Implementing a Direct Method for Certificate Translation*

Gilles Barthe[1], Benjamin Grégoire[2], Sylvain Heraud[2], César Kunz[1], and Anne Pacalet[2]

[1] IMDEA Software, Spain
[2] INRIA Sophia Antipolis - Méditerranée, France

Abstract. Certificate translation is a method that transforms certificates of source programs into certificates of their compilation. It provides strong guarantees on low-level code, and is useful for eliminating trust in the compiler (for high assurance code) and in the code producer for mobile code security. The theory of certificate translation has been developed in earlier work, but no implementation exists. As a result, it has been difficult to evaluate its practicality, and in particular the impact of certificate translation on the size of certificates.

In this paper, we report on the development of a certificate translator prototype. The tool takes as input a high-level program, defined in a small subset of the C programming language, and a logical specification *à la ACSL*, and computes a set of verification conditions for the Coq proof assistant. Once proof obligations are discharged, the tool compiles the source program into an intermediate RTL (i.e., three-address code) representation, and then performs a sequence of compiler optimizations. At each step, certificates are transformed automatically to produce a proof for the transformed programs. For optimizations that rely on arithmetic reasoning, such as constant propagation and common subexpression, the tool implements a new certificate translation strategy that minimizes certificate growth.

1 Introduction

Reasoning about source programs is important: it helps guarantee that programs have been correctly designed to meet some functionality, or some non-functional requirement. Reasoning about source programs is also successful: modern program verifiers for C, C#, Java, perform well, and their user base is growing across different areas of computer science.

Yet, reasoning about source programs does not provide guarantees about the behavior of executable code. The dissimilarity between a compliant source program and a misbehaved executable can arise on many accounts: the compiler may modify the functional semantics of the program, or it may preserve its functional semantics and yet alter its non-functional behavior (execution time, resource consumption), leading to security vulnerabilities or to ill-functioning

* Partially funded by the EU projects MOBIUS and HATS.

K. Breitman and A. Cavalcanti (Eds.): ICFEM 2009, LNCS 5885, pp. 541–560, 2009.
© Springer-Verlag Berlin Heidelberg 2009

applications. Or, the executable code may be produced by a malicious third party that deliberately aims to provide users with code that does not respect the behavior of the source programs. Or, the compiled code might have been tampered prior to execution, for example for efficiency reasons. Thus, source code verification must be complemented with verification of executable code.

Program verification is costly, and there is little chance that programmers will agree to verify their source programs, and then to verify again the corresponding executable code. Verification across the compilation chain is even more problematic for source programs that are compiled to different targets, as the verification effort would need to be repeated for each potential target. Therefore, it is important to develop methods that allow transferring the results of source code verification to lower levels in the compilation chain.

Certificate translation is a method to transfer evidence from source programs to compiled programs. They manipulate so-called certificates, i.e. mathematical objects that capture program correctness proofs, and amenable to transformation by syntactic methods. In short, certificate translators are functions that map certificates of source programs, into certificates of their compilation.

The theory of certificate translation has been developed in previous work: in particular, Barthe *et al* [3] show in the setting of a RTL language that certificate translators exist for common program optimizations; later, Barthe and Kunz [2] show that certificate translators are guaranteed to exist under mild conditions. However, the theoretical development has not been matched by a practical implementation. As a result, it has not been possible to validate experimentally the theoretical developments, nor to assess the practicality of certificate translation.

In this paper, we report on a prototype implementation[1] of certificate translation for a subset of the C language. Although our verifier for C programs and our examples remain modest in comparison with the state-of-the-art (compared e.g. with Spec# [1] or Frama-C [5]), our work allows to draw for the first time some preliminary conclusions about the practicality of certificate translation. In particular, we are able to provide a preliminary analysis of the impact of certificate translation on the size of certificates, which is an essential metrics for Proof Carrying Code [7]—the initial motivation of our work. In addition to the tool, we report on a new method for transforming certificates for optimizations based on arithmetic reasoning, such as constant propagation, and common subexpression elimination. The method is simpler, in that it treats the certificate of the original program as a black-box, whereas the previous method in [3] involved weaving certificates, using a well-founded induction principle on the control flow graph of the program. In comparison, the new method also yields smaller certificates, and thus contributes to the practicality of certificate translation.

2 Overview

Figure 2 provides an overview of the tool. The tool operates on programs written in a subset of the C language. A program consists of a declaration of global

[1] The tool is available at `http://mobius.inria.fr/CertificateTranslation`

```
l1  : i := 0;
l2  : while (i < m){
l3  :   j := 0;
l4  :   while (j < p){
l5  :     k := 0;
l6  :     c[i * p + j] := 0;
l7  :     while (k < n){
l8  :       c[i * p + j] := c[i * p + j] + a[i * n + k] * b[k * p + j];
l9  :       k := k + 1;
          }
l10 :     j := j + 1;
        }
l11 :   i := i + 1;
      }
l12 : return 0
```

Fig. 1. Source code of matrix multiplication example

variables, followed by a sequence of function declarations. Each function defines the return type, the name and type of the formal parameters, a set of local variables, and the statement that defines its body. Variable types are restricted to integers and pointers to integer values. Statements include assignments to scalar and pointer variables, function invocation, conditional and loop statements. Program points can be labeled to be used as the target of a `goto` statement. Local declarations of pointer variables and expressions containing pointer arithmetic operations are not allowed.

Although minimalistic, the fragment considered is sufficiently expressive for writing many algorithms of interest. We have programmed (and verified) sorting algorithms, and algorithms that manipulate matrices. Our running example is a matrix multiplication algorithm.

Example 1 (Matrix Multiplication: source code). Consider as a running example the following algorithm, that computes the multiplication of two matrices `a` and `b`, and stores it in `c`. Matrices are encoded as uni-dimensional arrays; for example,

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

is encoded as $\begin{pmatrix} a & b & c & d \end{pmatrix}$. More generally, if a $m \times n$ matrix A is represented by the uni-dimensional array `a` of size mn, we encode the array element $A_{i,j}$ as $\texttt{a}[i * n + j]$. The code is given in Figure 1. In the algorithm, the array variables `a`, `b`, and `c` represent a $\texttt{m} \times \texttt{n}$ matrix, a $\texttt{n} \times \texttt{p}$ matrix, and a $\texttt{m} \times \texttt{p}$ matrix, respectively. The l_is are program labels, and are used to add annotations in the program text.

The program verifier operates on annotated source programs. In order to support effective and modular verification, the program verifier requires that procedures are annotated with their preconditions and postconditions, and that loops are annotated with their invariants. However, the verifier allows assertions to be

inserted at arbitrary points in the program, which is particularly useful to verify some non-functional properties. Procedure specifications are triples of the form $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post}\rangle$, where Pre and Post are assertions and annot is a partial function from program labels to assertions. Assertions are written in a language similar to (but smaller than) the ACSL language used in the Frama-C project [5]; they are also allowed to refer to functions and predicates defined in an external Coq module. One minor difference with ACSL is that the specification language does not provide any syntactic sugar for modifiable clauses.

Example 2 (Matrix Multiplication: specification). The specification for matrix multiplication is given by the triple $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post}\rangle$ where:

$$
\begin{array}{l}
\mathsf{Pre} \doteq 0 < \mathtt{m} \wedge 0 < \mathtt{n} \wedge 0 < \mathtt{p} \\
\mathsf{annot}(l_2) \doteq \forall i, j.\ ((0 \leq i < \mathtt{i}) \Rightarrow (0 \leq j < \mathtt{p}) \Rightarrow \mathtt{c}[i,j] = (\mathtt{a} \times \mathtt{b})[i,j]) \wedge \mathtt{i} \leq \mathtt{m} \\
\mathsf{annot}(l_4) \doteq \forall j.((0 \leq j < \mathtt{j}) \Rightarrow \mathtt{c}[\mathtt{i},j] = (\mathtt{a} \times \mathtt{b})[\mathtt{i},j]) \wedge (0 \leq \mathtt{j} \leq \mathtt{p}) \wedge \\
\qquad\qquad \forall i, j.\ ((0 \leq i < \mathtt{i}) \Rightarrow (0 \leq j < \mathtt{p}) \Rightarrow \mathtt{c}[i,j] = \mathtt{c}^{l_2}[i,j]) \\
\mathsf{annot}(l_7) \doteq \mathtt{c}[\mathtt{i},\mathtt{j}] = \sum_{r=0}^{\mathtt{k}-1}(\mathtt{a}[\mathtt{i},r] * \mathtt{b}[r,\mathtt{j}]) \wedge (0 \leq \mathtt{j} \leq \mathtt{p}) \wedge \\
\qquad\qquad \forall j.\ ((0 \leq j < \mathtt{j}) \Rightarrow \mathtt{c}[\mathtt{i},j] = \mathtt{c}^{l_4}[\mathtt{i},j]) \\
\mathsf{Post} \doteq \forall i, j.\ ((0 \leq i < \mathtt{m}) \Rightarrow (0 \leq j < \mathtt{p}) \Rightarrow \mathtt{c}[i,j] = (\mathtt{a} \times \mathtt{b})[i,j])\}
\end{array}
$$

For notational convenience, we write $x[i,j]$ instead of $x[i * n + j]$ if the array x represents an $m \times n$ matrix. The postcondition ensures that the array c is the result of matrix multiplication between a and b, i.e. for every i and j, $\mathtt{c}[i,j]$ is equal to $\sum_{k=0}^{\mathtt{n}-1} \mathtt{a}[i,k] * \mathtt{b}[k,j]$, denoted $(\mathtt{a} \times \mathtt{b})[i,j]$.

The innermost loop, defined in terms of the induction variable k, computes the sum $\sum_{k=0}^{\mathtt{n}-1} \mathtt{a}[\mathtt{i},k] * \mathtt{b}[k,\mathtt{j}]$, for the current value of the variables i and j. The first term in the conjunction that defines the loop invariant $\mathsf{annot}(l_7)$ expresses exactly this condition. In addition, $\mathsf{annot}(l_7)$ states that for any other pair (i', j') different from $(\mathtt{i},\mathtt{j})$, the array value $\mathtt{c}[i', j']$ remains unmodified ($\mathtt{c}^{l_4}$ stands for the value of c before the assignment $k = 0$). The loop statements at labels l_2 and l_4 traverses the rows and columns of the matrix represented by c, respectively. The invariants $\mathsf{annot}(l_4)$ and $\mathsf{annot}(l_2)$ extend the condition $\mathtt{c}[\mathtt{i},\mathtt{j}] = (\mathtt{a} \times \mathtt{b})[\mathtt{i},\mathtt{j}]$ for the previous values of the variables j and i, respectively.

The program verifier generates for each annotated program a set of proof obligations. The generation of proof obligations proceeds in two phases: first, a symbolic execution algorithm is used to strengthen program annotations. Then, a weakest precondition calculus generates proof obligations using the strengthened annotations. In order to avoid bloated proof obligations, the weakest precondition calculus does not strengthen annotations with all the results of symbolic execution, but only with those that refer to variables that would appear in the proof obligation.

The use of symbolic execution to strengthen invariants allows users provide weaker specifications[2].

Example 3 (Matrix Multiplication: symbolic execution). At program point l_4, a standard VCgen would generate, for the execution that does not enter the loop,

[2] A similar technique is implemented in the Caveat tool, the predecessor of Frama-C.

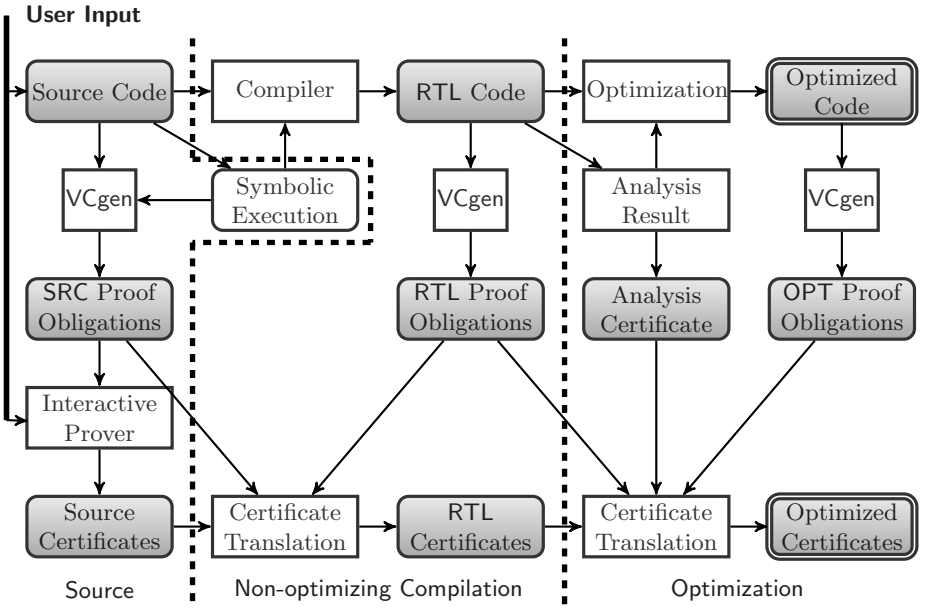

Fig. 2. Overall Tool Scheme

the proof obligation: $\mathsf{annot}(l_4) \Rightarrow \neg \mathtt{j} < \mathtt{p} \Rightarrow \mathsf{annot}(l_2)[^{\mathtt{i}+1}/_{\mathtt{i}}]$. Unfortunately, in this example the loop invariant $\mathsf{annot}(l_4)$ is not strong enough to discharge the proof obligation. In contrast, symbolic execution propagates automatically the condition provided by the outer invariants to strengthen the inner invariants, and generates provable proof obligations.

The obligations are collected in a Coq file, and must be discharged to guarantee that the program is correct w.r.t. its specification. This process is done interactively by the user, using the Coq proof assistant. The certificate of the program is the set of proof terms that are built automatically by Coq upon successful verification of the proof obligations.

Certificate translation starts after the user has completed the verification of the source program. The program is first compiled to an intermediate representation, written in a RTL language with arrays and procedure calls. The compiler from the source language to the target language does not perform any optimization, but replaces booleans, which do not exist in the RTL language, by integers—we do not discuss this transformation further.

Example 4 (Matrix multiplication: RTL *code).* The result of compiling the running example can be found in Figure 3. The boxes in gray show the optimized version of the RTL code, as explained in Section 5.2.

The checker of RTL programs is based on the same principles as the verifier for source programs: annotated RTL programs are sent to a weakest precondition calculus that generates a set of proof obligations; then the programs certificates

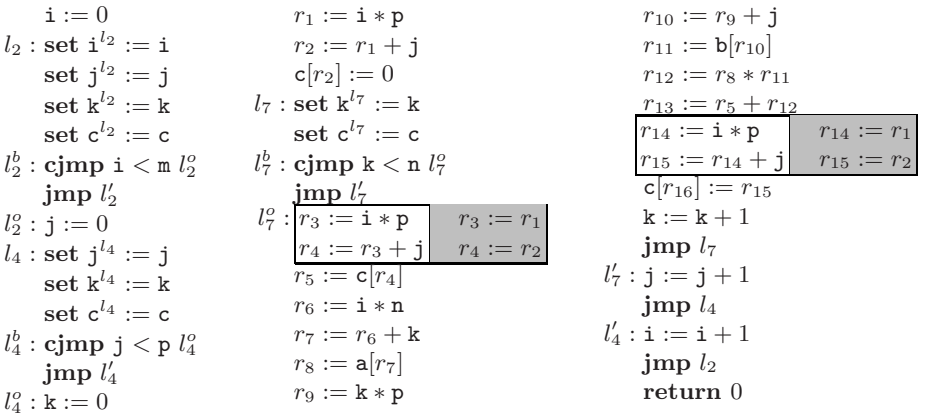

Fig. 3. RTL representation of the matrix multiplication example

integer expressions	$e ::= n \mid x \mid a[e] \mid e + e \mid e * e \mid \ldots$
boolean expressions	$b ::= \mathsf{true} \mid \mathsf{false} \mid \neg b \mid b \wedge b \mid e \leq e \mid e = e \mid \ldots$
statements	$c ::= \textbf{skip} \mid x := e \mid a[e] := e \mid c; c$
	$\mid x :=$ **invoke** $f(e) \mid$ **return** e
	$\mid$ **if** b **then** c **else** $c \mid$ **while** b **do** c

Fig. 4. Source Programs

are checked against the proof obligations. Unlike the verifier for source programs, the checker for RTL programs does not rely on symbolic execution—one reason for this discrepancy is that the RTL checker is trusted, and hence should be as simple as possible. To bridge the gap between the source code verifier and the RTL checker, the tool relies on a specification compiler that strengthens the original specification with the result of the symbolic execution. Although the proof obligations between source code and RTL programs are very similar, there are some minor differences that prevent directly reusing certificates—for many examples, including the running example, the proof obligations coincide syntactically. The tool implements a Coq tactic to transform the original certificates into certificates for their RTL compilation.

The tool then performs a series of optimizations: constant propagation, common subexpression elimination, partial redundancy elimination, and unreachable code elimination. For each optimization, the tool transforms the program, its specification and the certificates. This is done by comparing the original and final proof obligations as a black-box, instead of considering the definition of the compiler. Therefore, although we consider a particular compiler, the certificate transformation technique presented here can be applied to any optimization based on arithmetic simplification. However, it is not completely independent on the compiler, in that sense that the certificate transformation may fail if the transformation is not semantically correct.

As noted in [3], certificate translation require program analyzers to generate a certificate for the result of the analysis. The certificates of the analyses are then merged with the original certificates. In Section 5, we present a new method to perform the merging of certificates. Section 6 provide experimental results.

3 Source Program Verification

For the clarity of exposition, we base the presentation on a simpler language, restricting the heap model to array operations. The set of statements is given in Figure 4. $\mathcal{V}$ and $\mathcal{A}$ represent the set of scalar and array variables, respectively.

Scalar variables are local to the execution of a procedure body, and array variables are global to the execution of the whole program. Array variables represent pointers from a common and global heap. In order to avoid pointer aliasing, array variables hold distinct memory pointers, that are statically defined and cannot be modified along the execution of the program. To enforce this, assignments of the form $a := a'$, where a and a' are array variables, are not allowed in this paper. The program semantics is standard.

3.1 Specification

A procedure specification is provided by a set of logical formulae. The specification may refer to any program variable, and to the special purpose variable `res` that refers to the value returned by a procedure, and to ghost variables, which are used only for specification purposes. A particular subset of ghost variables are the *starred* variables $x^\star$, referring to the initial value of the variable x. In addition to starred variables, we consider also *labeled* variables. One can interpret the ghost variable x^l as standing for the value held by the program variable x at the program point l.

Preconditions are logical formulae that only refers to any array variable and any scalar arguments of the procedure. Postconditions can refer to the variable `res`, any array variables, and any ghost variable.

To compute verification conditions from a procedure and a specification $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post} \rangle$, the verification framework requires that every loop statement l : **while** b **do** c is annotated, that is, that l is in $\mathsf{dom}(\mathsf{annot})$.

3.2 Symbolic Execution

In this section, we describe the invariant strengthening technique based on the symbolic propagation of invariants and path conditions.

A symbolic state is defined by a pair (S, C), where S is an execution store that maps variables to integer expressions composed of only ghost variables. Similarly, C is a formula whose every free variable is a ghost variable. We denote $[\![e]\!]S$ and $[\![\phi]\!]S$ the syntactic substitution in e and ϕ, respectively, of every variable v by Sv.

The symbolic execution of a statement c from a symbolic state (S, C) and returning the symbolic state (S', C'), is denoted $(S, C, c) \rightarrow (S', C')$. The rules that define the symbolic execution can be found in Figure 5. In the figure, a

$$\frac{}{\langle S, C, \mathbf{skip}\rangle \to \langle S, C\rangle}$$

$$\frac{}{\langle S, C, a[e] := e'\rangle \to \langle [S : a \mapsto [\![[a : e \mapsto e']]\!]S], C\rangle}$$

$$\frac{}{\langle S, C, x := e\rangle \to \langle [S : x \mapsto [\![e]\!]S], C\rangle}$$

$$\frac{}{\langle S, C, l : x := \mathbf{invoke}\ f(e)\rangle \to \langle [S : (x, a) \mapsto (x^l, a^l)], C\rangle}$$

$$\frac{\langle S, C, c_1\rangle \to \langle S_1, C_1\rangle \qquad \langle S_1, C_1\rangle \sqsubseteq_{l_2} \langle S_2, C_2\rangle \qquad \langle S_2, C_2, c_2\rangle \to \langle S', C'\rangle}{\langle S, C, l_1 : c_1; l_2 : c_2\rangle \to \langle S', C'\rangle}$$

$$\frac{\langle S, C \wedge [\![b]\!]S, c_1\rangle \to \langle S_t, C_t\rangle \qquad \langle S, C \wedge [\![\neg b]\!]S, c_2\rangle \to \langle S_f, C_f\rangle}{\langle S, C, l : \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\rangle \to \langle S_t \oplus_l S_f, ([\![b]\!]S \Rightarrow C_t) \wedge ([\![\neg b]\!]S \Rightarrow C_f)\rangle}$$

$$\frac{\text{for all } x \text{ modifiable by } c,\ Sx = x^l \qquad \langle S, C \wedge [\![\mathsf{annot}(l)]\!]S \wedge [\![b]\!]S, c\rangle \to \langle S', C'\rangle}{\langle S, C, l : \mathbf{while}\ b\ \mathbf{do}\ c\rangle \to \langle S, C \wedge [\![\neg b]\!]S\rangle}$$

Fig. 5. Symbolic Execution Rules

stands for the array variables modified by the procedure f. The expression $[f : x \mapsto n]$ stands for the function f' such that $f'x = n$ and $f'y = fy$ for all $y \neq x$.

The symbolic execution is assumed to start in an initial state (S_0, C_0) such that S_0 maps every variable x to $x^\star$, that is, to its initial value. The execution proceeds by case analysis on the statement. In the following paragraphs we explain some of the execution rules:

Assignment: In the case of an assignment $x := e$, the symbolic store S is updated so that after the assignment is executed, the variable x holds the corresponding symbolic value $[\![e]\!]S$. A similar substitution is applied in the case of assignments to array variables.

Conditional statement: In the case of the conditional statement of the form **if** b **then** c_1 **else** c_2, each branch c_1 and c_2 is evaluated under the conditions b and $\neg b$, respectively. To this end, the symbolic conditional C is strengthened with the interpretation of the boolean expressions b and $\neg b$, denoted $[\![b]\!]S$ and $[\![\neg b]\!]S$, respectively. Finally, the symbolic states S_t and S_f, resulting from the execution of each branch are merged into a symbolic state $S_t \oplus_l S_f$. The definition of the merging operation preserves the assignments of variables in which S_t and S_f coincide. If S_t and S_f differ on a variable x, this variable is mapped to a fresh labeled variable x^l. That is

$$(S_1 \oplus_l S_2)x = \begin{cases} S_1 x & \text{if } S_1 x = S_2 x \\ x^l & \text{otherwise} \end{cases}$$

Loop statements: The execution of a loop statement may invalidate symbolic conditions if they refer to a variable that may be modified by the loop body. We follow thus a safe approach and require the symbolic execution to start in

an initial state that sets every modifiable variable to a fresh ghost variable x^l, from which nothing is known. The loop body, is then executed in a state in which the interpretation of the guard $[\![b]\!]S$ is incorporated to the conditional C. As the final state, the conditional expression C is strengthened with the negation of the loop guard $[\![\neg b]\!][S : x \mapsto x^l]$.
The tool expects a procedure specification to provide the set of variables that each loop may modify. In return, the tool outputs a set of proof obligations that ensures the correctness of this part of the specification. To simplify the presentation, we assume here that the set of modifiable variables are automatically overapproximated by a static analysis.

Sequential composition: Loops statements require initial symbolic stores S to map modifiable variables to fresh ghost variables from which nothing is known in C. To satisfy this requirement, the composition rule allows one to weaken symbolic states. In the figure, $\langle S, C\rangle \sqsubseteq_l \langle S', C'\rangle$ is defined as $C = C'$, $\mathsf{dom}(S) = \mathsf{dom}(S')$ and for all $x \in \mathsf{dom}(S)$ we have $S(x) = S'(x)$ or $S'x = x^l$.

Note that the symbolic result can be arbitrarily weak by application of the rule for sequential composition, from the requirement $\langle S, C\rangle \sqsubseteq_l \langle S', C'\rangle$. However, since the only statement that put restrictions on the symbolic pre-state is the **while** statement, one can make it as strong as possible by choosing $S' = [S : x \mapsto x^l]$, where x represents every variable modifiable by the loop body, and l is the label of the loop statement. For any other case, one can simply choose $\langle S, C\rangle = \langle S', C'\rangle$. As a result, the symbolic execution becomes deterministic. Therefore, in the rest of the paper we associate every program label l to exactly one symbolic state, represented by the pair (S_l, C_l).

3.3 Verification Condition Generator

In this section we define a verification framework for source-level programs. A verification condition generator (VCgen) takes as input a source program and its specification, and returns a set of proof obligations. The validity of the generated proof obligations ensure that the program satisfies its specification.

The verification framework defined in this section relies on the symbolic execution presented in the paragraphs above. When computing the set of proof obligations, the VCgen strengthens invariants by incorporating the information that is computed by the symbolic analyzer.

The extraction of verification conditions are defined by the function VCG, formalized by the rules in Figure 6. During the computation, every reference to the intermediate specification annot is strengthened with the result of the symbolic execution, and denoted $\overline{\mathsf{annot}}$:

$$\overline{\mathsf{annot}}(l) \doteq \mathsf{annot}(l) \wedge C_l \wedge \bigwedge_{v \in \mathsf{dom}(S_l)} v = S_l(v)$$

Since the VCgen process incorporates the result of the symbolic analysis, the definition of VCG is tightly coupled with the symbolic execution rules. Indeed, the standard weakest precondition must be modified to deal with the fresh ghost

$$\frac{}{\mathsf{VCG}(\mathbf{skip}, \phi) = \langle \phi, \emptyset \rangle} \qquad \frac{}{\mathsf{VCG}(\mathbf{return}\ e, \phi) = \langle \mathsf{Post}[{}^{e}/_{\mathsf{res}}], \emptyset \rangle}$$

$$\frac{}{\mathsf{VCG}(l : x := e, \phi) = \langle \phi[{}^{x}/_{x^l}][{}^{e}/_{x}], \emptyset \rangle}$$

$$\frac{}{\mathsf{VCG}(l : a[e_1] := e_2, \phi) = \langle \phi[{}^{a}/_{a^l}][{}^{[a:e_1 \mapsto e_2]}/_{a}], \emptyset \rangle}$$

$$\frac{\begin{array}{c}\Phi = \mathsf{Pre}_f[{}^{e}/_{x_f}] \wedge \forall \mathtt{res}, V'.\ \mathsf{Post}_f[{}^{V', V}/_{V, V^\star}][{}^{e}/_{x_f{}^\star}] \Rightarrow \phi[{}^{a}/_{a^l}][{}^{v}/_{v^l}][{}^{V'}/_{V}][{}^{\mathtt{res}}/_{v}] \\ V \text{ array variables modified by } f\end{array}}{\mathsf{VCG}(l : v := \mathbf{invoke}\ f(e), \phi) = \langle \Phi, \emptyset \rangle}$$

$$\frac{\mathsf{VCG}(c_1, \phi) = \langle \phi_1, \theta_1 \rangle \qquad \mathsf{VCG}(c_2, \phi) = \langle \phi_2, \theta_2 \rangle}{\mathsf{VCG}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \phi) = \langle b \Rightarrow \phi_1 \wedge \neg b \Rightarrow \phi_2, \theta_1 \cup \theta_2 \rangle}$$

$$\frac{\mathsf{VCG}(c, \overline{\mathsf{annot}}(l)[{}^{x}/_{x^l}]) = \langle \phi_1, \theta \rangle \qquad \Psi \doteq \overline{\mathsf{annot}}(l) \Rightarrow (b \Rightarrow \phi_1) \wedge (\neg b \Rightarrow \phi)}{\mathsf{VCG}(l : \mathbf{while}\ b\ \mathbf{do}\ c, \phi) = \langle \overline{\mathsf{annot}}(l)[{}^{x}/_{x^l}], \{\Psi\} \cup \theta \rangle}$$

$$\frac{\mathsf{VCG}(c_1, \phi_2) = \langle \phi_1, \theta_1 \rangle \qquad \mathsf{VCG}(c_2, \phi) = \langle \phi_2, \theta_2 \rangle}{\mathsf{VCG}(c_1; c_2, \phi) = \langle \phi_1, \theta_1 \cup \theta_2 \rangle}$$

$$\frac{\langle \phi, \theta \rangle = \mathsf{VCG}(c, \mathsf{Post}) \qquad c \text{ the body of } p}{\mathsf{PO}(p) \doteq \{\mathsf{Pre} \Rightarrow \phi[{}^{\vec{V}}/_{\vec{V}^\star}]\} \cup \theta}$$

Fig. 6. Source Code VCgen Rules

variables that are introduced by the symbolic execution. Consider for instance the case of the function invocation $v := \mathbf{invoke}\ f(e)$. Recall that, from the definition of symbolic execution, the final state S' maps v to the fresh ghost variable v^l, and similarly with the arrays that may be modified by f. Then, one must substitute the ghost variable v^l by v, as well as a^l by a, for all array variable a that may be modified by f. A similar situation occurs with loop and conditional statements.

The set of verification conditions is defined as $\mathsf{PO}(p)$ in Figure 6. We say that a procedure p is correct with respect to its specification if $\mathsf{PO}(p)$ is a set of valid formulae. Let the judgment $\langle c, \sigma \rangle \rightsquigarrow \langle n, \sigma' \rangle$ represent the determinist execution of the statement c, starting in a state σ and returning a value n and a final state σ'. The VCgen defined above is sound, that is, the execution of a valid procedure p satisfies its specification. Formally, if c is the statement of a procedure that is valid w.r.t. $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post} \rangle$, and the σ satisfies the precondition Pre, the execution $\langle c, \sigma \rangle \rightsquigarrow \langle n, \sigma' \rangle$ implies that σ' satisfies the postcondition $\mathsf{Post}[{}^{n}/_{\mathtt{res}}]$.

Example 5. Consider again the source code specification provided in Example 2 for the matrix multiplication example. The program code between labels l_2 and l_4 does not modify the array $\mathtt{c}$ in the range $0 \leq i < \mathtt{i}$ and $0 \leq j < \mathtt{p}$. Therefore, the symbolic execution can propagate the information provided by the invariant $\mathsf{annot}(l_2)$:

$$\varphi \doteq \mathtt{i} = \mathtt{i}^{l_2} \wedge \forall i, j.\ ((0 \leq i < \mathtt{i}^{l_2}) \Rightarrow (0 \leq j < \mathtt{p}) \Rightarrow \mathtt{c}^{l_2}[i, j] = (\mathtt{a} \times \mathtt{b})[i, j])$$

to strengthen the invariant $\mathsf{annot}(l_4)$, making thus the specification valid. In this particular case, one can see that the new proof obligation is

$$\mathsf{annot}(l_4) \wedge \varphi \Rightarrow \left| \begin{array}{l} (\forall i, j.\ ((0 \leq i < \mathtt{i}) \Rightarrow (0 \leq j < \mathtt{p}) \Rightarrow \mathtt{c}[i, j] = (\mathtt{a} \times \mathtt{b})[i, j]) \\ \wedge \\ \forall j.\ ((0 \leq j < \mathtt{p}) \Rightarrow \mathtt{c}[\mathtt{i}, j] = (\mathtt{a} \times \mathtt{b})[\mathtt{i}, j])) \end{array} \right.$$

which is a valid logical formula.

4 RTL Program Checking

As a compiler intermediate program representation we assume a Register Transfer Language (RTL), that is, three-address based code. It is an unstructured low-level language with conditional jumps, assignments of atomic expressions and function invocations.

A RTL program is defined as a collection of RTL procedures. Each RTL procedure is defined by its formal parameters and a sequence of labeled low-level instructions. RTL instructions and expressions are defined in Figure 7.

In Figure 7, x stands for a scalar program variable or an integer constant and a stands for an array variable. Assignments involve at most one array access or two scalar variables. In the figure, $\bar{e}$ represents an atomic integer expression (one array access or an arithmetic operation between at most two scalar variables), e an integer expression as defined in the source program syntax, and $\hat{v}$ stands for a ghost variable. The symbol $\bowtie$ stands for any integer comparison. Since ghost variables cannot interfere with the program semantics, they can only appear on **set** instructions. RTL instructions include also a **return** instruction, procedure invocation and conditional and unconditional jumps.

For a RTL procedure p and a label l, we denote $p[l]$ the instruction located at the position with label l. For a label of a RTL procedure p, we define the successors labels, denoted $\mathsf{succ}(l)$, by case analysis on the instruction $p[l]$. For $p[l]$ equal to **return** x we have $\mathsf{succ}(l) = \emptyset$, for **jmp** l' we have $\mathsf{succ}(l) = \{l'\}$, and for **cjmp** l', $\mathsf{succ}(l) = \{l', l+1\}$, where $l+1$ is the label of the next instruction in the sequence p. In any other case, $\mathsf{succ}(l) = \{l+1\}$.

Program Verification. Procedure specifications are provided by triples of the form $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post} \rangle$, where Pre and Post are assertions representing the pre and postconditions and annot is a partial function that maps labels to assertions specifying the loop invariants. Pre, Post and annot are written in the same language and follow the same restrictions as source program specifications.

$$\begin{array}{rrl} \textbf{(atomic expressions)} & \bar{e} & ::= x \mid x + x \mid x * x \mid \ldots \mid a[x] \\ \textbf{(instructions)} & \mathit{ins} & ::= \textbf{nop} \mid v := \bar{e} \mid a[x] := \bar{e} \\ & & \quad \mid \textbf{invoke}\ f\ (\vec{x}) \mid \textbf{return}\ x \mid \textbf{set}\ \hat{v} := e \\ & & \quad \mid \textbf{cjmp}\ x \bowtie x\ l \mid \textbf{jmp}\ l \end{array}$$

Fig. 7. RTL Instructions

$$
\begin{array}{l}
\mathsf{wpi}(l : \mathbf{nop}) = \mathsf{wp}(l+1) \\
\mathsf{wpi}(l : v := e) = \mathsf{wp}(l+1)[{}^{e}/_{v}] \\
\mathsf{wpi}(l : \mathbf{set}\ \hat{v} := e) = \mathsf{wp}(l+1)[{}^{e}/_{\hat{v}}] \\
\mathsf{wpi}(l : a[v_1] := v_2) = \mathsf{wp}(l+1)[{}^{[a:v_1 \mapsto v_2]}/_{a}] \\
\mathsf{wpi}(l : \mathbf{cjmp}\ v_1 \bowtie v_2\ l') = ((v_1 \bowtie v_2) \Rightarrow \mathsf{wp}(l')) \wedge (\neg(v_1 \bowtie v_2) \Rightarrow \mathsf{wp}(l+1)) \\
\mathsf{wpi}(l : \mathbf{invoke}\ f\ x) = \mathsf{Pre}_f[{}^{x}/_{x_f}] \wedge \forall_{\mathtt{res},a'} \mathsf{Post}_f[{}^{a',a}/_{a,a^\star}][{}^{x_f{}^\star}/_{x}] \Rightarrow \mathsf{wp}(l+1)[{}^{a'}/_{a}] \\
\mathsf{wpi}(\mathbf{return}\ v) = \mathsf{Post}[{}^{v}/_{\mathtt{res}}]
\end{array}
$$

$$
\mathsf{wp}(l) = \begin{cases} \mathsf{annot}(l) & \text{if } l \in \mathsf{dom}(\mathsf{annot}) \\ \mathsf{wpi}(l : p[l]) & \text{otherwise} \end{cases}
$$

Fig. 8. RTL VCgen rules

To compute a set of proof obligations from a procedure p and its specification, the RTL checker requires that p is well-annotated. A procedure p with specification $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$ is well-annotated if every cyclic path in its control flow graph contains at least one annotated label in $\mathsf{dom}(\mathsf{annot})$. The result of compiling a well-annotated source procedure is a well-annotated RTL procedure: without further ado, we thus consider only well-annotated programs.

For a RTL procedure p with specification $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post} \rangle$, the proof obligations are defined by the set:

$$
\mathsf{po}(p) = \{\mathsf{Pre} \Rightarrow \mathsf{wp}(l_{\mathsf{in}})[{}^{V}/_{V^\star}]\} \cup \{\mathsf{annot}(l) \Rightarrow \mathsf{wpi}(l : p[l]) \mid l \in \mathsf{dom}(\mathsf{annot})\}
$$

where the predicate transformers wpi and wp are defined in Figure 8. In the figure, a represents every array variable that may get modified by f. The assertion $\varphi[{}^{V}/_{V^\star}]$ stands for the substitution in φ of every array variable $a^\star \in V^\star$ by a.

Consider a RTL program such that every procedure is verified correct with respect to its specification. Then, as with source programs, every terminating execution of a procedure with specification $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post} \rangle$, from an initial state that satisfies the precondition Pre, will reach a final state satisfying the postcondition Post.

5 Certificate Translation

In this section, we deal with certificate transformation along a common compilation process. We show that the first compiler step, that is, non-optimizing compilation, preserves verification conditions up to minor differences. Then we explain how we deal with the transformation of certificates in the presence of compiler optimizations. We first provide an introduction to certificate translation in the most general case. Then, we provide a more ad-hoc technique, implemented in the tool, that considers a particular class of optimizations, for which the growth of the original certificate size is reduced.

5.1 Non-optimizing Compilation

The non-optimizing compiler transforms every source code statement into an intermediate RTL representation. The transformation is defined as a function

$$\mathcal{C}_{(l')}(v := e) = \mathcal{C}^{\mathsf{e}}(v, e) :: \textbf{jmp } l'$$
$$\mathcal{C}_{(l')}(l : \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2) = \mathcal{C}^{\mathbf{b}}_{(l_t, l_f)}(b) :: l_f : \mathcal{C}_{(l'')}(c_2) :: l_t : \mathcal{C}_{(l'')}(c_1) :: l'' : \textbf{set } x^l := x :: \textbf{jmp } l'$$
$$\mathcal{C}_{(l')}(l : \textbf{while } b \textbf{ do } c) = l : \textbf{set } x^l := x :: l^b : \mathcal{C}^{\mathbf{b}}_{(l_o, l')}(b) :: l_o : \mathcal{C}_{(l)}(c)$$
$$\mathcal{C}_{(l')}(v := \textbf{invoke } p(e_1, .., e_k)) = \mathcal{C}^{\mathsf{e}}(v_1, e_1) :: .. :: \mathcal{C}^{\mathsf{e}}(v_k, e_k) :: v := \textbf{invoke } f(v_1, .., v_k) :: \textbf{set } v^l := v :: \textbf{set } a^l := a :: \textbf{jmp } l'$$
$$\mathcal{C}_{(l')}(c_1; c_2) = \mathcal{C}_{(l'')}(c_1) :: l'' : \mathcal{C}_{(l')}(c_2) :: \textbf{jmp } l'$$
$$\mathcal{C}_{(l')}(\textbf{return } e) = \mathcal{C}^{\mathsf{e}}(v, e) :: \textbf{return } v$$

Fig. 9. Non-optimizing Compiler (Excerpt)

$\mathcal{C}$, shown in Figure 9, that takes a source code statement and a label l' where execution must continue after the execution of the compiled statement. In addition, auxiliary functions are defined, in order to compile boolean and integer expressions. $\mathcal{C}^{\mathbf{b}}$ takes a boolean expression, and two output labels, to which execution must flow depending on the evaluation of the boolean expression. This function decomposes the evaluation of the boolean condition into two sequences of RTL instructions. Each of them evaluates an atomic conditional expression. Similarly, the function $\mathcal{C}^{\mathsf{e}}$ takes a variable v and an expression e and returns RTL code that stores the evaluation of e in the variable v. It does so by decomposing the evaluation of the expression e into the evaluation of its atomic subexpressions.

Since we adopt a standard VCgen for RTL procedures, the compiler must incorporate the result of the symbolic execution, used to assist source code verification, to the specification of the resulting RTL procedure.

Consider a source code procedure p, with specification $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post} \rangle$. Let $p' = l_{\mathsf{in}} : \mathcal{C}_{(l_{\mathsf{out}})}(c)$, where c is the body of p. We define the specification for p' as $\langle \mathsf{Pre}, \mathsf{annot}', \mathsf{Post} \rangle$, where $\mathsf{dom}(\mathsf{annot}') = \{l^b \mid l \in \mathsf{dom}(\mathsf{annot})\}$ and

$$\mathsf{annot}'(l^b) = \mathsf{annot}(l) \wedge C_l \bigwedge_{v \in \mathsf{dom}(S_l)} v = S_l(v)$$

and (S_l, C_l) is the result of the symbolic execution at the source program point labeled as l.

In order to deal with the ghost variables that appear in the final specification, the compilation of every statement that may introduce a ghost variable in the symbolic state must be followed by a **set** statement on that ghost variable.

Consider for instance the case of loop statements. Recall that for every variable x that may be modified by the loop, a **while** statement introduces a fresh ghost variable x^l in the symbolic state. Then, the VCG function is defined accordingly to deal with the introduction of these variables. In the case of a standard RTL verification, the compiler must introduce **set** statements in order to make the augmented specification valid. Similar criteria are used to define the compilation of conditional statements and function invocation. In Figure 9, x stands for every variable that may get modified by any of the two statements. In the case of function invocation, a stands for every array variable that the function may modify.

Consider a source procedure p with body c and specification $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post} \rangle$, and the RTL procedure $p' = l_{\mathsf{in}} : \mathcal{C}(c)$. From the definition of the non-optimizing compiler, one can show that the proof obligations in $\mathsf{PO}(p)$ are equivalent to the proof obligations in $\mathsf{po}(p')$. The only source of syntactic differences is due to the compilation of boolean expressions. For instance, for the statement c defined as **if** $b_1 \wedge b_2$ **then** c_1 **else** c_2, the function $\mathsf{VCG}(c, \varphi)$ returns a precondition of the form $(b_1 \wedge b_2 \Rightarrow \varphi_t) \wedge (\neg(b_1 \wedge b_2) \Rightarrow \varphi_f)$. If we compute the $\mathsf{wp}(l)$ for the subgraph that results from the compilation $l : \mathcal{C}_{(l')}(c)$ we get a precondition of the form $(b_1 \Rightarrow ((b_2 \Rightarrow \varphi_t) \wedge (\neg b_2 \Rightarrow \varphi_f))) \wedge (\neg b_1 \Rightarrow \varphi_f)$. Both assertions are trivially equivalent, but since proof obligations do not coincide syntactically, the tool implements a Coq tactic to transform the original certificates.

Example 6. The result of compiling the running example can be found in Figure 3. In this case, proof obligations coincide syntactically since every boolean condition of the source program is atomic.

Consider for instance one of the proof obligations for the code in Figure 3:

$$\mathsf{annot}'(l_4^b) \Rightarrow \left| \begin{array}{l} (\mathtt{j} < \mathtt{p} \Rightarrow \mathsf{annot}'(l_7^b)[{}^{\mathtt{k},\mathtt{c}}/_{\mathtt{k}^{l_4},\mathtt{c}^{l_4}}][{}^{[c:r_2 \mapsto 0]}/_{c}][{}^{r_1+\mathtt{j}}/_{r_2}][{}^{\mathtt{i}*\mathtt{p}}/_{r_1}][{}^{0}/_{\mathtt{k}}]) \wedge \\ (\neg \mathtt{j} < \mathtt{p} \Rightarrow \mathsf{Post}) \end{array} \right.$$

By definition of annot', and since r_1 and r_2 are fresh variables that do not appear in $\mathsf{annot}(l_7)$, this is equal to:

$$\overline{\mathsf{annot}}(l_4) \Rightarrow (\mathtt{j} < \mathtt{p} \Rightarrow \overline{\mathsf{annot}}(l_7)[{}^{\mathtt{k},\mathtt{c}}/_{\mathtt{k}^{l_4},\mathtt{c}^{l_4}}][{}^{[c:\mathtt{i}*\mathtt{p}+\mathtt{j} \mapsto 0]}/_{c}][{}^{0}/_{\mathtt{k}}]) \wedge (\neg \mathtt{j} < \mathtt{p} \Rightarrow \mathsf{Post})$$

which coincides with the proof obligation computed for the source program at label l_4.

5.2 Optimizing Compilation

In this section, we describe how the tool implements a certificate translator in the context of a basic but common class of program transformations. Standard compiler optimizations operates in a two-step basis. First, an automatic analysis gathers static information from the procedure. Then, based on this information, a second compiler step transforms the code preserving the original semantics.

We provide first a representation of an analysis framework and a characterization of the result of an analysis. Then, we explain how the tool returns, in addition to the analysis results, a certificate of its representation in the verification setting. We then show that for these optimizations one can define a certificate translator that avoids the growth of certificate size caused by the more general technique developed in previous work [3].

Certifying Analyzers. In general, program transformations not only modify proof obligations, but may render the original specification invalid. For the class of optimizations considered in this paper, the certificate translator must strengthen the specification with the result of the analysis that motivates the program transformation. Therefore, a certificate translation procedure automatically generates a certificate for the analysis result, and then merges it with the original certificate.

An analysis module is implemented in the tool as a data type A that represents properties on states, and a transfer function T. The transfer function T takes an instruction ins and an element of type A and returns a new element in A. To give an intuition, if a state satisfies a property a, then the state after the execution of a satisfies the property $T(\mathsf{ins}, a)$.

The result of the analysis, represented by a mapping S from program labels to elements in A, is computed by the tool by fixpoint approximation. For every label l, the property $S(l)$ characterizes the execution states that may reach that program point.

In order to certify the result of the analysis, the tool must define, for each analysis module, a function that maps every element a in A to its representation as a logical assertion. We omit, however, the application of this function and do not make the distinction between an element a in A and its logical representation.

From an analysis result S, the tool generates automatically a certificate for the procedure specification $\langle \mathsf{true}, S, \mathsf{true} \rangle$. First, the tool computes a set of verification conditions. Then, proof obligations are automatically discharged in Coq by application of the ring tactic, that solves equations on ring structures by associative and commutative rewriting.

Example 7. Consider for instance the analysis in which common-subexpression elimination is based. For the program in Figure 3, consider a labeling S, representing a result of the analysis such that $S(l_7^b) \doteq r_1 = \mathtt{i} * \mathtt{p} \wedge r_2 = \mathtt{i} * \mathtt{p} + \mathtt{j}$ and $S(l) = \mathsf{true}$ for $l \in \{l_2^b, l_4^b\}$. Ignoring for simplicity the symbolic execution process that strengthens invariants, the verification condition computed with specification $\langle \mathsf{true}, S, \mathsf{true} \rangle$ at label l_4^b has the form

$$\mathsf{true} \Rightarrow (\mathtt{j} < \mathtt{p} \Rightarrow \mathsf{true}) \wedge (\neg \mathtt{j} < \mathtt{p} \Rightarrow \mathtt{i} * \mathtt{p} = \mathtt{i} * \mathtt{p} \wedge \mathtt{i} * \mathtt{p} + \mathtt{j} = \mathtt{i} * \mathtt{p} + \mathtt{j})$$

At label l_7^b, the proof obligation is

$$r_1 = \mathtt{i} * \mathtt{p} \wedge r_2 = \mathtt{i} * \mathtt{p} + \mathtt{j} \Rightarrow (\mathtt{k} < \mathtt{n} \Rightarrow r_1 = \mathtt{i} * \mathtt{p} \wedge r_2 = \mathtt{i} * \mathtt{p} + \mathtt{j}) \wedge (\neg \mathtt{k} < \mathtt{n} \Rightarrow \mathsf{true})$$

The implemented Coq tactic can clearly discharge these verification conditions.

Optimization Based on Arithmetic Simplification. The class of optimizations considered in this section consists in the replacement of expressions in the instructions of a RTL procedure, without modifying its control-flow graph.

We can formalize the result of applying these class of optimizations to a procedure p, as a procedure p' such that for every label l, $p'[l] = p[l]$ or one of the following conditions holds:

- $p[l] = \mathbf{cjmp}\ x_1 \bowtie x_2\ l'$ and $p'[l] = \mathbf{cjmp}\ x_1' \bowtie x_2'\ l'$ for some variables or constants x_1, x_2, x_1' and x_2', or
- $p[l] = v := e$ and $p'[l] = v := e'$, or
- $p[l] = a[x] := e$ and $p'[l] = a[x'] := e'$, or
- $p[l] = \mathbf{invoke}\ f(\vec{x})$ and $p'[l] = \mathbf{invoke}\ f(\vec{x'})$, or
- $p[l] = \mathbf{return}\ x$ and $p'[l] = \mathbf{return}\ x'$.

for some atomic expressions x, x', e, e'.

$$\frac{}{\vdash e_1 \bowtie e_2 \sim e_1 \bowtie e_2} \qquad \frac{}{e_1 = e_1', e_2 = e_2' \vdash e_1 \bowtie e_2 \sim e_1' \bowtie e_2'}$$

$$\frac{}{e_1 = e_1' \vdash e_1 \bowtie e_2 \sim e_1' \bowtie e_2} \qquad \frac{}{e_2 = e_2' \vdash e_1 \bowtie e_2 \sim e_1 \bowtie e_2'}$$

$$\frac{\Gamma \vdash \phi \sim \phi'}{\Gamma \vdash \neg\phi \sim \neg\phi'} \qquad \frac{\Gamma \vdash \phi \sim \phi'}{\Gamma \vdash \forall v.\ \phi \sim \forall v.\ \phi'} \qquad \frac{\Gamma \vdash \phi \sim \phi' \qquad \Gamma' \vdash \psi \sim \psi'}{\Gamma, \Gamma' \vdash \phi \diamond \psi \sim \phi' \diamond \psi'} \diamond \in \{\wedge, \vee, \Rightarrow\}$$

Fig. 10. Definition of Structural Congruence

The main result of this characterization, is that the computation of verification conditions along the graph of the optimized program coincides in their logical structure with the original ones. That is, there is a syntactical correspondence between the logical formulae, up to substitution of equal expressions. This condition on a pair of assertions ϕ and ϕ' are formalized, for a set of equalities Γ, by the relation $\Gamma \vdash \phi \sim \phi'$ defined by the rules in Figure 10.

The tool relies on the fact that if p' is the result of applying arithmetic simplification to p, then, for every label l, there is a set of equations Γ' such that the relation $\Gamma' \vdash \mathsf{wpi}(p[l]) \sim \mathsf{wpi}(p'[l])$ holds.

Consider two logical formulae φ and φ', and assume they coincide modulo substitution of equalities in the set Γ, that is, $\Gamma \vdash \varphi \sim \varphi'$. It should be clear that from a set of certificates for the equalities in Γ, one can produce a certificate for the goal $\varphi \Rightarrow \varphi'$. The tool implements a tactic that produces this certificate, by traversing the logical structure of φ, and applying the Coq rewrite rule when needed, taking as input the certificate of the equations in Γ. In the following paragraphs, we explain how the tool obtains the certificates for the set of equations in Γ.

Consider the procedure p with specification $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post} \rangle$, and S a result of the analysis. Let p' be the result of transforming p by arithmetic simplification. Then, if the transformation is correct, for every label l there is a set Γ such that $\Gamma \vdash \mathsf{wp}_p(l) \sim \mathsf{wp}_{p'}(l)$, and such that $S(l)$ implies $e = e'$, for every $e = e'$ in Γ. Based on this result, the tool constructs a certificate of $S(l) \Rightarrow e = e'$, for every label l, and equality $e = e'$ in Γ, where Γ is such that $\Gamma \vdash \mathsf{wp}_p(l) \sim \mathsf{wp}_{p'}(l)$. It does so by relying on the Coq tactic ring. By composing these results the tool generates a certificate for the goal $S(l) \Rightarrow \mathsf{wp}_p(l) \Rightarrow \mathsf{wp}_{p'}(l)$ for every label l.

Then, the tool strengthens the original specification with the result of the analysis. The resulting specification $\langle \mathsf{Pre}, \mathsf{annot} \wedge S, \mathsf{Post} \rangle$ will be used also as the specification of the transformed program. To transform the certificates according to the new specification, the analysis is required to produce a certificate for its results, as explained before. Then, a Coq tactic implements a transformation that merges the certificate for the analysis with the original certificate. As a result, we have, for every $l \in \mathsf{annot}$, a proof for $\mathsf{annot}(l) \wedge S(l) \Rightarrow \mathsf{wpi}_p(l : p[l])$.

The tool then generates a certificate for the proof obligation corresponding to the transformed program $\mathsf{annot}(l) \wedge S(l) \Rightarrow \mathsf{wpi}_{p'}(l : p'[l])$. As mentioned above, the tool implements a Coq tactic, combining the ring tactic with rewriting of

expressions, to discharge the goal $\mathsf{annot}(l) \Rightarrow \mathsf{wpi}_p(l\!:\!p[l]l) \Rightarrow \mathsf{wpi}_{p'}(l\!:\!p'[l])$. The final certificate is then created by composition of the latter certificate and the original certificate $\mathsf{annot}(l) \Rightarrow \mathsf{wpi}_p(l\!:\!p[l])$.

Example 8 (Matrix Multiplication: Common-subexpression elimination). From the analysis result computed in Example 7, one can apply common-subexpression elimination on the RTL code shown in Figure 3. In the figure, the optimization replaces the original instructions, in the white boxes, by the optimized instructions inside the gray boxes. The assignments are simplified taking advantage of the conditions $r_1 = \mathtt{i} * \mathtt{p}$ and $r_2 = \mathtt{i} * \mathtt{p} + \mathtt{j}$.

If we compute the proof obligation at label l_7^b in the RTL program of Figure 3 we get something of the form:

$$\mathsf{annot}'(l_7^b) \Rightarrow (\mathtt{k} < \mathtt{n} \Rightarrow \mathsf{annot}'(l_7^b)[^{[\mathtt{c}:e_1 \mapsto e_2]}/_{\mathtt{c}}][^{[\mathtt{k}+1}/_{\mathtt{k}}]) \wedge \varphi$$

for some φ, where e_1 stands for $\mathtt{i} * \mathtt{p} + \mathtt{j}$ and e_2 stands for the expression $\mathtt{c}[\mathtt{i} * \mathtt{p} + \mathtt{j}] + \mathtt{a}[\mathtt{i} * \mathtt{n} + \mathtt{k}] * \mathtt{b}[\mathtt{k} * \mathtt{p} + \mathtt{j}]$. The corresponding proof obligation at label l_7^b in the optimized program of Figure 3 has the form:

$$\mathsf{annot}'(l_7^b) \Rightarrow (\mathtt{k} < \mathtt{n} \Rightarrow \mathsf{annot}'(l_7^b)[^{[\mathtt{c}:e_1' \mapsto e_2']}/_{\mathtt{c}}][^{[\mathtt{k}+1}/_{\mathtt{k}}]) \wedge \varphi$$

where e_1' stands for r_2 and e_2 stands for $\mathtt{c}[r_2] + \mathtt{a}[\mathtt{i} * \mathtt{n} + \mathtt{k}] * \mathtt{b}[\mathtt{k} * \mathtt{p} + \mathtt{j}]$. Clearly, the two formulae differ only on the substitution of the terms e_1 by e_1', and e_2 by e_2'. The result of the analysis $r_2 = \mathtt{i} * \mathtt{p} + \mathtt{j}$ can prove the equations $e_1 = e_1'$ and $e_2 = e_2'$. Therefore, by strengthening the original annotation with the result of the analysis, the tool can generate a certificate for:

$$S(l_7^b) \wedge \mathsf{annot}'(l_7^b) \Rightarrow (\mathtt{k} < \mathtt{n} \Rightarrow \mathsf{annot}'(l_7^b)[^{[\mathtt{c}:e_1' \mapsto e_2']}/_{\mathtt{c}}][^{[\mathtt{k}+1}/_{\mathtt{k}}] \wedge \varphi$$

Redundant Conditional Elimination. The tool also implements redundant conditional elimination, that replaces conditional instruction **cjmp** l' by a non-conditional jump **jmp** l' (or **jmp** $l+1$) if we can statically infer that a condition (or its negation) is always valid.

After the application of this transformation, for an annotated label l such that $p[l] = \mathbf{cjmp}\ v_1 \bowtie v_2\ l'$, the tool must provide, from a certificate of the form

$$\mathsf{annot}(l) \Rightarrow ((v_1 \bowtie v_2) \Rightarrow \mathsf{wp}(l')) \wedge (\neg(v_1 \bowtie v_2) \Rightarrow \mathsf{wp}(l+1))$$

a certificate for the transformed proof obligation: $\mathsf{annot}(l) \Rightarrow \varphi$ where φ is equal to $\mathsf{wp}(l')$ or $\mathsf{wp}(l+1)$ depending on which of the conditions $v_1 \bowtie v_2$ or $\neg(v_1 \bowtie v_2)$ is always valid. The transformation is restricted to the case in which the tool can automatically prove the condition $v_1 \bowtie v_2$ true or false. Therefore, it can straightforwardly generate a certificate for $\mathsf{wp}(l')$ or $\mathsf{wp}(l+1)$ from the certificate of the original proof obligation.

Dead Code Elimination. Another optimization implemented by the tool consists in removing unreachable program points from a sequence of labeled RTL

instructions. This optimization is useful for the elimination of redundant conditional jumps. As a side effect, proof obligations are not modified. Instead, some of them may disappear if the optimization removes an unreachable annotated program label. Therefore, the new set of proof obligations is a subset of the original one, and no certificate transformation is needed.

6 Experimental Results

We have experimented with several examples to estimate the impact of certificate transformation in the size of the final certificates. Most of the examples are relatively small, but specifically suited to test the optimizations covered by the tool. To describe the size of certificates, we have considered the number of nodes of the tree structure that represents each Coq λ-term. In average, from the original certificate we have obtained a slight reduction on the certificate for the non-optimized RTL code. The size of the certificate of the result of the analysis is on average 0.43 times the size of the original certificate. Merging the RTL certificates with the certificates of the analysis yields certificates that are almost three times the size of the original certificate. Certificate translation for common-subexpression elimination increases the previous certificate by a factor of 1.46 on average. In total, the final certificates are on average approximately 4 times the size of the original certificates.

We show in the following table a more detailed analysis of the certificate size for the multiplication matrix example.

PO	Source	RTL	Analysis	Merge	CSE
Pre	2922	2960	109	7615	7615
annot(l_2)	8746	8272	138	23276	23276
annot(l_4)	33232	32418	261	86962	86962
annot(l_7)	95195	93907	229	178012	253575

The table shows each certificate size for each step of the compilation. The second column represents the original certificate discharged interactively by the tool user. The third column represents the certificate size after non-optimizing compilation. The fourth column represents the size of the certificate of the analysis result, automatically generated by the tool. The fifth column represents the certificate size after merging the certificate for the RTL program and the certificate for the result of the analysis. Finally, the last column show the certificate size for the optimized program.

Other optimizations, such as redundant conditional elimination and dead code elimination, reduce the size of certificates, since verification conditions are always simplified.

7 Concluding Remarks

This paper reports on a prototype implementation of certificate translation for a fragment of the C language to a RTL language. Although the prototype is modest

with respect to state-of-the-art verification tools, it brings a practical perspective on certificate translation and complements the theoretical developments.

Related work. We refer to [3] for a more comprehensive account of related work, and only focus on closely related work. Most of the practical attempts to transfer evidence from source code to lower levels are based on type systems: type-preserving compilers [6,4] aim at translating typable source programs into typable lower level programs; sometimes, they also generate typing information that can be used to make type checking of compiled code more efficient. Certifying compilers [7] aim at translating typable source programs into provably correct lower level programs: they generate logical annotations from the typing derivations, and a certificate for the annotated programs. However, there are only few practical efforts to transform provable source code programs into provable lower level programs, and to generate certificates of the latter. Pavlova [9] implements a certificate translator for a non-optimizing compiler from Java to the JVM. Nordio *et al* [8] formalize non-optimizing proof-transforming compilers from Eiffel to MSIL.

Future work. Our prototype is still in a preliminary stage, and there are many opportunities for improvements and extensions. A first improvement would be to reduce certificate size. We envision two complementary efforts: firstly, one can reduce the size of certificates for source programs using hybrid methods, combining program analysis and program verification. Secondly, one can reduce the growth of certificates during strengthening by symbolic execution and by certificate translation using methods to slice unused parts of the specification; some work in this direction is reported in [10].

Extensions may also be pursued in two complementary efforts. Firstly, one may extend the compilation scheme from RTL to assembly, and provide a corresponding certificate translator. Compiling RTL to assembly involves defining calling conventions, linearizing code, and spilling. We believe that writing certificate translators for these transformations is feasible, and no more difficult than dealing with the optimizations studied here. Secondly, one may extend language coverage, and aim to cover increasingly large fragments of C.

An ambitious goal, that encompasses many of these directions, would be to build a certificate translator that uses Frama-C [5] as a front end.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
2. Barthe, G., Kunz, C.: Certificate translation in abstract interpretation. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 368–382. Springer, Heidelberg (2008)
3. Barthe, G., Grégoire, B., Kunz, C., Rezk, T.: Certificate translation for optimizing compilers. ACM Transactions on Programming Languages and Systems 31(5), 18:1–18:45 (2009)

4. Chen, J., Hawblitzel, C., Perry, F., Emmi, M., Condit, J., Coetzee, D., Pratikakis, P.: Type-preserving compilation for large-scale optimizing object-oriented compilers. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, pp. 183–192. ACM, New York (2008)
5. Monate, B., Correnson, L.: Frama-C, `http://frama-c.cea.fr`
6. Morrisett, G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. ACM Transactions on Programming Languages and Systems 21(3), 527–568 (1999); Expanded version of a paper presented at POPL 1998 (1998)
7. Necula, G.C.: Compiling with Proofs. PhD thesis, Carnegie Mellon University, Available as Technical Report CMU-CS-98-154 (October 1998)
8. Nordio, M., Müller, P., Meyer, B.: Proof-transforming compilation of eiffel programs. In: Paige, R. (ed.) TOOLS-EUROPE. LNBIP. Springer, Heidelberg (2008)
9. Pavlova, M.: Java bytecode verification and its applications. Thèse de doctorat, spécialité informatique, Université Nice Sophia Antipolis, France (January 2007)
10. Seo, S., Yang, H., Yi, K., Han, T.: Goal-directed weakening of abstract interpretation results. ACM Transactions on Programming Languages and Systems 29(6), 39:1–39:39 (2007)