Using Checker Predicates in Certifying Code Generation

Jan Olaf Blech[1]

VERIMAG Laboratory, Université de Grenoble, France

Benjamin Grégoire[2]

INRIA Sophia Antipolis, France

Abstract

Certifying compilation is a way to guarantee the correctness of compiler runs. A certifying compiler generates for each run a proof that it has performed the compilation task 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. Our notion of compilation correctness is based on a human readable formalization of simulation of systems to ensure the same observable behavior of programs. We focus on certifying code generation translating an intermediate language into assembler code. In previous work we found out that the time spent for checking the proofs is the bottleneck of certifying compilation. We introduced the concept of checker predicates. These are formalized in an executable way within a theorem prover to increase the speed of distinct sub tasks of certificate checking. Once the checker predicates are proved correct we are able to use them instead of traditional proving techniques within our theorem prover environment.
In this paper, we present a more elaborate checker predicate for proving the simulation between intermediate and assembler programs correct. In the past this task has turned out to be the most complicated and time consuming task in certificate checking. Our checker predicate uses internally a specially formalized semantics representation of programs which is particularly suited for a fast conduction of proofs in the Coq theorem prover. Using this checker predicate we are able to speed up the task of certificate checking considerably.

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

1 Introduction

Guaranteeing correctness of code generation – apart from optimizations one of the most error prone tasks in a compiler – is a major precondition for correct software. We report on guaranteeing correct code generation using the certifying compilation approach together with checker predicates.

To achieve a trustable code generation we use the *certifying compilers* technique. From a piece of source code certifying compilers emit in addition to the target code a certificate guaranteeing the correctness of distinct compilation runs (cf. Figure 1):

[1] Email: Jan-Olaf.Blech@imag.fr

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

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

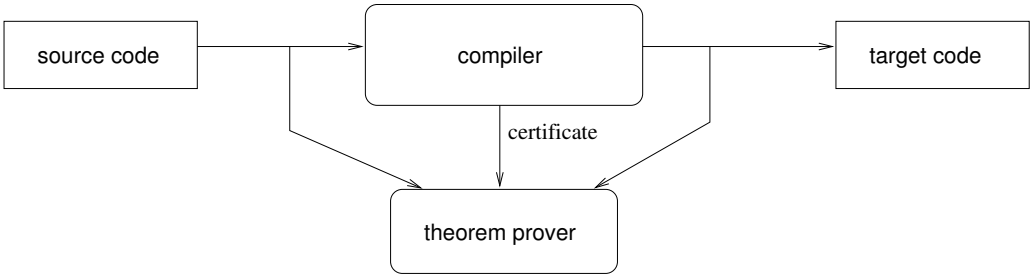

Fig. 1. Certifying Compiler

For each compilation(-phase) run source and target program are passed to a checker – the Coq theorem prover, in our case – together with a certificate – a Coq proof script – to check whether the compilation run has been done correctly. The proof script is automatically generated by the compiler and contains lists of tactic applications and other hints to guide the theorem prover through the checking process.

Certifying compilers have several advantages over *certified compilers*. These are compilers where the compilation algorithm and its implementation have been verified thereby guaranteeing correctness of compilation runs.

- First, the issue of implementation correctness can be completely avoided.
- Second, the technique provides a clear interface between compiler producer and user. No internal details of the compiler have to be given to the compiler for guaranteeing the correctness of a compiler run.
- Third, certifying compilers allow us to abstract from implementation details. This frees us from reverifying the compiler once an aspect of implementation changes slightly.
- Fourth, the generation of certificates is often easier than the verification of a compilation algorithm. We do not have to consider very unlikely input. Thus, we can give up completeness to make the verification task easier.

However, certificates have to be generated and checked for each compilation run.

The time it takes to automatically check certificates – i.e., run proof scripts – in higher order theorem provers can be a bottleneck. Search and rewrite steps triggered by the proof script can be a time consuming task, especially in the context of large tool generated proof scripts. A possible solution is to use theorem provers with less expressive powers which are often faster such as first order theorem provers. However, for some transformations like program transformations a distinct formalized correctness criterion and semantics is a very valuable feature to explain and convince people about the benefits of ones verification work. Such human understandable specifications often require the use of higher order logic, thus higher order theorem provers.

In this paper we further elaborate on checker predicates introduced in [3] and add them to our existing certifying code generation framework [4,3]. Checker predicates are small programs that are formalized within a higher order theorem prover language. They can be used to check certain facts very fast in a higher order theorem prover, much faster than with traditional theorem prover techniques. In our case we use checker predicates with the Coq theorem prover [22]. To profit from

the benefits of higher order specifications we prove our checkers correct with respect to a traditional specification. The checker predicate featured in this paper is characterized via the following key concepts:

- Our checker predicate takes parts of program definitions and the compiled counterparts and decides whether the transformations have been done correctly.
- It uses a special formalized *checker semantics* of the underlying program representation. This semantics formalization is designed for a fast conduction of proofs within a checker predicate.
- Our checker predicate internally computes a list of conditions that could not be proved. This list has to be empty. A checker predicate may easily be modified, such that it does not only return a boolean value, indicating whether the property to be checked does indeed hold, but in case it can not decide whether the property holds, returns this list of unproved conditions (of course strictly speaking we do not have a predicate anymore once this modification is done). After the checker predicate has finished its task this list may be handled with classical theorem proving techniques.
- We prove our checker predicate and the checker semantics correct with respect to our original semantics definition.

Checker predicates can be invoked out of generated proof scripts. It is important to notice that since the checker predicates and their correctness proofs are done in the theorem proving language, the trusted computing base is not enlarged. It comprises the theorem prover, the generation of theorem prover representations of programs (which is small and easy to understand), underlying operating system, and the hardware. The part that generates the proof scripts is not part of the trusted computing base, since false proof scripts can lead to the rejection of a correct compilation, but will never prove a wrong compilation correct.

Guided by certifying code generation as application, this paper focuses on introducing checker predicates as a proof technique for simulation steps in program equivalence proofs in higher-order theorem provers. It presents the steps that have to be carried out in order to use a checker predicate from a certifying compiler writer's point of view who wants to speed up the task of certificate checking. Due to space constraints we omit the presentation of internal details of the Coq theorem prover and their effect on our technique.

Overview

We discuss related work in Section 2. Checker predicates in general and their integration into traditional theorem prover techniques are covered in Section 3. The intermediate and MIPS language used for our simulation correctness checker predicate are introduced in Section 4. It covers both, our original semantics and the special checker semantics representations. Section 5 sketches a general scheme for proving simulation between programs and how to use this to ensure compilation correctness. Section 6 describes our checker predicate for checking simulation of two programs, thereby ensuring their correct compilation. Finally, a conclusion is drawn and future work is presented in Section 7.

2 Related Work

Apart from our own work [4,19,6,3] on certifying compilers the following approaches are most relevant to this paper:

In the translation validation approach [18,1,25] 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 [17]. Like in translation validation we regard correctness for each single compiler run. The analyzer generating the proofs corresponds to the certificate generation in our certifying compiler. 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. A translation validation like certifying compiler framework mapping program representations to an abstract domain for comparison is described in [21].

However, a translation validation checker (called validator) has been formally verified in [23]. Here – like in our work – correctness is based on a formalized semantics. The validator is used for verifying instruction scheduling. It is generated out of a verified Coq specification. Like us they define an additional abstract semantics similar to the approach described in this paper to ease the checking of a program transformation. In contrast to them, our checker predicates are integrated and used within generated proof scripts. This allows for even simpler checker predicates, since in most scenarios it is sufficient to use checker predicates only for the most time consuming tasks within a verification run. We believe that simplicity of checkers and their correctness proof is a necessary precondition to make their use more popular outside traditional theorem prover communities.

Credible compilation [20] 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 an explicitly formalized semantics.

Proof carrying code [16] 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 [2]; it uses an explicit simulation proof scheme for showing semantical

equivalence. 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 [7,8,24] and in the ProCoS project [5]. 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 Checker Predicates

In this section we introduce a framework for checker predicates in general. It does not only cover the checker predicate based on a checker semantics introduced in this paper, but also a checker predicate for injectivity of functions introduced in [3]. We discuss how to handle them inside the theorem prover Coq. The Coq theorem prover processes our certificates. The certificates consist of the following items:

- definitions, e.g., the definition of program representations and
- lemmata stating properties like compilation correctness followed by lists of tactic applications that guide the theorem prover through the process of proving that the stated properties do indeed hold.

While processing the list of tactic applications the theorem prover keeps track of proof goals that still have to be proved correct. Tactic applications either split a proof goal into several other proof goals, solve a proof goal, or rewrite one proof goal by another.

During certificate checking we often arrive at proof goals that require showing that some pieces of data $c_1, ..., c_n$ – e.g., built from constructors of an inductive datatype – fulfill some property D. This means that our theorem prover has to prove a subgoal that looks like:

subgoal:

$$D(c_1, ..., c_n)$$

D is usually a generic property that needs to be fulfilled by a large class of possible problems occurring in compilation run correctness proofs. It is formalized in a declarative way that is relatively easy to understand for humans but hard to decide automatically – it may even be part of the definition of compilation correctness. The $c_1, ..., c_n$ are different for each compilation run. One way to prove this lemma correct is to generate and run a script consisting of some tactic applications. The faster way is to use a checker predicate, which computes the result. The underlying technique for these proofs via computations in Coq is also called proof by reflection (cf. [9,10] on this).

In our case we may formalize a checker predicate E in an executable way that checks whether the property formalized by D holds. Its correctness is guaranteed

by proving the following lemma once and for all – thus this proof does not have to be redone during certificate checking:

lemma:
$$\forall x_1, ..., x_n. E(x_1, ..., x_n) \longrightarrow D(x_1, ..., x_n)$$

This correctness lemma can be applied via a classical implication rule each time a subgoal as sketched above appears within a proof. It will transform the sketched subgoal into another subgoal:

subgoal:
$$E(c_1, ..., c_n)$$

We can now use the executable formalized checker predicate instead of the declarative correctness notion in our proof script to solve this subgoal, i.e., resolve it to true. Thus, we have replaced a deductive proof by computation.

Note, that since this method is entirely realized within the description language of the theorem prover, it does not enlarge the trusted computing base.

An Example Checker Predicate

A very simple example is a checker predicate for checking that a natural number is divisible by two. The human readable specification might define this property as:

$$D(x) = \exists k.\ x = 2 \cdot k \qquad \text{k, x are natural numbers}$$

This is not executable. A checker predicate for the same property might be formalized in an executable way:

$$E(0) = True$$
$$E(1) = False$$
$$E(x) = E(x-2) \qquad x \geq 2$$

One might also think of a checker predicate that internally transforms the natural number into a binary representation and checks the last digit. Such a transformation of the input data is typical for our checker predicates.

4 The IL and MIPS Language

In this section we sketch the syntax and semantics of our intermediate language (IL) and MIPS language [3]. 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. In addition to the syntax and semantics used in our program and correctness definitions, we present checker semantics of IL and MIPS.

[3] The acronym MIPS originally stands for "Microprocessor without Interlocked Pipeline Stages".

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

loperand ::=
   LVAR var | LLOCVAR var |
   LARRAYC (var × val) | LARRAYV (var × var)

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

Fig. 2. Intermediate Language Syntax (excerpt)

```
(termstate : flag, output : list val,

    varvals : (var × val) ⇒ val, locvarsstack : list ((var ⇒ val) × loc), pc : loc)
```

Fig. 3. Intermediate Language States (Signature)

4.1 The Intermediate Language

An excerpt of the definition of the intermediate language's syntax (cf. [3]) 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 of type *var*. *val* denotes a generic type for values, *loc* denotes locations – positions in the list of statements.

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.

4.2 The MIPS Language

Our formalized set of MIPS instructions (cf. [3]) comprises basic arithmetic operations, shift operations, and branch instructions. In addition instructions for basic output, procedure calls and returns from a procedure are provided. 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

```
(tltermstate : flag, tloutput : list val, regs : reg ⇒ val,mem : val ⇒ val, tlpc : loc)
```

Fig. 4. MIPS Code State Definition

```
absilstate =
    (absval alist) × pid × absvarstore × ((pid × loc × var × absvarstore) alist) × absloc
```

Fig. 5. Abstract IL States

```
MIPSstate =
        (absval' alist) × absregstore × absmemstore × absloc
```

Fig. 6. Abstract MIPS State Definition

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 (type *reg*) and memory to value mappings. A program counter (type *loc*) 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*.

4.3 Checker Semantics for IL and MIPS

The semantics definition for IL and MIPS as introduced above is used in our specification of compilation correctness. We refer to these definitions as the *original semantics*. In previous versions of our certifying code generation [4,3] we directly proved simulation of programs on it. The checker predicate described in this paper, however, uses special checker semantics: semantics definitions for the involved languages that are defined solely for them to compare and reason about programs. Compared to the original semantics the state definitions and the transition rules are changed. Program syntax representation is not changed. Our checker predicate uses these semantics to compute succeeding states rather than derive them from declarative rules. Compared to the original semantics, values are represented in an abstract way as abstract datatypes consisting of constructors representing read and write accesses and arithmetic operations. This makes comparison between state representations easier.

The biggest difference to the original semantics is that the definition of states changes as shown in Figures 5 and 6. We use abstract store and value definitions as shown in Figure 7 for IL and in Figure 8 for MIPS. An abstract store represents variables' values (absvarstore), memory (absmemstore) and register set (absregstore) as terms. It consists of an initial constructor INIT, or an update of a value at a primitive or array variable in the IL or an address or register in the MIPS language of an abstract store. An abstract value (absval, absval') can be a constant, an operator combining abstract values, or some read access to an abstract store. alist denotes an abstract list, that in addition to standard NIL and CONS constructors features an abstract constructor for an arbitrary list. In addition to this, we have abstract types for program counters (cf. definition of absloc) to indicate program counters that depend upon the evaluation of an abstract value. This is needed

```
absval::=
        ABSCONST val | ABSPLUS absval absval | ABSMINUS absval absval | ABSMULT absval absval |
        ABSLT absval absval | ABSLE absval absval |
        VARGET var absvarstore | ACGET var absval absvarstore |
        AVGET var var absvarstore

absvarstore ::=
        INIT | VARUPD var absval absvarstore | ACUPD var absval absval absvarstore |
        AVUPD var var absval absvarstore

absloc ::=

        LOCKNOWN loc | LOCUNKNOWN absval loc loc
```

Fig. 7. Abstract Store and Value Definition for IL

```
absval' ::=
        ABSCONST val | ABSPLUS absval' absval' | ABSMINUS absval' absval' | ABSMULT absval' absval' |
        ABSLT absval' absval' | ABSLE absval' absval' |
        REGGET reg absregstore | MEMGET absval' absmemstore

absregstore ::=
        INIT | REGUPD reg absval' absregstore

absmemstore ::=
        INIT | MEMUPD absval' absval' absmemstore
```

Fig. 8. Abstract Store and Value Definition for MIPS

for conditional expressions. Comparing memory accesses from an original and a transformed program are more easily and fast done automatically when using such a definition.

Based on these definitions, the state transition functions for the checker semantics ilnext^C and $\mathsf{tlnextn}^C$ compute checker semantics succeeding states.

4.4 Example: An Array Assignment

Consider Figure 9. It presents an array assignment evaluated by the checker semantics. Both states in intermediate language and MIPS code have unspecified initial variables' values, memory, and register set. This INIT value corresponds to an universally quantified store in the original semantics. The succeeding states consist of terms indicating the computations done during the step transition. Our checker predicate compares these terms from both intermediate and MIPS code. They are syntactically very similar: Both consist of an assignment to some kind of store. The value assigned is represented in both cases as a term consisting of an ABSPLUS constructor with comparable arguments. The checker predicate presented in this paper is defined on such terms. All it has to derive is the look-up that the variable 'x' corresponds to the memory location ABSCONST 1000 and 'y' corresponds to ABSCONST 1004. The checker predicate can do the whole checking process without the need for a unification at any point. Moreover, the checker predicate may need to rewrite the terms serving as its arguments but it never needs to rewrite any proof goals within the theorem prover logic.

Intermediate Language Code

ASSIGN_V 'x' (PLUS (VAR 'y') (CONST 1))

MIPS Code

LOAD a10 1004 0

ADDI a10 a10 1

STORE a10 1000 0

Intermediate Language's and MIPS States before the Step

(ILoutp,pid,INIT,stack,100)

(MIPSoutp,INIT,INIT,400)

Intermediate Language's and MIPS States after the Step

(ILoutp,pid,

VARUPD x (ABSPLUS (VARGET y INIT) (ABSCONST 1)) INIT,stack,LOCKNOWN 101)

(MIPSoutp, ...,

MEMUPD (ABSCONST 1000) (ABSPLUS (MEMGET (ABSCONST 1004) INIT)

(ABSCONST 1)) INIT, LOCKNOWN 403)

Fig. 9. An Assignment to a Variable Using the Checker Semantics

Lemma ilabstract_correct :

$\forall$ gvals lvals output . $\forall$ P as s as' s'.

interp_ilstate (gvals,lvals,output,as) = Some s $\Longrightarrow$

ilnext (P,s) = s' $\Longrightarrow$

ilnext^C (P,as) = Some as' $\Longrightarrow$

interp_ilstate (gvals,lvals,output,as') = Some s'

Fig. 10. Correctness of Checker Semantics (IL)

4.5 Correctness of the Checker Semantics

Correctness of checker semantics is stated with respect to the original semantics definitions. The correctness of the checker intermediate language semantics is formulated via the lemma shown in Figure 10. Assuming that an IL checker semantics state as has an original semantics representation s we have to derive that the succeeding abstract state as' when interpreted in an original semantics state representation equals the succeeding original semantics state s'. Note, that the interpretation function for intermediate checker semantics states interp_ilstate is defined to return an option datatype. This is especially relevant since some state transitions in the checker semantics may not be performed if the original state has an unknown program counter. In these cases None is returned. gvals, lvals and output are universally quantified variables used for interpreting a checker semantics state. They provide arbitrary interpretation values for the INIT constructors in stores.

Figure 11 shows the lemma stating correctness of the checker MIPS semantics. P is a piece of MIPS code. The lemma makes use of the interp_tlstate function for interpreting abstract MIPS states. mvals, rvals, and output provide arbitrary values of memory, register set and output to the interpretation function. It is very similar to the correctness lemma for the intermediate language. The only major difference is the extra n argument since we deal with state transition functions tlnextn and tlnextn^C that perform n consecutive steps.

We have proved the correctness of our checker semantics representations in Coq.

```
Lemma tlabstract_correct :
∀ mvals rvals output . ∀ P as s as' s' n.
    interp_tlstate (mvals,rvals,output,as) = Some s ⟹
    tlnextn (P,s,n) = s' ⟹
    tlnextn^C (P,as,n) = Some as' ⟹
    interp_tlstate (mvals,rvals,output,as') = Some s'
```

Fig. 11. Correctness of Checker Semantics (MIPS)

5 Correctness of Code Generation via Simulation

In this section we present our simulation based notion of compilation correctness and discuss how the certificate guides through the process of proving compilation runs automatically correct in a theorem prover (cf. [4,3]).

To verify that a transformation has been conducted correctly one needs to formalize a notion of correctness. The original and transformed programs shall semantical correspond to each other. We regard two programs as semantically corresponding if they behave the same. This means they generate the same output values in the same order under the same input values. For simplicity reasons, we regard inputs as fixed in this paper. 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.

Formally we require the intermediate language program and the MIPS program to be in a (weak) simulation. The corresponding simulation relation – comparing an intermediate language state with a MIPS state – has to ensure the equivalence of output values of related states:

- The simulation relation has to hold for the initial states.
- For two intermediate and MIPS states in the relation, if there is a next intermediate operation, there has to be one or more MIPS instructions such that the succeeding states are in the simulation relation again.

In addition to equivalence of outputs it is convenient to require further constraints within the simulation relation like:

- IL variables' values have to be mapped to MIPS memory locations as specified by a distinct mapping relation: the *variable mapping*,
- program counters in IL and MIPS have to correspond to each other as indicated by a *program counter relation*,
- 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.

Proving Simulation

The first task to verify the correctness of a code generation run is establishing a simulation relation for the concrete code generation run based on information provided by the compiler. We prove that it indeed implies correctness, i.e., it ensures the same output traces (cf. first two items of the code generation correctness criterion). Next we prove that the initial states of both programs are in the simulation relation.

For showing that for each two states in the simulation relation the succeeding states are in the relation again, we make a case distinction on possible locations in the intermediate language code (program counter). An intermediate language state being in a simulation relation with some MIPS state requires that it must point to some intermediate language statement. In addition, 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 to the intermediate language instruction. We make a case distinction on all possible intermediate language program points. Hence we split intermediate language and MIPS code into corresponding *program slices* of IL statements and MIPS instructions which have to semantically correspond to each other (cf. Figure 9 for an example of a pair of slices). For each corresponding pair of slices we prove in the theorem prover 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 of a pair of program slices correct we require a number of prerequisites. Various properties concerning the mapping from variables to memory have to be ensured. Crucial to our proofs is the fact that the mapping between variables and memory is injective: If we change a variable and a corresponding memory cell no other variable's memory cell is affected.

The case distinction on the intermediate languages program points realizing the program slices is implemented by different lemmata: one for each program point/pair of slices. We call such a lemma a *step lemma*. Finally, it is all put together in a last phase proving the compilation correctness for an intermediate and a MIPS program.

Proving step lemmata correct 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. Compared to the original semantics, where undefined parts of states have to be specified as free variables, our checker semantics can represent them using abstract datatypes without using free variables.

6 A Checker Predicate for Simulation Steps

In this section we present our checker predicate for proving simulation steps correct. The most important part of proving compiler transformations correct with respect to a simulation based notion of correctness is the proof of the simulation steps. We can not explicitly investigate all possible steps, since there may be infinite many ones in case of non-terminating programs. For this reason we look at distinct classes of steps and prove that all steps in the class respect the simulation requirements. Typically a class of steps is defined by a pair of program slices (cf. Section 5) encapsulating all steps that may be performed from a given program point. By looking at all program points we achieve full coverage.

For verifying a concrete compiler transformation a checker predicate takes an original program, the transformed program, the simulation relation, and a pair of program points from the original and transformed program. It decides whether the instructions involved in the step have been transformed correctly. Once we have checked every class of simulation steps and proved the inclusion of the initial states in the simulation relation we can conclude the correctness of the whole program transformation.

To verify that a pair of program slices realizing a step is correct the checker predicate performs the following tasks:

- It takes the symbolic representations of the states before the steps and computes the symbolic representations for the states after the step.
- It compares the resulting symbolic state representations and checks whether they are in the simulation relation.

The checker-semantics allows for very fast checks of step correctness. However, our approach requires us to prove the following items:

- We have to prove the checker-semantics correct with respect to the original semantics – which is usually defined in a more declarative way.
- We have to prove the checker predicate correct. This means, it has to ensure the requirements of the simulation relation.

Once this is done we gain a very fast and flexible tool that guarantees the correctness of simulation steps.

The lemma stating the correctness of the checker is not only used to convince users that the checker may be trusted. We use it within Coq to rewrite proof goals demanding the original, more declarative notion of correctness into proof goals demanding the application of a checker. Once this is done we use Coq's *vm_compute* tactic to execute the checker and prove the proof goal.

6.1 Definition of the Simulation Step Checker Predicate

The following describes the implementation of our step correctness checker predicate checker shown in Figure 12. It uses a helper function checker_step. This helper function computes a list of conditions that need to be true in order to regard a symbolic execution step as correct.

First based on given program counters underspecified initial states are gener-

```
checker_step (MemMap,Vars,PCRel,P_IL,P_MIPS,pc_IL,pc_MIPS) =
    let ais := mk_il_rawstate (pc_IL) in
    let ats := mk_tl_rawstate (pc_MIPS) in
    match ilnext^C (P_IL,ais) with
        | Some (aoutp,pid,agvs,(rpid,raddr,rvar,alvs)::stack,apc) =>
        match tlnextn^C (P_MIPS,ats,n) with
            | Some (tlaoutp,aregs,amem,tlapc) =>
                compmem (MemMap,Vars,agvs,alvs,amem,aregs) ++
                comppc (MemMap,Vars,PCRel,apc,tlapc) ++
                compoutp (MemMap,Vars,aoutp,tloutp)
            | None => false :: nil
        end
        | None => false::nil
    end.

checker(MemMap,Vars,PCRel,P_IL,P_MIPS,pc_IL,pc_MIPS) ≡
    checker_step(MemMap,Vars,PCRel,P_IL,P_MIPS,pc_IL,pc_MIPS) = []
```

Fig. 12. The checker Checker Predicate

ated using the functions mk_il_rawstate and mk_tl_rawstate. They represent states where variables and memory can contain any value, as long as IL variables and MIPS memory fulfill the simulation relation properties. Using these states the succeeding abstract state representations are computed. These are compared using the functions compmem, comppc, and compoutp. The function compmem compares abstract store representations for global and local variables, registers and memory. Comparing program counters is done by comppc. Output lists are handled by compoutp. These functions are defined inductively on the term structure of the terms representing abstract stores and program counters. compoutput only regards the last value in the list – if any is computed. Thereby a list of conditions that these functions cannot prove on their own is generated (concatenated by ++). Usually this list is empty. checker only checks whether the list is empty or not. If an error occurs a constant false is added to the list of conditions that need to be verified.

6.2 Checker Predicate Correctness

Figure 13 shows our correctness lemma for our checker predicate: checker. For all possible values of the compiler provided information: MemMap, Vars, PCRel, an intermediate language program P_{IL} and MIPS P_{MIPS}, as well as two program counters from the states s_{IL}, s_{MIPS} and the number of steps taken in the MIPS language n the following correctness property has to hold: if the checker is content, some additional properties that do not belong to a step lemma are fulfilled (e.g., injectivity, alignment [3]), and the state correspondence captured in the simulation relation H as created by createsimulation holds for the two states before the step, than the state correspondence holds after the step, too. The correctness proof is done by unfolding the checker definition. The main step of the proof is the rewriting of the computation and comparison of the checker semantics states by their concrete counterparts within the checker predicate. This is done by using the checker semantics correctness proofs.

6.3 Using the Correctness Proof

The first step in proving simulation of programs as required by our notion of code generation correctness consists in a case distinction of possible program

Lemma :

∀ MemMap Vars PCRel P_{IL} P_{MIPS} s_{IL} s_{MIPS} n.
checker (MemMap,Vars,PCRel,P_{IL},P_{MIPS},get_pc (s_{IL}),get_pc' (s_{MIPS}),n) ⟹
"*additional properties on* MemMap, Vars, P_{IL}, and P_{MIPS}" ⟹
"H *ensures output equivalence*" ⟹
H (s_{IL},s_{MIPS}) ⟹
H (ilnext (P_{IL},s_{IL}),tlnextn (P_{MIPS},s_{MIPS},n))

Fig. 13. Correctness of the checker Checker Predicate

counters. Thus, we have to prove subgoals realizing step lemmata of the following form (one for each possible program counter in the intermediate language and its corresponding MIPS counterpart):

H (s_{IL},s_{MIPS}) ⟹
H (ilnext (P_{IL},s_{IL}),tlnextn (P_{MIPS},s_{MIPS},n))

We can apply our checker predicate correctness proof to this subgoal via an implication rule. This results in the following subgoals:

(i) checker (MemMap,Vars,PCRel,P_{IL},P_{MIPS},get_pc (s_{IL}),get_pc' (s_{MIPS}),n)

(ii) "*additional properties on* MemMap, Vars, P_{IL}, and P_{MIPS}"

(iii) "H *ensures output equivalence*"

The first goal can be computed in a fast way by evaluating the checker predicate with the Coq tactic *vm_compute*. The properties of the variable mapping are proved once and for all for a program. The last subgoal is trivially true, since we use a scheme for constructing H that automatically ensures output equivalence.

6.4 Evaluation

We have implemented and proved the presented checker predicate correct. It checks the correctness of simulation steps appearing in concrete programs. The checker predicate is realized and proved correct within Coq. Based on our original semantics for intermediate language and MIPS code we have established checker semantics for both languages. Some parts in our specification are generic to both semantics formalisms. This made the process of proving the checker semantics correct with respect to the original semantics more easy. The checker predicate itself consists of less than 100 lines of Coq specifications. Most parts of it do transformations on the involved abstract store representations to make intermediate language and MIPS states comparable. The correctness proof of the checker contains auxiliary lemmata and definitions. It consists of a few hundred lines of Coq proof scripts written in a compact form. Most of the verification work is spent with proving the transformations conducted in the checker correct. Our checker predicate can verify programs, consisting of various arithmetic operations, array assignments, and conditional branches. So far we did not integrate function calls and returns into our checker. However, most parts of programs using function call and returns can be handled by our checker predicate, too. Only the step lemmata encapsulating

function call and returns must be verified using our traditional proof script generation based technique [3]. Our checker predicate can verify programs and their transformations consisting of several hundred lines of intermediate language code within a few seconds – thus, provide a solution for the biggest bottleneck in certifying code generation [3]. The verification of code generation for several thousands lines of intermediate language code, can be done within several minutes. The verification time grows non-linear for larger programs, due to an artifact in the Coq implementation.

7 Conclusion and Future Work

We have defined the notion of checker predicates and presented a general way to use them to speed the process of proving Coq scripts up. We presented a checker predicate for checking that simulation steps in a compilation run correctness proof hold. It uses a special checker semantics to conduct computations in a fast way. We proved our checker predicate correct with respect to the original correctness definition of simulation steps. The correctness proof – which is done once and for all and is reused in each certificate checking – is entirely carried out in Coq. Our checker predicate handles the most complicated and time consuming parts of certified code generation. Thus, it speeds the automatic verification of certificates considerably up and simplifies the certificate generation part of the compiler.

Currently we are focussing on certificate generation for the results of verification tools like model checkers. For this application, we are developing a checker predicate used for checking invariants of an asynchronous, component-based language based on transition systems in the Coq theorem prover. The challenges occurring here are very similar to compiler code generation: We use a specialized checker semantics and reason about transition steps of the involved systems. A related long term goal is a certifying code generation for this language.

Apart from this, we believe, that checker predicates can be useful for all phases of certifying compilation and proof carrying code scenarios, where a higher-order theorem prover is used as a proof checker.

References

[1] 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-Verlag, 2005.

[2] J. O. Blech, L. Gesellensetter, and S. 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] J. O. Blech and B. Grégoire. Certifying code generation with coq. In *Proceedings of the 7th Workshop on Compiler Optimization meets Compiler Verification (COCV 2008), Budapest, Hungary*, ENTCS. April 2008.

[4] J. O. Blech and A. Poetzsch-Heffter. A certifying code generation phase. In *Proceedings of the 6th Workshop on Compiler Optimization meets Compiler Verification (COCV 2007), Braga, Portugal*, ENTCS, March 2007.

[5] B. Buth, K-H. Buth, M. Fränzle, B. von Karger, Y. Lakhnech, H. Langmaack, and M. Müller-Olm. Provably correct compiler development and implementation. In *CC '92: Proceedings of the 4th International Conference on Compiler Construction*, pages 141–155, London, UK, 1992. Springer-Verlag.

[6] 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.

[7] 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.

[8] G. Goos and W. Zimmermann. Verification of compilers. In Bernhard Steffen and Ernst Rdiger Olderog, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 201–230. Springer-Verlag, November 1999.

[9] B. Grégoire, L. Théry, and B. Werner. A computational approach to pocklington certificates in type theory. In Proc. of Functional and Logic Programming, 8th International Symposium,LNCS, Springer-Verlag, 2006.

[10] B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, LNCS, Springer-Verlag, 2005.

[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 *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.

[14] X. 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.

[15] 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.

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

[17] G. 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.

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

[19] A. Poetzsch-Heffter and M. 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] X. Rival. Symbolic Transfer Functions-based Approaches to Certified Compilation. In 31st Symposium on Principles of Programming Languages (POPL'2004), Venice, Jan. 2004 ACM.

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

[23] 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.

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

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