Certifying Compilers based on Formal Translation Contracts

Marek Jerzy Gawkowski, Jan Olaf Blech, Arnd Poetzsch-Heffter

Internal Report
No. 355/06

November 2006

Computer Science Department
University of Kaiserslautern

{gawkowsk, blech, poetzsch}@informatik.uni-kl.de

Abstract. A translation contract is a binary predicate $corrTransl(S, T)$ for source programs S and target programs T. It precisely specifies when T is considered to be a correct translation of S. A certifying compiler generates –in addittion to the target T– a proof for $corrTransl(S, T)$. Certifying compilers are important for the development of safety critical systems to establish the behavioral equivalence of high-level programs with their compiled assembler code. In this paper, we report on a certifying compiler, its proof techniques, and the underlying formal framework developed within the proof assistent Isabelle/HOL. The compiler uses a tiny C-like language as input, has an optimization phase, and generates MIPS code. The underlying translation contract is based on a trace semantics. We investigate design alternatives and discuss our experiences.

1 Introduction

The compiler is a crucial part in the development of software systems. Most software systems are described in high-level model or programming languages. Their runtime behavior, however, is controled by the compiled code. For uncritical software it might be sufficient to just test the runtime behavior of the code. If an error is detected, it might be caused by the programmer, by the compiler, or by a semantical ambiguity[1]. For critical software, 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 translation. Thus, translation correctness is essential to close the formalization chain from high-level formal methods to the machine-code level.

Since more than thirty years researchers have worked on the problem of translation correctness (see Sect. 2 for a review of related work). We can distinguish two general approaches to establish the correctness of a translation[2]:

- *Certified compiler*: Prove (a) that the algorithms of the compiler define a correct translation for all given well-formed input programs (*compiler algorithm correctness*) and (b) that the algorithms are correctly implemented on a given machine (*compiler implementation correctness*). We call a compiler for which machine checked proofs for both parts are developed a *certified compiler (algorithm/implementation)*.
- *Certifying compiler*: Provide a proof 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 automatically generate such proofs (see Sect. 2). If the compiler generates — in addition to the target program T — a machine-checkable proof that T correctly implements its source program, we call it a *certifying compiler* and the generated proof a *translation certificate*.

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. [17] on this problem). Second, similar to the proof carrying code approach ([12, 11, 1]), the technique provide 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

[1] E.g. the programmer might assume a particular evaluation order of expressions that is not realized by the used compiler.

[2] We follow the notions given in [9] and slightly refine them based on a discussion at the Dagstuhl Seminar 05311 "Verifying Optimizing Compilers".

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.

In the last two years, we constructed an optimizing certifying compiler that generates proof scripts as certificates. More precisely, *our approach* is characterized by the following three aspects (cf. [17]):

1. 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 as our specification and verification framework.
2. Translation contract: We require an explicit *translation contract* formally specifying the semantics of source and target language and the translation correctness predicate $corrTransl(S, T)$ expressing the fact that T is a correct translation of S.
3. Certifying compiler: We are interested in a technique where the compiler generates proof scripts as checkable certificates.

Machine-checkability is advisable because of the complexity and size of the proof tasks. Using a logical framework that is not specifically developed for the translation task and used in many other areas, increases the confidence in the framework. Of course, as argued in [17], a framework in which only a very small core has to be trusted is desirable. An explicit translation contract plays the role of the specification of the proof task. It is the contract between producer and client of the compiler and should thus be available to and comprehensible for the client. In particular, it can and should be independent of the structure and algorithms of the compilers satisfying the contract.

We developed our certifying compiler to gain experience with the described approach and to create a testbed for the validation of different techniques to generate machine-checkable certificates. The techniques can differ in the needed efforts to instrument the compiler for certificate generation, in the structure and size of the certificate, and the efficiency of checking certificates. The main technical contributions of this paper are:

- Techniques for structuring the certification into program dependent and independent parts.
- The application of the approach to trace-based translation contracts.
- A refined technique to automate contract verification.
- Methods to combine proof techniques to construct certifying compilers.
- First experimental results, experiences, and technical propositions on how to run proofs more efficiently. (To the best of our knowledge, we are the first who implemented this approach and gained practical experience withit.)

As in this section, S denotes a source program and T a target program throughout this paper.

Overview. After the discussion of related work in Sect. 2, we explain translation contracts and specify the contract for our compiler (Sect. 3). Sect. 4 presents our proof techniques and describes how certificates are generated. Sect. 5 shows how other proof techniques could be integrated into our approach. Sect. 6 describes our experimental results and techniques to make certificate checking more efficient. Sect. 7 contains the conclusions.

2 Related Work

Rinard et al. present in [18, 19] the credible compilation approach for certifying compilers. In particular, they provide dedicated proof rules to verify program invariants, even for programs with pointers. Our work builds on their approach and extends it by the notion of an explicit translation contract. Other distinctions are that we looked at a semantics based on output streams and that central part of our contribution is the implementation of the approach based on a general higher-order proof assistant.

Proof carrying code [12] is a framework for guaranteeing that certain requirements or properties of a compiled program are met, e.g. type safety or the abscence of stack overflows. That is, the carried proof certifies a property only depending on `T` whereas we are interested in a property depending on `S` and `T`. In [10], Necula and Lee described a certifying compiler for their approach guaranteeing that target programs are type and memory safe. What is related to our work, is the clear separation between the compilation infrastructure and the checkable ceritificate. That is why many techniques developed for proof carrying code apply as well to our approach (e.g. [1]).

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 [9], the algorithms for a sophiticated 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 [7]. 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 [8]. There, a specification language is described for writing program transformations and their soundness properties. The properties are verified by an automatic theorem prover.

In the translation validation approach [16, 20] 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 [13]. The paper [5] examplifies that a compiler certificate checker implementation may be much easier to verify than a concrete compiler algorithm (and its implementation). The Verifix project [6, 3] had the goal to achieve correct

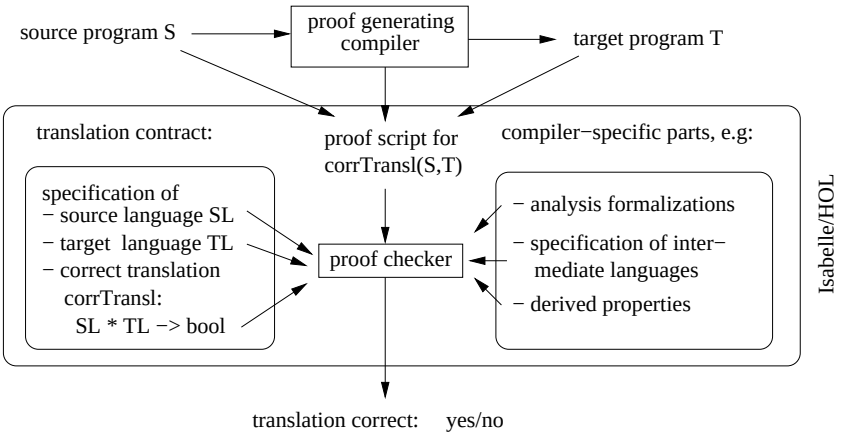

Fig. 1. Certifying compiler infrastructurecertifying compiler infrastructure

compilation, too. Techniques and formalisms for compiler result checkers, decomposition of compilers, and notions of semantical equivalence of source and target program were developed.

3 Translation Contract and Checking Infrastructure

In this section, we describe translation contracts, the checking infrastructure, and explain how it is realized for our compiler. Figure 1 gives an overview of the certifying compiler infrastructure for our approach. Specification and verification is done in a general proof assistant for higher-order logic. We use Isabelle/HOL [14] for this purpose. The specifications are divided into the compiler-independent translation contract (shown on the left of Fig. 1) and definitions and properties that are compiler-specific, but program independent (shown on the right of Fig. 1). Typically, the compiler-specific part contains definitions and properties of program analyses and intermediate languages. Given S, the certifying compiler generates T and a proof script as certificate. Running this proof script by the proof checker establishes the correctness of $corrTransl(S, T)$. In our current setting, source and target programs are given as abstract syntax trees. Parsing of concrete syntax is not considered so far. The proof must include proofs for all compiler-specific properties that are used, that is, it builds only on the translation contract. In the following, we describe the translation contract for our compiler. The compiler-specific definitions and properties are presented in Sect. 4.

Our compiler reads programs written in a small C subset, translates them into a control flow graph (FGL), performs constant folding (CF), dead assignment elimination (DAE) and loop invariant hoisting (LIH) on the flow graph, and finally generates MIPS code (CG) (see [15]). The current implementation of project only covers optimization and code generation. We considered these

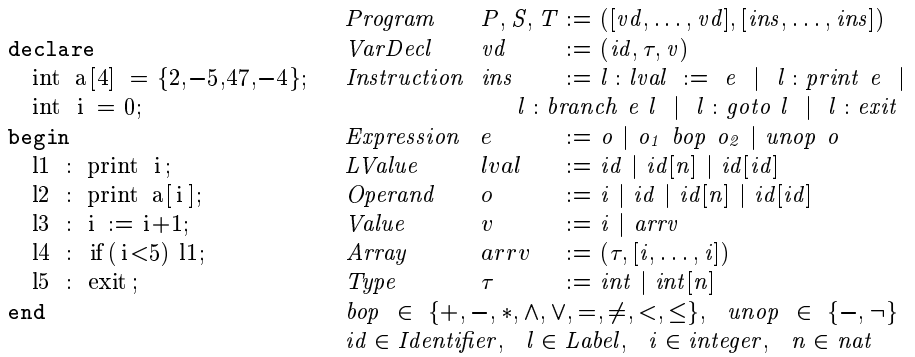

Fig. 2. Example and syntax of language FGL

phases first, because they are more challenging from a verification point of view. Consequently, the translation contract specifies the flow graph language *FGL*, the used MIPS subset *MSub* and the translation correctness predicate *corrTransl*.

Source Language. Our source language is the flow graph language FGL supporting variables of primitive types, arrays, simple assignments, a print statement to output an integer, conditional and unconditional braches, and an exit statement. Figure 2 presents a simple program example and the definition of the abstract syntax[3]. As we are interested in the compilation of both terminating and nonterminating programs[4], we use a semantics based on sequences of outputs produced by the print instructions. (Similarly, we could handle reads.) More precisely, the semantics of a program is denoted by a pair (s, o): s captures the termination *Status*: Token NORMAL indicates normal termination, ABRUPT abrupt termination, NONTERM nontermination. The second component o is a possibly infinite sequence of integers. Infinite sequences are modeled as functions $o : nat \rightarrow integer \cup \{\mathsf{undef}\}$ where o is either defined for all elements of nat or for all $k \in [0 \ldots n]$, $n \in nat$. We call nat or $[0 \ldots n]$ the domain of o, denoted by $\mathsf{dom}(o)$. The k-th output of the program is $o(k)$. The type of the output functions is named *Output*.

The interesting parts of the operational semantics of FGL are given in Fig. 3. The main technical difficulty is to handle nontermination and infinite output. We use the number of execution steps to inductively define the semantics as follows. A program configuration consists of the label of the current instruction, the state of the variables, the status of the execution, and the output produced so far. Each component of the configuration has its corresponding selector function.

If the current termination status is NORMAL or ABRUPT, $step_{FGL}$ does not change the configuration; otherwise it executes the instruction at the given label and yields the resulting configuration. As it is standard, we dispense with

[3] To keep the presentation short, we slightly simplified our language for this paper.
[4] Nonterminating programs occur for example in controler software.

$$
\begin{array}{l}
\mathit{Status} \quad = \{\mathsf{NORMAL}, \mathsf{ABRUPT}, \mathsf{NONTERM}\} \\
\mathit{Configuration} = \mathit{Label} \times \mathit{State} \times \mathit{Status} \times \mathit{Output} \\
\mathit{step}_{FGL} : \mathit{Program} \times \mathit{Configuration} \longrightarrow \mathit{Configuration}
\end{array}
$$

$$
\begin{array}{l}
\mathit{nstep}_{FGL} : \mathit{nat} \times \mathit{Program} \times \mathit{Configuration} \longrightarrow \mathit{Configuration} \\
\mathit{nstep}_{FGL}(0, P, C) \quad = C \\
\mathit{nstep}_{FGL}(n+1, P, C) = \mathit{nstep}_{FGL}(n, \mathit{step}_{FGL}(P, C))
\end{array}
$$

$$
\begin{array}{l}
\mathit{run}_{FGL} : \mathit{nat} \times \mathit{Program} \longrightarrow (\mathit{Status} \times \mathit{Output}) \\
\mathit{run}_{FGL}(n, P) = \texttt{if } \neg \mathit{wellFormed}(P) \texttt{ then } (\mathsf{ABRUPT}, \lambda\, n.\, \mathsf{undef}) \texttt{ else} \\
\quad \texttt{let}(l, \sigma, s, o) = \mathit{nstep}_{FGL}(n, P, (l_0, \mathit{init}_{FGL}(P), \mathsf{NONTERM}, \lambda\, n.\, \mathsf{undef})) \texttt{ in } (s, o)
\end{array}
$$

$$
\begin{array}{l}
\mathit{sem}_{FGL} : \mathit{Program} \longrightarrow (\mathit{Status} \times \mathit{Output}) \\
\mathit{sem}_{FGL}(P) = \texttt{if } \exists\, n.\, \mathit{run}_{FGL}(n, P) == (\mathsf{NORMAL}, o) \texttt{ then } (\mathsf{NORMAL}, o) \\
\qquad\qquad \texttt{elseif } \exists\, n.\, \mathit{run}_{FGL}(n, P) == (\mathsf{ABRUPT}, o) \texttt{ then } (\mathsf{ABRUPT}, o) \\
\qquad\qquad \texttt{else } \quad (\mathsf{NONTERM}, \lambda\, k.\, \texttt{if } \exists\, n.\, k \in \mathsf{dom}(\mathit{output}(\mathit{run}_{FGL}(n, P))) \\
\qquad\qquad\qquad\qquad\qquad \texttt{then choose } n.\, k \in \mathsf{dom}(\mathit{output}(\mathit{run}_{FGL}(n, P))) \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{in } \mathit{output}(\mathit{run}_{FGL}(n, P))(k) \\
\qquad\qquad\qquad\qquad\qquad \texttt{else undef})
\end{array}
$$

Fig. 3. Semantics of language FGL

the formal definition of step_{FGL}. Function nstep_{FGL} performs n steps. Function run_{FGL} checks whether a program P is well-formed, runs P for n steps with initial state $\mathit{init}_{FGL}(P)$ that is extracted from the variable declaration, and selects the result from the final configuration. The semantic function sem_{FGL} expresses the overall behavior of a program P. If P terminates after n steps, sem_{FGL} yields the corresponding result. Otherwise, the k-th output is obtained by looking for a number n of program steps that produce at least k outputs. If such an n exists, let the program run n steps and take the k-th outputs. Otherwise the output function remains undefined for k.

Target Language. As target language, we use *MSub*, a subset of the MIPS assembler [15]. An MSub program is a list of MIPS instructions. In particular, we support the following instructions: integer addition, subtraction, multiplication, the compare operation "set less than", conditional and unconditional branches, store and load instructions, system call instructions for output and return. Abrupt termination is indicated by setting a dedicated flag and calling return. The formalization is very similar to that of FGL. The main difference is that MSub uses registers and addressable memory as storage. The functions step_{MSub}, nstep_{MSub}, run_{MSub}, and sem_{MSub} are almost defined as in Fig. 2.

Translation Correctness Predicate. The correctness predicate $\mathit{corrTransl}(S, T)$ defines when T is considered to be a correct translation of S. It should be independent of the developed compiler. For our compiler and in comparable

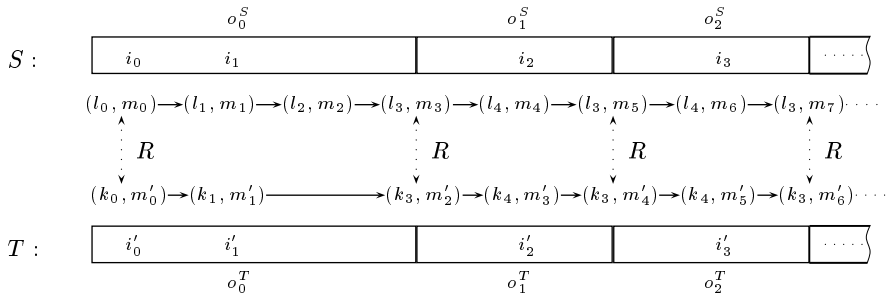

Fig. 4. Runtime behavior of programs S and Tp

scenarios, one can simply use the equality of the semantics of source and target language to define *corrTransl*:

$$corrTransl(S,T) \;=_{def}\; (\; sem_{FGL}(S) = sem_{MSub}(T)\;)$$

For full-blown programming languages, the correctness predicate can become more complex. One reason is nondeterminism in the source language, for example caused by concurrency constructs. Another reason is the use of bounded ressources. For example, the source language semantics might not capture program abortion due to lack of memory. Then, a translation of source S might be considered correct, if the target obey S's semantics as long as there is enough memory and aborts otherwise.

4 Proof Technique for Certifying Translations

This section explains the simulation-based proof technique that we use in our certifying compiler. It describes how the proofs are structured into program-independent and program-dependent parts and how the generated program-dependent parts look like.

As it is the case for most compilers, we assume that translations are divided into a number of phases, each phase having a source and target language. For example, our compiler consists of five phases, one that translates the C subset into the intermediate language FGL, three optimization phases with FGL as source and target language, and finally the code generation producing MSub code. We show for each[5] phase the semantic equality $sem_S(S) = sem_T(T)$. The proof of our correctness predicate follows by transitivity of equality.

4.1 Simulation Proof for a Single Phase

Semantic equality of a single phase is shown by a simulation-based technique (for the background e.g. [4]). Figure 4 illustrates the underlying proof idea. It shows

[5] As said above, the proof generation for the first phase is not yet implemented.

the execution traces of a source program and a target program consisting of labels l_j and k_j resp. and states m_j and m'_j. The outputs i_j and i'_j are given above and below the traces. The simulation proof is based on a static *decomposition* of the flow graphs of S and T into paths of finite length. Each finite path is regarded as an atomic step in our simulation proof.

The definitions of the flow graphs for FGL and MSub are straightforward. We assume that nodes are identified by their labels, and that the successor relation is denoted by *succ*. A *path* π of length $|\pi|$ is a non-empty list of labels $\pi(0), \ldots, \pi(|\pi|)$ such that $succ(\pi_j, \pi_{j+1})$. For constant folding (CF), dead assignment elimination (DAE), and codegeneration (CG), we only need a decomposition technique with non-overlapping paths where all paths starting in the same node have the same length. For CF the length of all paths in source and target is one. For DAE the paths in the source may be longer than one, containing one live assignment and several dead assignments; in the target paths have the length one. For CG the length of all paths in the source is one and in the target one or larger. For optimizations modifying the program structure, like our loop invariant hoisting (LIH), we developed a decomposition technique with overlapping paths. For brevity, we only consider a simpler decomposition here. The simulation-based proof technique is the same for both cases.

A decomposition for a program P_L of language L with labels B_L is formalized as a function $d_L : B_L \to nat$ such that $d_L(l) > 0$ iff l is the start node of a path. In that case $d_L(l)$ is the length of the paths starting at l. Otherwise, $d_L(l)$ is zero, indicating that d_L is not defined for l. The details of d_L and definition of well-formedness are given in the appendix.

The informal idea underlying the simulation proof is that whenever we start source program S and target T in configurations satisfying the simulation invariant R (see Fig. 4) and then iteratively follow a path in S and the corresponding path in T, we reach configurations that satisfy R. In the following, we describe those aspects of the proof technique in more detail that we need to explain which parts of the proofs are program-independent and which parts are generated. For any program P with initial configuration c_0^P, a wellformed decomposition d_P defines a sequence of configurations c_i^P by $c_{i+1}^P =_{def} nstep(d_P(label(c_i^P)), P, c_i^P)$ and a sequence of partial outputs o_i^P such that $output(c_i^P)$ concatenated with o_i^P equals c_{i+1}^P, that is, o_i^P is the output generated by executing the instructions of a path starting at $label(c_i^P)$.

For any source and target programs S and T and wellformed decompositions d_S and d_T, a binary relation $R[S, d_S, T, d_T]$ over the configurations of S and T is called a *simulation invariant* iff

$$
\begin{array}{l}
R[S, d_S, T, d_T](c_0^S, c_0^T) \ \wedge \\
\forall\, i \in nat.\ \ R[S, d_S, T, d_T](c_i^S, c_i^T) \Longrightarrow \\
\qquad\qquad R[S, d_S, T, d_T](c_{i+1}^S, c_{i+1}^T) \ \wedge\ status(c_{i+1}^S) = status(c_{i+1}^T) \ \wedge\ o_i^S = o_i^T
\end{array}
$$

The correctness proofs of all phases are based on the following *program-independent* main lemma that is proved once and used in all program dependent proofs:

Lemma 1. *(bisimulation lemma) For any S and T with wellformed decompositions d_S and d_T, if there exists a simulation invariant $R[S, d_S, T, d_T]$, then S and T are semantically equivalent, that is, $sem_{SL}(S) = sem_{TL}(T)$*

The task of a certifying compiler is to come up with appropriate decompositions d_S and d_T, a relation $R[S, d_S, T, d_T]$ over configurations, and a proof that $R[S, d_S, T, d_T]$ is a simulation invariant. The invariant typically consists of program-independent and program-dependent parts. The program-independent parts capture the behavior underlying the optimization or translation phase. Program-dependent are the label relation $CLABS$ expressing the correspondence between the labels of source and target and further information relevant for the particular phase. We demonstrate this in the following subsection.

4.2 Phase-specific Simulation Relations

In this subsection, we explain the simulation relations for constant folding and code generation.

Simulation Relation for CF. As constant folding only modifies the instructions of the source program, but not the flow graph structure, we can use the trivial decomposition where all paths have length one and $CLABS$ is the equality on labels. This is illustrated on the left-hand side of Fig. 5. Thus, the only program-dependent part of the simulation relation $\mathbf{R_{CF}}$ is the result of the Constant Folding Analysis. The analysis result $\alpha_S : B(S) \to (Identifier \hookrightarrow integer)$ maps the label set $B(S)$ of the source program S to partial functions capturing for a subset of the variables in the program a constant value. For example, $\alpha_S(l)$ captures for all variables that are detected to be constant at l their values. Based on this information, $\mathbf{R_{CF}}$ is defined by:

$$R_{CF}[S, d_S, T, d_T](\alpha_S)((l, m, s, o), (l', m', s', o')) =_{def} \texttt{let } inv = \alpha_S(l) \texttt{ in}$$
$$l = l' \wedge m = m' \wedge s = s' \wedge o = o' \wedge \forall\, id \in dom(inv).\, inv(id) = m(id)$$

Simulation Relation for CG. The simulation relation $\mathbf{R_{CG}}$ for the code generation depends on the relation $CLABS$ of labels in source and target and on the allocation of variables to registers and memory cells. As illustrated in Fig. 5, $CLABS$ maps FGL labels to labels of MSub instructions such that paths in FGL have length one and paths in MSub are usually larger than one. Allocation is described as a mapping μ from the identifiers $Identif[S]$ of program S to adresses. An address is a register number or a memory address. The predicate $isArr[S](id)$ indicates whether id denotes an array in S. In that case, $indices[S](id)$ denotes the set of allowed indices.

$$\begin{aligned}
&R_{CG}[S, d_S, T, d_T](CLABS, \mu)((l, m, s, o), (l', (regs, mem), s', o'))) =_{def} \\
&\quad (l, l') \in CLABS \ \wedge\ s = s' \ \wedge\ o = o' \ \wedge \\
&\quad \forall\, id \in Identif[S]). \\
&\qquad (\ \neg isArr[S](id) \ \wedge\ \mu(id) \in dom(regs) \wedge m(id) = regs(\mu(id))\ ) \\
&\quad \vee\ (\ \neg isArr[S](id) \ \wedge\ \mu(id) \in dom(mem) \wedge m(id) = mem(\mu(id))\ ) \\
&\quad \vee\ (\ isArr[S](id) \ \wedge\ \forall n \in indices[S](id). \\
&\qquad\qquad (\mu(id) + 4 * n) \in dom(mem) \wedge m(id, n) = mem(\mu(id) + 4 * n)\ \ )
\end{aligned}$$

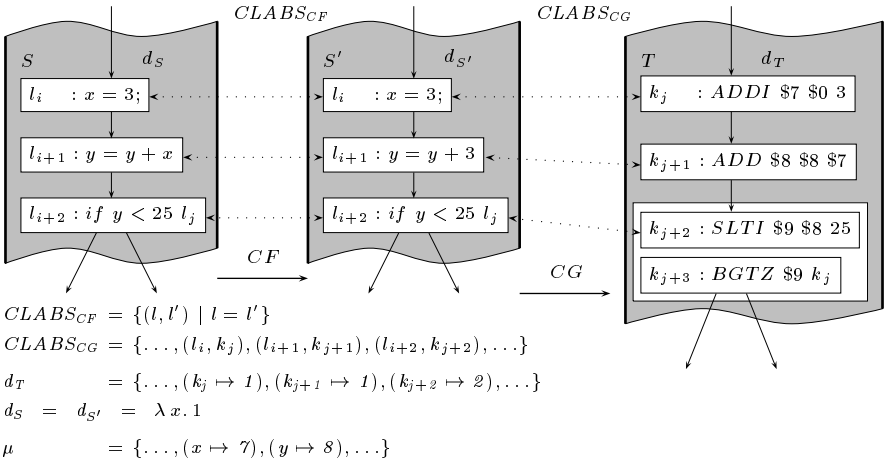

Fig. 5. Corresponding label relation for CF and CG

4.3 Proof Generation

This subsection describes the proof parts generated by the compiler for a given phase and program. (The proof for a multi-phase translation is obtained by using the transitivity of equality; see above.) As illustrated for R_{CF} and R_{CG}, the simulation relations are parameterized by source and target programs, by the flow graph decompositions, and by some phase-specific information like for example α and μ. Based on the knowledge on S and T, the compiler generates the following specifications for the proof assistant:

1. Definions for programs, the flow graph decompositions, the relation $CLABS$, and the phase-specific information *psinfo*.
2. A proof script showing that the decompositions d_S and d_T are well-formed.
3. Proof scripts showing phase-specific properties (e.g. the injectivity of the allocation mapping μ).
4. A proof script showing that $R[S, d_S, T, d_T](psinfo)(c_0^S, c_0^T)$ holds for the initial configurations.
5. For each pair of labels $(\mathtt{l}, \mathtt{l}') \in CLABS$ a lemma of the form given in Fig. 6. These lemmas are called *simulation-block lemmas*.
6. Proof scripts for the simulation-block lemmas.
7. A proof script showing that the simulation-block lemmas imply that $R[S, d_S, T, d_T](psinfo)$ is a simulation invariant and that applies the bisimulation lemma to obtain the correctness of the considered phase.

The simulation-block lemmas in their concrete form, that is, with all program-dependent parameters instantiated express that running one or a small number of instructions on the source side has the same effect as running a certain number of instructions on the target side. For given S and T, there is a finite number of simulation-block lemmas and each lemma covers the execution

$$
\begin{array}{l}
\forall m, s, o, m', s', o'. \\
\quad R[S, d_S, T, d_T](psinfo)((\mathtt{l}, m, s, o), (\mathtt{l}', m', s', o')) \\
\Longrightarrow \\
\quad \mathtt{let}\ \ c\ = nstep_{SL}(d_S(\mathtt{l}), S, (\mathtt{l}, m, s, o)) \\
\quad \mathtt{and}\ \ c' = nstep_{TL}(d_T(\mathtt{l}'), T, (\mathtt{l}', m', s', o')) \\
\quad \mathtt{in}\ \ R[S, d_S, T, d_T](psinfo)(c, c') \wedge\ status(c) = status(c')\ \wedge\ output(c) = output(c')
\end{array}
$$

Fig. 6. Lemma for simulation blocks starting at $\mathtt{l}$ and $\mathtt{l}'$

of corresponding paths of S and T. As the paths cover the flow graph of S, a proof by case distinction allows us to derive from the simulation-block lemmas that $R[S, d_S, T, d_T](psinfo)$ is a simulation invariant and that the bisimulation lemma yields the overall proof goal. Except for this case distinction, most proofs are essentially rewriting proofs enfolding the semantics definitions for the instructions.

5 Using Other Techniques for Certifying Compilers

The central idea of the certifying compiler approach is that the client of the compiler obtains a checkable certificate for the correctness of a translation. The verification technique presented in the last section is only one way to generate the certificates. Here, we shortly describe and discuss how techniques for algorithm verification and translation validation can be used for our goals.

Algorithm Verification. Following a technique sketched in [9], Sect. 2, correctness proofs for all or some of the algorithms in a compiler can be used to obtain translation certificates. Let us assume that the compilation algorithm is specified as a computable function *comp* in the higher-order logic and that we have a correctness proof for it, i.e. a proof for:

$$\forall S.S \in SL : corrTransl(S, comp(S))$$

If an implementation *icomp* of the compiler produces a target T for a source program S, we can construct a certificate for $corrTransl(S, T)$ by verifying $comp(S) = T$ using rewriting techniques and then instantiating the above general correctness proof. The advantage of this approach over compiler certification is that a proof of the implementation of *icomp* is correct can be by passed. The advantage over our approach is that the compiler implementation needs no instrumentation. Similar to our approach, the construction of the certificate can fail, namely if $comp(S) = T$ cannot be established. The disadvantages compared to our approach are the following:

1. The certificates become huge because they includes the correctness proof for the translation of all programs. Leroy suggests in [9] to mitigate this problem by developing techniques of specializing proofs.

2. Checking $comp(S) = T$ might be slower than checking dedicated certificates.
3. To our experience, the proof of algorithm correctness is more complex than to proof the correctness of the translation result.

Translation Validation. As said above, one disadvantage of our approach is the instrumentation of the compiler, because instrumentation causes development effort and increases the complexity of the compiler. By using techniques from translation validation, the last problem could be almost avoided. The idea is to reduce instrumentation to a minimum and let the compiler only generate some "hints", for example, on the allocation of variables to memory cells. Techniques from translation validation (see in particular [16]) could then be used to construct a complete proof script from these hints. Even more in the line of translation validation is a technique that avoids explicit proof scripts. Based on the strategy mechanisms of the underlying proof assistant, one could develop proof tactics that take the hints as input and directly construct a proof from them, that is, one would implement translation validation using the mechanisms of the proof assistant. This technique allows to use algorithm-independent proof techniques of Sect. 4 with a minimum of program-dependent information. We applied this technique to an optimization phase. Our first experiences are very encouraging.

6 Evaluation and Performance Issues

This section briefly discusses performance issues concerning the proof checking. The generated proof scripts are run by Isabelle/HOL and it is checked whether they correctly construct a proof. In the current implementation of Isabelle/HOL, checking/construction of proofs that our approach generates is rather slow.[6] So far, we identified the following reasons for this behaviour:

- Many steps in our proofs are of a computational nature. Executing these steps in a theorem proving environment is very slow because most of these steps are done by term rewriting on the data structure underlying HOL formulas that is overly general and complex for our tasks.
- In our proof scripts, several steps still use tactics of the theorem prover that do some search.
- Finding an optimal order for the application of tactics is challenging, partially because the efficiency properties of the proof assistant are difficult to analyse.

Concerning the first item, we plan to compare with other provers. The problem stated as second item may be solved by using lower level tactics or special user defined tactics. In the following part, we give a simple example of how to improve the problem mentioned in the third item: Improving efficiency by restructuring the underlying proof techniques.

The time consuming part of a typical code generation correctness proof is a case distinction on labels in FGL/MSub as described in Sect. 4: For each pair

[6] According to our experience this is as well true for comparable proof assistants.

of corresponding labels in an FGL- and MSub-program, we have to prove the simulation-block lemma. As a straightforward approach to prove a single step of the programs correct, one could execute the programs symbolically. Although such proofs always succeed in theory, they are forbiddingly slow to handle realistic programs. The problem is that the approach needs a case distinction on all variables involved in the program. And, every array element counts as a single variable in this distinction. Each variable had to have a value equal to the corresponding memory location. Thus, in each step for every variable occuring in the FGL program the corresponding memory location in the MSub program had to be looked up. This correspondance relation between variables and memory is stored in a list. Using Isabelle tactics each look up took $O(v)$ time with v being the number of variables. Hence, the time to process the proof for steps of the program was in $O(l \times v^2)$ with l being the length of the program.

In our current approach, we make use of the fact that each step can be proved correct without looking at other variables not occuring in the step if the allocation mapping μ is injective. Hence a variable's corresponding memory location is not altered if some others variable's memory location is changed. With the help of this we can dismiss of the last case distinction when proving the injectivity of the mapping between variables and memory upfront. The proof of the steps can be conducted in $O(l \times v)$. The proof of injectivity can be done in time $O(v)$ for non-pathological cases. Hence the complete proof can be done in roughly $\mathrm{O}(l \times v)$ time.

7 Conclusion

Formal translation contracts are the requirements specification for the development of certified or certifying compilers. We used a contract that specifies semantical equivalence on the basis of output traces of the considered source and target language. This avoids to define a relation between final program state and final memory state, and it supports nonterminating programs. We implemented a simple certifying compiler with optimization and code generation phases that produces machine-checkable proof scripts. Whereas current specification and verification technology is sufficient to express the translation contract, additional properties, and proofs in a fairly convenient way, the proof checking technology could be improved: It is mainly targeted at complex interactive proofs and not suitable to check simple, but large proofs. Future work includes the extension of our compiler, as well as the application of the 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. 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.
4. R.J. van Glabbeek. The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001. Available at `http://boole.stanford.edu/pub/spectrum1.ps.gz`.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. George C. Necula. Proof-carrying code. ACM Symposium on Principles of Programming Languages and Systems, Paris, France, January 1997.
12. George C. Necula. *Compiling with Proofs*. PhD thesis, 1998.
13. 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.
14. 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.
15. 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.
16. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151+, 1998.
17. Arnd Poetzsch-Heffter and Marek J. Gawkowski. Towards proof generating compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):37–51, 2005.
18. M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.
19. Martin Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, MIT Laboratory for Computer Science, March 1999.

20. 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.

Appendix: Program Decomposition

A decomposition for a program P_L of language L with labels B_L is formalized as a function $d_L : B_L \to nat$ such that $d_L(l) > 0$ iff l is the start node of a path. In that case $d_L(l)$ is the length of the paths starting at l. Otherwise, $d_L(l)$ is zero, indicating that d_L is not defined for l. Let $start(d_L) =_{def} \{l \mid d_L(l) > 0\}$ be the set of starting labels of paths, $end(d_L) =_{def} \{l \mid \exists \pi : d_L(\pi(0)) \in start(d_L) \wedge l = \pi(|\pi|)\}$ be the set of end labels of paths, and $between(d_L) =_{def} \{l \mid \exists \pi, j : d_L(\pi(0)) \in start(d_L) \wedge 0 < j < |\pi| \wedge l = \pi(j)\}$ be the set of labels between start and end. We say that d_L is wellformed for P_L, if the program entry label $l_0 \in start(d_L)$, the program exit label $l_e \in end(d_L)$, each path ending in l different from l_e has a successor path starting in l, i.e., $end(d_L) \setminus \{l_e\} = start(d_L) \setminus \{l_0\}$, and nodes between start and end label are not end labels of other paths, i.e., $between(d_L) \cap (end(d_L) \cup \{l_0\} = \emptyset$.