundness properties. The properties are verified by an automatic theorem prover.

The Verifix project [7,8,4,22] developed and implemented methodologies for correct compilation. Techniques and formalisms for compiler result checkers, decomposition of compilers, and notions of semantical equivalence of source and target program were developed. Verifix uses a combination of algorithm verification and program checking to produce *certified* compilers (this is nicely described in [7]). The main motivation of using program checking in Verifix is that proving the correctness of the checker programs is simpler than proving the correctness of the compilation phase.

In the translation validation approach [18,23] the compiler is regarded as a black box with atmost minor instrumentation. For each 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. If the proof checker says *OK*, both programs are regarded as semantically equivalent. A translation validation approach and implementation for the GNU C compiler is described in [15]. The paper [6] exemplifies that a compiler certificate checker implementation may be much easier to verify than a concrete compiler algorithm (and its implementation).

## 7 Conclusion and Future Work

In this paper we have presented our first experiences with a certifying code generation phase of a compiler. We did extend the code generation phase in such a way that it produces Isabelle/HOL correctness proofs (certificates) for each compiler run. These may be proved correct in the Isabelle/HOL system giving us the guarantee that the compiler has worked correctly. We have shown that a naive generation of the certificates can be a bottleneck in the system because it may take

a lot of time to prove them correct. Hence, we have demonstrated some techniques to speed this proof checking. Therewith we have demonstrated the feasibility of the certifying compilation approach for the code generation phase of a compiler.

A goal for the near future is to optimize the proofs of the remaining compiler phases. We also want to investigate how Isabelle's datatype operations can be made faster. A related area of future work is to investigate the potential advantages of other theorem provers for use as certificate checkers. Future work does not only include the extension of our compiler, but also the application of the certificate checking approach to other areas of software technology.

# References


[1] Andrew W. Appel. Foundational proof-carrying code. In *LICS*, 2001.

[2] Jan Olaf Blech, Lars Gesellensetter, and Sabine Glesner. Formal Verification of Dead Code Elimination in Isabelle/HOL. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods*, pages 200–209. IEEE, IEEE Computer Society Press, September 2005.

[3] Jan Olaf Blech, Sabine Glesner, Johannes Leitner, and Steffen Mülling. A Comparison between two Formal Correctness Proofs in Isabelle/HOL. In *Proceedings of the COCV-Workshop (Compiler Optimization meets Compiler Verification), 8th European Conferences on Theory and Practice of Software (ETAPS 2005)*, pages 33–51. Elsevier, April 2005.

[4] Bettina Buth, Karl-Heinz Buth, Martin Fränzle, Burghard von Karger, Yassine Lakhnech, Hans Langmaack, and Markus Müller-Olm. Provably correct compiler development and implementation. In *CC '92: Proceedings of the 4th International Conference on Compiler Construction*, pages 141–155, London, UK, 1992. Springer-Verlag.

[5] Marek Jezry Gawkowski, Jan Olaf Blech, and Arnd Poetzsch-Heffter. Certifying Compilers based on Formal Translation Contracts. Technical Report 355-06, Technische Universität Kaiserslautern, November 2006.

[6] Sabine Glesner. Using program checking to ensure the correctness of compiler implementations. *Journal of Universal Computer Science (J.UCS)*, 9(3):191–222, March 2003.

[7] Sabine Glesner and Gerhard Goos and Wolf Zimmermann. Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers) it - Information Technology, Issue 5/2005, pages 265–276, May 2004.

[8] Gerhard Goos and Wolf Zimmermann. Verification of compilers. In Bernhard Steffen and Ernst Rüdiger Olderog, editors, *Correct System Design*, volume 1710, pages 201–230. Springer-Verlag, November 1999.

[9] Gerwin Klein and Tobias 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.

[10] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 364–377, New York, NY, USA, 2005. ACM Press.

[11] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54, New York, NY, USA, 2006. ACM Press.

[12] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[13] George C. Necula. Proof-carrying code. ACM Symposium on Principles of Programming Languages and Systems, Paris, France, January 1997.

[14] George C. Necula. *Compiling with Proofs*. PhD thesis, 1998.

[15] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 83–95, 2000.

[16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[17] David A. Patterson and John L. Hennessy. *Computer organization and design (2nd ed.): the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[18] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151+, 1998.

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

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

[21] Martin Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, MIT Laboratory for Computer Science, March 1999.

[22] Wolf Zimmermann On the Correctness of Transformations in Compiler Back-Ends. *Leveraging Applications of Formal Methods, volume 4313 of Lecture Notes in Computer Science*, Springer-Verlag, 2006

[23] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A translation validator for optimizing compilers. In *COCV'02, Compiler Optimization Meets Compiler Verification (Satellite Event of ETAPS 2002)*, volume 65 of *Electronic Notes in Theoretical Computer Science*, pages 1–17, April 2002.