Certificate Translation for Optimizing Compilers

Abstract

Source code verification is increasingly been used to improve the reliability and security of software, in particular for embedded systems and smart cards. The appropriateness of source code verification relies on two major hypotheses: first, the source code is available to the code consumer or the code consumer trusts the code producer. Second, the code consumer trusts the compiler. The first hypothesis is valid for domains such as embedded systems and smart cards, thus for these application domains one is left to show that one can trust the compiler. There are several means to ensure trust in the compiler, including translation validation and certified compilation if we focus on preservation of semantics, or certifying compilation if we focus on guaranteeing basic safety policies.

However, availability of source code cannot be realistically assumed in the context of mobile code, or even in the context of midlets for cell phones. In such domains, there is a need for methods to bring the benefits of source code verification to code consumers. Building upon ideas of Proof Carrying Code, in which programs are required to carry formal proofs, a.k.a. certificates, that they obey to a security policy, we propose *certificate translation* as a means to transform certificates of source code into certificates of executable code. We outline the general principles of certificate translation, and instantiate them in the context of an optimizing compiler from a simple imperative language to an intermediate RTL language.

The resulting PCC architecture can be used in numerous application domains including mobile code and midlets.

1 Introduction

Proof Carrying Code (PCC) [8] provides a means to establish trust in a mobile code infrastructure, by requiring that mobile code is sent along with a formal proof that it adheres to a security policy agreeable by the code consumer. A typical PCC architecture comprises at least the following items: a formalism for specifying policies, a verification condition generator (VCGen) that generates proof obligations from the program and the policy, a formal representation of proofs, known as certificates, and a certificate checker. While PCC does not make any assumption on the way certificates are generated, the prominent approach to certificate generation is *certifying compilation* [9, 4], which generates certificates automatically for safety policies such as memory safety or type safety. Yet, experience has shown that powerful verification technology, including possibly interactive verification, is often required, both for basic safety policies such as exception safety, and for more advanced security policies such as non-interference and resource control. Thus, there is a need for generating certificates from program verification environments.

However, verification environments target high-level languages whereas code consumers require certificates to bring correctness guarantees for compiled programs. One naive approach to solve the mismatch is to develop verification environments for compiled programs, but the approach has major drawbacks, especially in the context of interactive verification: one looses the benefit of reasoning on a structured language, and the verification effort is needlessly duplicated, as each program must be verified once per target language and compiler. A better solution is to develop methods for transferring evidence from source code to compiled programs, so that verification can be performed as usual with existing tools, and that code is only proved once (which could be summarized by the slogan: write *and prove* once, *verify and* run everywhere).

In order to bring the benefits of source code verification to the code consumers, we propose *certificate translation* as a mechanism that transforms certificates of source programs into certificates of compiled programs. Given a compiler $[\![.]\!]$, a function $[\![.]\!]_{\mathsf{spec}}$ to transform specifications, and certificate checkers (expressed as a ternary relation "c is a certificate that P adheres to ϕ", written

$c : P \models \phi$), a certificate translator is a function $[\![.]\!]_{\text{cert}}$ such that for all source-level programs p, policies ϕ, and certificates c,

$$c : p \models \phi \quad \Longrightarrow [\![c]\!]_{\text{cert}} : [\![p]\!] \models [\![\phi]\!]_{\text{spec}}$$

Certificate translation is tightly bound to the infrastructures used for verification and compilation. W.r.t. verification infrastructures, we focus on approaches based on VCGens, because they are used in typical PCC architectures and many verification environments. W.r.t. compilation infrastructures, we focus on an optimizing compiler from a simple imperative language to a Register Transfer Language (RTL). The compiler proceeds in two phases: during the first phase, imperative programs are translated into RTL programs in a standard fashion. During the second phase, common program optimizations are performed. For each step, we build an appropriate certificate translator, and combine them to obtain a certificate translator for the complete compilation process.

Building a certificate translator for the non-optimizing compiler is easy, because non-optimizing compilation preserves proof obligations, hence certificates for source code programs can be reused at RTL level, i.e. $[\![]\!]_{\text{spec}}$ and $[\![]\!]_{\text{cert}}$ are the identity function. Dealing with optimizations may be more intricate because:

- for optimizations that perform arithmetic simplifications (such as constant propagation), certificate translators need to be given evidence that justifies the optimization. In order to provide such evidence, we introduce *certifying analyzers*, which express the results of the analysis in the logic of the PCC architecture, and produce a certificate of the analysis for each program. Then, we allow certificate translators to take as arguments both the certificate of the original program, and the certificate produced by the certifying analyzer.

- some program optimizations that eliminate instructions without computational role (assignments to dead registers, `nop` instructions) may also eliminate information that is required to prove the program correct. For example, dead register elimination may eliminate registers that occur in intermediate assertions of the program. Then, it is possible that proof obligations containing assertions with dead registers cannot be proved since all hypotheses about these registers have been lost. As one cannot assume in general that dead registers do not appear in assertions, one must modify the existing optimizations in such a way that they are compatible with programs being annotated. Thus, in order to define a certificate translator for dead register elimination, we are led to propose a different kind of transformation that performs simultaneously dead variable elimination in instructions and in assertions.

Summarizing, our main contributions are the introduction of certificate translation as a means to bring the benefits of interactive source code verification to code consumers, and in order to cover standard optimizations, the development of certifying analyzers and assertions-aware analyzes. The resulting architecture, which is presented in Figure 1, significantly extends the scope of PCC to complex security policies.

In summary, our main contributions are:

- the introduction of certificate translation, which enables to bring the benefits of interactive source code verification to code consumers and significantly extends the scope of PCC to complex security policies; we have applied certificate translation to constant propagation, loop induction, dead register elimination, common subexpression elimination, inlining, register allocation, loop unrolling, unreachable code elimination, and non-optimizing compilation. For space constraints, only the first three optimizations are shown in the paper. Others will be described in the full version of the article;

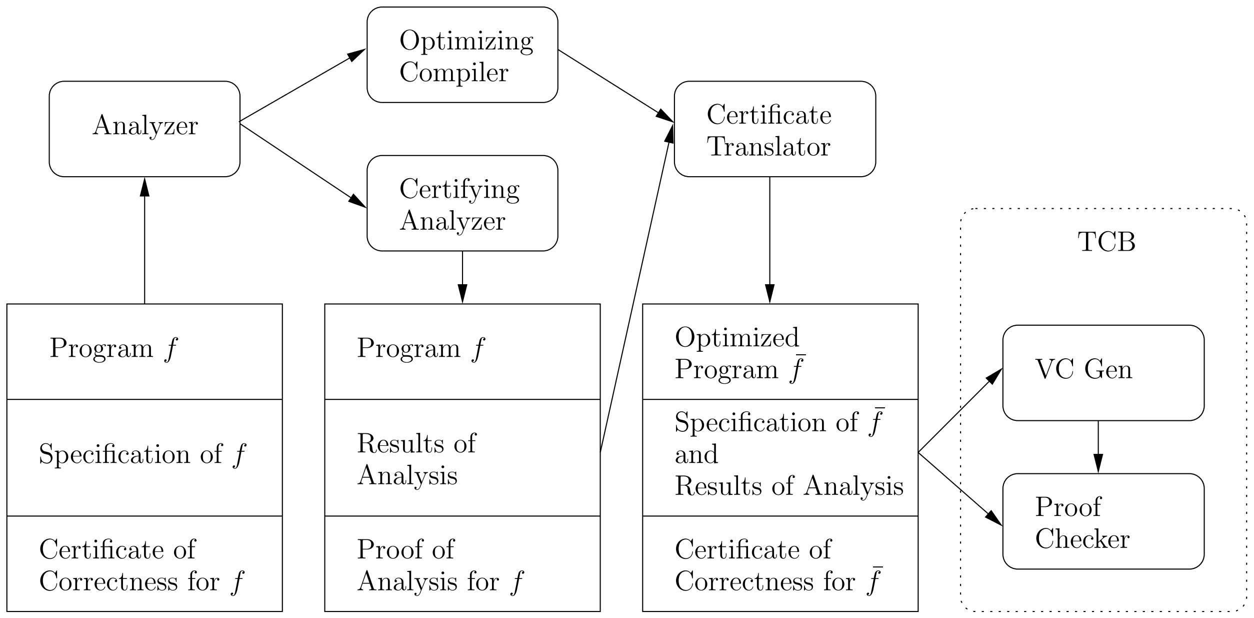
Optimizing Compiler
Analyzer
Certificate Translator
Certifying Analyzer
TCB
Program f
Specification of f
Certificate of Correctness for f
Program f
Results of Analysis
Proof of Analysis for f
Optimized Program $\bar{f}$
Specification of $\bar{f}$ and Results of Analysis
Certificate of Correctness for $\bar{f}$
VC Gen
Proof Checker

Figure 1: Overall picture of certificate translation

- in order to make certificate translation feasible, we introduce certifying analyzers (their relevance escapes the scope of certificate translation, and their in-depth study will be developed elsewhere), and the reformulation of existing optimizations such as dead register elimination to deal with annotated programs.

Contents Sections 2 introduces our programming language RTL and our PCC infrastructure. Section 3 provides a high-level overview of the principles and components that underline certificate translation, whereas Section 4 describe certificate translators for several standard optimizations (at RTL level), and for non-optimizing compilation from an imperative language to RTL. Section 5 discusses potential application scenarios for certificate translation. We conclude in Section 6 with related and future work.

2 Setting

2.1 RTL language

Our language RTL (Register Transfer Language) is a low-level, side-effect free, language with conditional jumps and function calls, extended with annotations drawn from a suitable assertion language. The choice of the assertion language does not affect our results, provided assertions are closed under the connectives and operations that are used by the verification condition generator.

The syntax of expressions, formulas and RTL programs (suitably extended to accommodate certificates, see Subsection 2.3), is shown in Figure 2, where $n \in \mathbb{N}$ and $r \in \mathcal{R}$, with $\mathcal{R}$ an infinite set of register names. We let ϕ and ψ range over assertions.

A program p is defined as a function from RTL function identifiers to function declarations. We assume that every program comes equipped with a special function, namely main, and its declaration. A declaration F for a function f includes its formal parameters $\vec{r}$, a precondition φ, a

comparison	$\lhd$	::=	$< \mid \leq \mid = \mid \geq \mid >$
expressions	e	::=	$n \mid r \mid -e \mid e+e \mid e-e \mid e*e \mid \dots$
assertions	ϕ	::=	$\top \mid e \lhd e \mid \neg\phi \mid \phi \wedge \phi \mid \phi \Rightarrow \phi \mid \forall r,\ \phi \mid \dots$
	$\star$	::=	$< \mid \leq \mid = \mid \geq \mid >$
comparisons	cmp	::=	$r \star r \mid r \star n$
operators	op	::=	$n \mid r \mid \mathsf{cmp} \mid -r \mid r+r \mid n+r \mid r-r \mid n-r \mid r*r \mid n*r \mid \dots$
instr. desc.	ins	::=	$r_d := \mathsf{op},\ L \mid r_d := f(\vec{r}),\ L \mid \mathsf{cmp}\ ?\ L_t : L_f \mid \mathsf{return}\ r \mid \mathsf{nop},\ L$
instructions	I	::=	$(\phi,\ \mathsf{ins}) \mid \mathsf{ins}$
graph	G	::=	$L \mapsto I$
	$\vec{\Lambda}$	::=	$L \mapsto \lambda$
fun. decl	F	::=	$\{\vec{r};\ \varphi;\ G;\ \psi;\ \lambda;\ \vec{\Lambda}\}$
program	p	::=	$f \mapsto F$

Figure 2: Syntax of RTL

(closed) graph code G, a postcondition ψ, a certificate λ, and a function $\vec{\Lambda}$ from reachable labels to certificates (the notion of reachable label is defined below). For clarity, we often use in the sequel a subscript f for referring to elements in the declaration of a function f, e.g. the graph code of a function f as G_f.

Formal parameters are a list of registers from the set $\mathcal{R}$, which we suppose to be local to f. For specification purposes, we introduce for each register r in $\vec{r}$ a (pseudo-)register r^*, not appearing in the code of the function, and which represents the initial value of a register declared as formal parameter. We let $\vec{r^*}$ denote the set $\{r^* \in \mathcal{R} \mid r \in \vec{r}\}$. We also introduce a (pseudo-)register res, not appearing in the code of the function, and which represents the result or return value of the function. The annotations φ and ψ provide the specification of the function, and are subject to well-formedness constraints. The precondition of a function f, denoted by function $\mathsf{pre}(f)$, is an assertion in which the only registers to occur are the formal parameters $\vec{r}$; in other words, the precondition of a function can only talk about the initial values of its parameters. The postcondition of a function f, denoted by function $\mathsf{post}(f)$, is an assertion in which the only registers to occur are res and registers from $\vec{r^*}$; in other words, the postcondition of a function can only talk about its result and the initial values of its parameters.

A graph code of a function is a partial function from labels to instructions. We assume that every graph code includes a special label, namely L_{sp}, corresponding to the starting label of the function, i.e. the first instruction to be executed when the method is called. Given a function f and a label L in the domain of its graph code, we will often use $f[L]$ instead of $G_f(L)$, i.e. application of code graph of f to L.

Instructions are either instruction descriptors or pairs consisting of an annotation and an instruction descriptor. An instruction descriptor can be an assignment, a function call, a conditional jump or a return instruction. Operations on registers are those of standard processors, such as movement of registers or values into registers $r_d := r$, and arithmetic operations between registers or between a register and a value. Furthermore, every instruction descriptor carries explicitly its successor(s) label(s); due to this mechanism, we do not need to include unconditional jumps, i.e. "goto" instructions, in the language. Immediate successors of a label L in the graph of a function f are denoted by the set $\mathsf{succ}_f(L)$. We assume that the graph is closed; in particular, if L is associated with a return instruction, $\mathsf{succ}_f(L) = \emptyset$.

The operational semantics of RTL is standard, and thus omitted. In particular, neither proofs nor assertions interfere with the semantics.

$$\begin{array}{rcll}
\mathsf{vcg}_f(L) & = & \phi & \text{if } G_f(L) = (\phi,\ \mathsf{ins}) \\
\mathsf{vcg}_f(L) & = & \mathsf{vcg}^{\mathtt{id}}_f(\mathsf{ins}) & \text{if } G_f(L) = \mathsf{ins} \\
\mathsf{vcg}^{\mathtt{id}}_f(r_d := \mathsf{op},\ L) & = & \mathsf{vcg}_f(L)\{r_d \leftarrow \langle \mathsf{op} \rangle\} & \\
\mathsf{vcg}^{\mathtt{id}}_f(r_d := g(\vec{r}),\ L) & = & \mathsf{pre}(g)\{\vec{r}_g \leftarrow \vec{r}\} \wedge (\forall res.\ \mathsf{post}(g)\{\vec{r}^{\,*}_g \leftarrow \vec{r}\} \Rightarrow \mathsf{vcg}_f(L)\{r_d \leftarrow res\}) & \\
\mathsf{vcg}^{\mathtt{id}}_f(\mathsf{cmp}\ ?\ L_t : L_f) & = & ((\langle \mathsf{cmp} \rangle \Rightarrow \mathsf{vcg}_f(L_t)) \wedge (\neg \langle \mathsf{cmp} \rangle \Rightarrow \mathsf{vcg}_f(L_f)) & \\
\mathsf{vcg}^{\mathtt{id}}_f(\mathsf{return}\ r) & = & \mathsf{post}(f)\{\mathsf{res} \leftarrow r\} & \\
\mathsf{vcg}^{\mathtt{id}}_f(\mathsf{nop},\ L) & = & \mathsf{vcg}_f(L) &
\end{array}$$

Figure 3: Verification condition generator

2.2 Verification condition generator

Verification condition generators (VCGens) are partial functions that compute, from a partially but sufficiently annotated program, a fully annotated program in which all labels of the program have an explicit precondition attached to them. Programs in the domain of the VCGen function are called well annotated and can be characterized by an inductive definition. Our definition is decidable and does not impose any specific structure on programs.

Definition 2.1 (Well annotated program).

- *A label L in a function f reaches annotated labels, if its associated instruction contains an assertion, or if its associated instruction is a* return *(in that case the annotation is the post condition), or if all its immediate successors reach annotated labels:*

$$\begin{array}{l}
f[L] = (\phi,\ \mathsf{ins}) \Rightarrow L \in \mathsf{reachAnnot}_f \\
f[L] = \mathsf{return}\ r \Rightarrow L \in \mathsf{reachAnnot}_f \\
(\forall L' \in \mathsf{succ}_f(L), L' \in \mathsf{reachAnnot}_f) \Rightarrow L \in \mathsf{reachAnnot}_f
\end{array}$$

- *A function f is well annotated if every reachable point from starting point L_{sp} reaches annotated labels. A program p is well annotated if all its functions are well annotated.*

Given a well-annotated program, one can generate an assertion for each label, using the assertions that were given or previously computed for its successors. This assertion represents the precondition that an initial state before the execution of the corresponding label should satisfy for the function to terminate in a state satisfying its postcondition.

The computation of the assertions for the labels of a function f is performed by a function vcg_f, and proceeds in a modular way, using annotations from the function f under consideration, as well as the preconditions and postconditions of functions called by f. The definition of $\mathsf{vcg}_f(L)$ proceeds by case: if L points to an instruction that carries an assertion ϕ, then $\mathsf{vcg}_f(L)$ is set to ϕ; otherwise, $\mathsf{vcg}_f(L)$ is computed by the function $\mathsf{vcg}^{\mathtt{id}}_f$.

The formal definitions of vcg_f and $\mathsf{vcg}^{\mathtt{id}}_f$ are given in Figure 3, where $e\{r \leftarrow e'\}$ stands for substitution of all occurrences of register r in expression e by e'. The definition of $\mathsf{vcg}^{\mathtt{id}}_f$ is standard for assignment and conditional jumps, where $\langle \mathsf{op} \rangle$ and $\langle \mathsf{cmp} \rangle$ is the obvious interpretation of operators in RTL into expressions in the language of assertions. For a function invocation, $\mathsf{vcg}^{\mathtt{id}}_f(r_d := g(\vec{r}),\ L)$ is defined as a conjunction of the precondition in the declaration of g where formal parameters are replaced by actual parameters, and of the assertion $\forall res.\ \mathsf{post}(g)\{\vec{r}^{\,*}_g \leftarrow \vec{r}\} \Rightarrow \mathsf{vcg}_f(L)\{r_d \leftarrow res\}$.

$$
\begin{array}{lll}
\mathsf{axiom} & : \mathcal{P}(\Gamma; A; \Delta \vdash A) & \\
\mathsf{ring} & : \mathcal{P}(\Gamma \vdash n_1 = n_2) & \text{if } n_1 = n_2 \text{ is a ring equality} \\
& & \\
\mathsf{intro}_{\Rightarrow} & : \mathcal{P}(\Gamma; A \vdash B) \to \mathcal{P}(\Gamma \vdash A \Rightarrow B) & \\
\mathsf{elim}_{\Rightarrow} & : \mathcal{P}(\Gamma \vdash A \Rightarrow B) \to \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma \vdash B) & \\
& & \\
\mathsf{elim}_{=} & : \mathcal{P}(\Gamma \vdash e_1 = e_2) \to \mathcal{P}(\Gamma \vdash A\{r \leftarrow e_1\}) \to \mathcal{P}(\Gamma \vdash A\{r \leftarrow e_2\}) & \\
& & \\
\mathsf{subst} & : \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma\{r \leftarrow e\} \vdash A\{r \leftarrow e\}) &
\end{array}
$$

Figure 4: Proof Algebra (excerpts)

The second conjunct permits that information in $\mathsf{vcg}_f(L)$ about registers different from r_d is propagated to other preconditions. In the remaining of the paper, we shall abuse notation and write $\mathsf{vcg}_f^{\mathtt{id}}(\mathsf{ins})$ or $\mathsf{vcg}_f^{\mathtt{id}}(L)$ instead of $\mathsf{vcg}_f^{\mathtt{id}}(\mathsf{ins}, L')$ if $f[L] = \mathsf{ins}, L'$ and neither L' or ins are relevant to the context.

The verification condition generator is sound, in the sense that if the program p is called with registers set to values that verify the precondition of the function main, and p terminates normally, then the final state will verify the postcondition of main. Soundness is proved first for one step of execution, and then extended to execution traces by induction on the length of the execution.

2.3 Certificates, proof algebras and certified programs

Certificates provide a formal representation of proofs, and are used to verify that the proof obligations generated by the VCGen hold. For the purpose of certificate translation, we do not need to commit to a specific format for certificates. Instead, we assume that certificates are closed under specific operations on certificates, which are captured by an abstract notion of proof algebra.

Recall that a judgment is a pair consisting of a list of assertions, called context, and of an assertion, called goal. Then a *proof algebra* is given by a set-valued function $\mathcal{P}$ over judgments, and by a set of operations, all implicitly quantified in the obvious way. The operations are standard (selected operations are given in Figure 4) , to the exception perhaps of the substitution operator that allows to substitute selected instances of equals by equals, and of the operator ring, which establishes all ring equalities that will be used to justify the optimizations.

As a result of working at an abstract level, we do not provide an algorithm for checking certificates. Instead, we take $\mathcal{P}(\Gamma \vdash \phi)$ to be the set of valid certificates of the judgment $\Gamma \vdash \phi$. In the sequel, we write $\lambda : \Gamma \vdash \phi$ to express that λ is a valid certificate for $\Gamma \vdash \phi$, and use proof as a synonym of valid certificate.

Finally, we define a certified program as one whose functions are certified, i.e. carry valid certificates for the proof obligations attached to them.

Definition 2.2 (Certified Program).

- *A function f with declaration $\{\vec{r};\ \varphi;\ G;\ \psi;\ \lambda;\ \vec{\Lambda}\}$ is certified if:*
  - *λ is a proof of $\vdash \varphi \Rightarrow \mathsf{vcg}_f(L_{\mathsf{sp}})\{\vec{r}^* \leftarrow \vec{r}\}$*
  - *$\vec{\Lambda}(L)$ is a proof of $\vdash \phi \Rightarrow \mathsf{vcg}_f^{\mathtt{id}}(\mathsf{ins})$ for all reachable labels L in f such that $f[L] = (\phi,\ \mathsf{ins})$.*
- *A program is certified if all its functions are.*

3 Principles of certificate translation

In a classical compiler, transformations operate on unannotated programs, and are performed in two phases: first, a data flow analysis gathers information about the program. Then, on the basis of this information, (blocks of) instructions are rewritten. In certificate translation, we may also rewrite assertions, and we must also generate certificates for the optimized programs.

For simplicity, assume for a moment that the transformation $\bar{\cdot}$ does not modify the set of reachable annotated labels. Then certificate translation may be achieved by defining two functions:

$$\begin{array}{l} T_0\colon\ \mathcal{P}(\vdash \mathsf{pre}(f) \Rightarrow \mathsf{vcg}_f^{\mathtt{id}}(L_{\mathsf{sp}})) \to \mathcal{P}(\vdash \mathsf{pre}(\bar{f}) \Rightarrow \mathsf{vcg}_{\bar{f}}^{\mathtt{id}}(L_{\mathsf{sp}})) \\ T_\lambda\colon\ \forall L,\ \mathcal{P}(\vdash \phi_L \Rightarrow \mathsf{vcg}_f^{\mathtt{id}}(L)) \to \mathcal{P}(\vdash \bar{\phi}_L \Rightarrow \mathsf{vcg}_{\bar{f}}^{\mathtt{id}}(L)) \end{array}$$

where $\bar{f}$ is the optimized version of f, and ϕ_L is the original assertion at label L, and $\bar{\phi}_L$ is the rewritten assertion at label L. Here the function T_0 transforms the proof that the function precondition implies the verification condition at program point L_{sp} for f into a proof of the same fact for $\bar{f}$, and likewise, the function T_λ transforms for each reachable annotated label L the proof that its annotation implies the verification condition at program point L for f into a proof of the same fact for $\bar{f}$.

Certificate translators fall in one of the following categories:

- PPO/IPO (Preservation/Instantiation of Proof Obligations): PPO deals with transformations for which the annotations are not rewritten, and where the verification conditions coincide, i.e. $\bar{\phi}_L$ is equal to ϕ_L and $\mathsf{vcg}_f^{\mathtt{id}}(L)$ is equal to $\mathsf{vcg}_{\bar{f}}^{\mathtt{id}}(L)$ for all reachable labels L. This category covers transformations such as non-optimizing compilation and unreachable code elimination. IPO deals with transformations where the annotations and proof obligations for $\bar{f}$ are instances of annotations and proof obligations for f, thus certificate translation amounts to instantiating certificates. This category covers dead register elimination and register allocation. For transformations in this category, the certificate for $\bar{f}$ is of similar size as the certificate of f;

- SCT (Standard Certificate Translation): SCT deals with transformations for which the annotations are not rewritten, but where the verification conditions do not coincide. This category covers transformations such as loop unrolling and in-lining. For transformations in this category, the growth in the certificate size is similar to the growth of the program size incurred by the transformation;

- CTCA (Certificate Translation with Certifying Analyzers): CTCA deals for which the annotations need to be rewritten, and for which certificate translation relies on having certified previously the analysis used by the transformation, using *certifying analyzers* that produce a certificate that the analyzer is correct on the source program. This category covers constant propagation, common subexpression elimination, loop induction, and other optimizations that rely on arithmetic. Studying certificate size for such transformations is left for future work.

In the remaining of this section, we justify the need for certifying analyzers, and show how they can be used for transformations in CTCA. The following example, which will be used as a running example in the subsequent paragraphs, illustrates the need for certifying analyzers.

Example 3.1. *Let f be a certified function with specification:* $\mathsf{pre}(f) \equiv \top$ *and* $\mathsf{post}(f) \equiv res \geq b{*}n$*,* *where b and n are constants. The graph code of f and its proofs obligations are given by:*

$$\begin{array}{lll} L_1: & & r_i := 0,\ L_2 \\ L_2: & \xi, & r_1 := b + r_i,\ L_3 \\ L_3: & & r_i := c + r_i,\ L_4 \\ L_4: & & r_j := r_1 * r_i,\ L_5 \\ L_5: & \varphi, & (r_i = n)\ ?\ L_6 : L_3 \\ L_6: & & \mathsf{return}\ r_j \end{array} \qquad \begin{array}{l} \vdash \top \Rightarrow 0 \geq 0 \\ \vdash \xi \Rightarrow \phi \\ \vdash \varphi \Rightarrow (r_i = n \Rightarrow \phi_t \wedge r_i \neq n \Rightarrow \phi_f) \end{array}$$

where, $\xi \triangleq 0 \leq r_i$ and $\varphi \triangleq r_j = r_1 * r_i \wedge r_1 \geq b \wedge r_i \geq 0$ and

$$\begin{array}{lcl} \phi & \triangleq & (b + r_i) * (c + r_i) = (b + r_i) * (c + r_i) \\ & & \wedge\ b + r_i \geq b \wedge c + r_i \geq 0 \\ \phi_t & \triangleq & r_j \geq b * n \\ \phi_f & \triangleq & r_1 * (c + r_i) = r_1 * (c + r_i) \wedge r_1 \geq b \wedge c + r_i \geq 0 \end{array}$$

Suppose that constant propagation is applied to the original program, substituting an occurrence of r_1 with b and $b + r_i$ with b, as shown in program (a) in Figure 3. If we do not rewrite assertions, that is we let $\xi_{cp} = \xi$ and $\varphi_{cp} = \varphi$ then the proof obligation that replaces the third proof obligation is $\vdash \varphi \Rightarrow (r_i = n \Rightarrow \phi_t \wedge r_i \neq n \Rightarrow \phi'_f)$, where $\phi'_f \triangleq b * (c + r_i) = r_1 * (c + r_i) \wedge r_1 \geq b \wedge c + r_i \geq 0$ cannot be proved since there is no information about the relation between r_1 and b. A fortiori the certificate of the original program cannot be used to obtain a certificate for the optimized program.

Motivated by the example above, optimized programs are defined augmenting annotations by using the results of the analysis expressed as an assertion, and denoted $\mathrm{RES}_{\mathcal{A}}(L)$ below.

Definition 3.1. *The optimized graph code of a function f is defined as follows:*

$$G_{\bar{f}}(L) = \begin{cases} (\phi \wedge \mathrm{RES}_{\mathcal{A}}(L),\ [\![\mathsf{ins}]\!]) & \text{if } G_f(l) = (\phi,\ \mathsf{ins}) \\ [\![\mathsf{ins}]\!] & \text{if } G_f(l) = \mathsf{ins} \end{cases}$$

where $[\![\mathsf{ins}]\!]$ is the optimized version of instruction ins. In the sequel, we write $\bar{\phi}_L$ for $\phi_L \wedge \mathrm{RES}_{\mathcal{A}}(L)$.

In addition, we define the precondition and postcondition of $\bar{f}$ to be those of f. Then one can encode elementary reasoning with the rules of the proof algebra to obtain a valid certificate for the optimized function $\overline{f}$ from a function:

$$T_L^{\mathsf{ins}}:\ \forall L,\ \mathcal{P}(\vdash \mathsf{vcg}_f^{\mathtt{id}}(L) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{vcg}_{\bar{f}}^{\mathtt{id}}(L))$$

and a certified program:

$$f_{\mathcal{A}} = \{\vec{r}_f;\ \top;\ G_{\mathcal{A}};\ \top;\ \lambda_{\mathcal{A}};\ \vec{\Lambda}_{\mathcal{A}}\}$$

where $G_{\mathcal{A}}$ is a new version of G_f annotated with the results of the analysis, i.e. G_f such that $G_{\mathcal{A}}(L) = (\mathrm{RES}_{\mathcal{A}}(L),\ \mathsf{ins})$ for all label L in f.

Thus, certificate translation is reduced to two tasks: defining the function T_L^{ins}, and producing the certified function $f_{\mathcal{A}}$. The definition of T_L^{ins} depends upon the program optimization. In the next paragraph we show that T_L^{ins} can be built for many common program optimizations, using the induction principle attached to the definition of $\mathsf{reachAnnot}_f$. As to the second task, it is delegated to a procedure, called certifying analyzer, that produces for each function f the certified function $f_{\mathcal{A}}$. There are two approaches for building certifying analyzers: one can either perform the analysis and build the certificate simultaneously, or use a standard analysis and use a decision procedure to

(a) Constant propagation			(b) Loop induction			(c) Dead register		
$L_1:$		$r_i := 0,\ L_2$	$L_1:$		$r_i := 0,\ L_2$	$L_1:$		$r_i := 0,\ L_2$
$L_2:$	$\xi_{cp},$	$r_1 := b,\ L_3$	$L_2:$	$\xi_{li},$	$r_1 := b,\ L_3$	$L_2:$	$\xi_{dr},$	set $\hat{r_1} := b,\ L_3$
$L_3:$		$r_i := c + r_i,\ L_4$	$L_3:$		$r'_j := b * r_i,\ L'_3$	$L_3:$		$r'_j := b * r_i,\ L'_3$
			$L'_3:$		$r_i := c + r_i,\ L''_3$	$L'_3:$		$r_i := c + r_i,\ L''_3$
			$L''_3:$		$r'_j := m + r'_j,\ L_4$	$L''_3:$		$r'_j := m + r'_j,\ L_4$
$L_4:$		$r_j := b * r_i,\ L_5$	$L_4:$		$r_j := r'_j,\ L_5$	$L_4:$		set $\hat{r_j} := r'_j,\ L_5$
$L_5:$	$\varphi_{cp},$	$(r_i = n)\ ?\ L_6 : L_3$	$L_5:$	$\varphi_{li},$	$(r_i = n)\ ?\ L_6 : L'_3$	$L_5:$	$\varphi_{dr},$	$(r_i = n)\ ?\ L_6 : L'_3$
$L_6:$		return r_j	$L_6:$		return r_j	$L_6:$		return r'_j

Figure 5: Example of different optimizations

generate the certificate post-analysis. The merits of both approaches will be reported elsewhere; here we have followed the second approach. Note that the approach implicitly assumes that the decision procedure is sufficiently powerful to verify the results produced by the analysis; otherwise the certifying analyzer could not be built. However, the completeness of the decision procedure w.r.t. the "logic" of the analysis has not been a problem for the analyzes we considered (the analyzes are based on standard data-flow algorithms, and are fully automatic; it would become an issue if the analysis would be performed interactively).

As shown in Figure 1, certifying analyzers do not form part of the Trusted Computing Base. In particular, no security threat is caused by applying an erroneous analyzer, or by verifying a program whose assertions are too weak (e.g. taking $\mathrm{RES}_{\mathcal{A}}(L_5) = \top$ in the above example) or too strong (by adding unprovable assertions), or erroneous. In these cases, it will either be impossible to generate the certificate of the analysis, or of the optimized program.

4 Instances of certificate translation

This section provides instances of certificate translations for a non-optimizing compiler from source code to RTL, and for common RTL optimizations. The order of optimizations is chosen for the clarity of exposition and does not necessarily reflect the order in which the optimizations are performed by a compiler. Due to space constraints, we only describe certificate translators for constant propagation, loop induction, and dead register elimination. Other transformations (common subexpression elimination, inlining, register allocation, loop unrolling, unreachable code elimination) will be described in the full version of the article.

4.1 Certificate translation from an imperative language to RTL

This section is concerned with showing that proof obligations are preserved a non-optimizing compiler from an imperative language with procedures to RTL.

A program p in the source language is defined as a function from function identifiers to function declarations. Declarations are similar to those in RTL, except that program graphs are replaced by commands, whose syntax is given in Figure 6. Note that the while command is annotated with an assertion (from the same annotation language as RTL). We overload vcg to denote as well verification condition generator for the source code, which is given in Figure 6. The definition of certified source program is similar to that of certified RTL program: that is, a source function f with declaration $\{\vec{x};\ \varphi;\ c;\ \psi;\ \lambda;\ \vec{\Lambda}\}$ is certified if the proof obligations corresponding to its precondition

$$\begin{array}{lcl}
\star & ::= & < \mid \leq \mid = \mid \geq \mid > \\
e & ::= & x \mid n \mid -e \mid e\ +\ e \mid e\ -\ e \mid e\ *\ e \\
c & ::= & \mathsf{skip} \mid x := e \mid c;c \mid \mathsf{while}\ \{I\}\ e \star e\ \mathsf{do}\ c \mid \mathsf{if}\ e \star e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid y := \mathsf{call}\ f(\vec{x})
\end{array}$$

$$\begin{array}{rcl}
\mathsf{vcg}(\mathsf{skip}, \psi) & = & \psi \\
\mathsf{vcg}(x := e, \psi) & = & \psi[e/x] \\
\mathsf{vcg}(c_1; c_2, \psi) & = & \mathsf{vcg}(c_1, \mathsf{vcg}(c_2, \psi)) \\
\mathsf{vcg}(\mathsf{while}\ \{\phi\}\ \mathsf{cmp}\ \mathsf{do}\ c_1, \psi) & = & \phi \\
\mathsf{vcg}(\mathsf{if}\ \mathsf{cmp}\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \psi) & = & \langle \mathsf{cmp} \rangle \Rightarrow \mathsf{vcg}(c_1, \psi) \wedge \neg \langle \mathsf{cmp} \rangle \Rightarrow \mathsf{vcg}(c_2, \psi) \\
\mathsf{vcg}(y := \mathsf{call}\ g(\vec{x})), \psi) & = & \mathsf{pre}(g)\{\vec{x}_g \leftarrow \vec{x}\} \wedge (\forall res.\ \mathsf{post}(g)\{\vec{x}_g^* \leftarrow \vec{x}\} \Rightarrow \psi\{y \leftarrow res\})
\end{array}$$

Figure 6: Syntax of commands and VCGen

and its invariants can be discharged by certificates. Note that there is a one-to-one correspondence between proof obligations at source code and at RTL level. Furthermore, verification condition generation commutes with compilation, i.e. the proof obligations are syntactically equal, thus the certificates can be reused.

Lemma 4.1. *Let f with declaration $\{\vec{x};\ \varphi;\ c;\ \psi;\ \lambda;\ \vec{\Lambda}\}$ be a source certified function. Then $\overline{f}$ declared as $\{\vec{x};\ \varphi;\ [\![c]\!];\ \psi;\ \lambda;\ \vec{\Lambda}\}$ is a RTL certified function.*

By composing adequately the certificate translators obtained in the previous section with the above result, one can therefore conclude the existence of certificate translators for optimizing compilers from the source language to RTL.

4.2 Constant propagation

Goal Constant propagation aims at minimizing at run-time evaluation of expressions and access to registers with constant values.

Description Constant propagation relies on a data flow analysis that returns a function $\mathcal{A} : \mathcal{PP} \times \mathcal{R} \to \mathbb{Z}_\perp$ such that $\mathcal{A}(L, r) = n$ if r holds value n every time execution reaches label L. The optimization consists in replacing instructions by an equivalent one that exploits the information provided by $\mathcal{A}$. For example, if r_1 is known to hold n_1 at label L, and the instruction is $r := r_1 + r_2$, then the instruction is rewritten into $r := n_1 + r_2$. Likewise, conditionals which can be evaluated are replaced with nop instructions.

Certifying analyzer We have implemented a certifying analyzer for constant propagation as an extension of the standard data flow algorithm. First, we attach to each reachable label L the assertion $\mathrm{EQ}_\mathcal{A}(L)$ (since the result of the analysis is a conjunction of equations, we now write $\mathrm{EQ}_\mathcal{A}(L)$ instead of $\mathrm{RES}_\mathcal{A}(L)$):

$$\mathrm{EQ}_\mathcal{A}(L) \equiv \bigwedge_{r \in \{r \mid \mathcal{A}(L,r) \neq \perp\}} r = \mathcal{A}(L, r)$$

To derive a certificate for the analysis, we must prove for each reachable label L:

$$\vdash \mathrm{EQ}_\mathcal{A}(L) \Rightarrow \mathsf{vcg}^{\mathtt{id}}_{f_\mathcal{A}}(L)$$

After performing all $\Rightarrow$-eliminations (i.e. moving hypotheses to the context), and rewriting all equalities from the context in the goal, one is left to prove closed equalities of the form $n = n'$ (i.e. n, n' are numbers and not arithmetic expressions with variables). If the assertions are correct, then the certificate is obtained by applying reflexivity of equality (an instance of the ring rule). If the assertions are not correct, the program cannot be certified.

Certificate translation The function T_L^{ins} is defined by case analysis, using that the transformation of operations is correct relative to the results of the analysis:

$$T_{\mathsf{op}} \; : \; \forall L,\; \forall \mathsf{op}, \mathcal{P}(\vdash \mathrm{EQ}_{\mathcal{A}}(L) \Rightarrow \langle \mathsf{op} \rangle = \langle [\![\mathsf{op}]\!]_L^{\mathsf{op}} \rangle)$$

The function T_{op} is built using the ring axiom of the proof algebra; a similar result is required for comparisons and branching instructions.

Example 4.1. *Recall function f, defined in Example 3.1. Using the compiler and the transformation of assertions given in Figure 12, we obtain the optimized program shown in Figure 3 (a), where assertions at L_1 and L_3 have been transformed into $\xi_{cp} \triangleq \xi \wedge r_i = 0$ and $\varphi_{cp} \triangleq \varphi \wedge r_1 = b$. It is left to the reader to check that all proof obligations become provable with the new annotations.*

4.3 Loop induction register strength reduction

Goal Loop induction aims at reducing the number of multiplication operations inside a loop, which in many processors are more costly than addition operations.

Description Loop induction depends on two analyzes. The first one is a loop analysis that detects loops and returns for each loop its set of labels $\{L_1, \ldots, L_n\}$, and its header L_H, a distinguished label in the above set such that any jump that goes inside the loop from an instruction outside the loop, is a jump to L_H.

The second analysis detects inside a loop an induction register r_i (defined in the loop by an instruction of the form $r_i := r_i + c$) and its derived induction register r_d (defined in the loop by an instruction of the form $r_d := r_i * b$). More precisely, the analysis returns : an induction register r_i and the label L_i in which its definition appears, a derived induction register r_d and the label L_d in which its definition appears, a new register name r'_d not used in the original program, two new labels L''_i and L''_H not in the domain of G_f and two constant values b, c that correspond to the coefficient of r_d and increment of r_i.

The transformation replaces assignments to the derived induction register r_d with less costly assignments to an equivalent induction register r'_d. Then r_d is defined as a copy of r'_d.

Certifying analyzer Only the second analysis needs to be certified. First, we define $EQ_{\mathcal{A}}(L) \equiv r'_d = b * r_i$ if $L \in \{L''_H, L_1, \ldots, L_n\} \setminus \{L_H\}$ and $EQ_{\mathcal{A}}(L) \equiv \top$ if L is a label outside the loop or equal to L_H. Then, we need to create a certificate that the analysis is correct. One (minor) novelty w.r.t. constant propagation is that the definition of $f_{\mathcal{A}}$ includes two extra labels L''_H and L''_i, not present in the original function f. The definition of $f_{\mathcal{A}}$ is given by the clauses:

$$\begin{array}{lcl}
\overline{f}[L_H] & = & r'_d := b * r_i,\ L''_H \\
\overline{f}[L''_H] & = & f[L_H] \\
\overline{f}[L_i] & = & r_i := r_i + c,\ L''_i \\
\overline{f}[L''_i] & = & r'_d := r'_d + b * c,\ L'_i\{L_H \leftarrow L''_H\} \\
\overline{f}[L_d] & = & r_d := r'_d,\ L'_d\{L_H \leftarrow L''_H\} \\
\overline{f}[L] & = & (\phi \wedge r'_d = b * r_i,\ \mathsf{ins}\{L_H \leftarrow L''_H\}) \text{ if } f[L] = (\phi,\ \mathsf{ins}) \\
\overline{f}[L] & = & f[L]\{L_H \leftarrow L''_H\} \text{ in any other case inside the loop}
\end{array}$$

Figure 7: Loop Induction Register Strength Reduction

$$\begin{array}{lcl}
f_{\mathcal{A}}[L_H] & = & (EQ_{\mathcal{A}}(L_H),\ r'_d := b * r_i,\ L''_H) \\
f_{\mathcal{A}}[L''_H] & = & (EQ_{\mathcal{A}}(L''_H),\ \mathsf{ins}_{L_H}) \\
f_{\mathcal{A}}[L] & = & (EQ_{\mathcal{A}}(L),\ \mathsf{ins}_L) \text{if } L \in dom(G_f),\ L \notin \{L_H, L_i\} \\
f_{\mathcal{A}}[L_i] & = & (EQ_{\mathcal{A}}(L_i),\ \mathsf{ins}_{L_i}\{L'_i \leftarrow L''_i\}) \\
f_{\mathcal{A}}[L''_i] & = & (\top,\ r'_d := r'_d + b * c,\ L'_i)
\end{array}$$

where ins_L is the instruction descriptor of $f[L]$, and L'_i is the successor label of L_i in f. Interestingly, the certified analyzer must use the fact that the loop analysis is correct in the sense that one can only enter a loop through its header. If the loop analysis is not correct, then the certificate cannot be constructed.

Certificate translation Figure 7 shows how instructions for labels $L_1 \ldots L_n$ of a function f are transformed into instructions for the optimized function $\overline{f}$. As expected, the transformation for instructions outside the loop is the identity, i.e. $\overline{f}[L] = f[L]$ for $L \notin \{L_1, \ldots, L_n\}$.

Certificate translation proceeds as with constant propagation, using the induction principle attached to the definition of $\mathsf{reachAnnot}_f$, and the certificate of the analysis, to produce a certificate for $\bar{f}$.

Example 4.2. *Applying loop induction to program (a) in Figure 3, we obtain program (b) where m denotes the result of the product $b * c$ and $\xi_{li} \triangleq \xi_{cp}$ and $\varphi_{li} \triangleq \varphi_{cp} \wedge r'_j = b * r_i$.*

4.4 Dead register elimination by ghost variable introduction

Goal Dead register elimination aims at deleting assignments to registers that are not live at the label where the assignment is performed. As mentioned in the introduction, we propose a transformation that performs simultaneously dead variable elimination in instructions and in assertions.

Description A register r is live at label L if r is read at label L or there is a path from L that reaches a label L' where r is read and does not go through an instruction that defines r (including L, but not L'). A register r is read at label L if it appears in an assignment with a function call, or it appears in a conditional jump, or in a return instruction, or on the right side of an assignment of an assignment operation to a register r' that is live. In the following, we denote $\mathcal{L}(L, r) = \top$ when a register is live at L.

In order to deal with assertions, we extend the definition of liveness to assertions. A register r is live in an assertion at label L, denoted by $\mathcal{L}(L, r) = \top_\phi$, if it is not live at label L and there is a path from L that reaches a label L' such that r appears in assertion at L' or where r is used to define a register which is live in an assertion.

By abuse of notation, we use $\mathcal{L}(L,r) = \bot$ if r is dead in the code and in assertions.

The transformation deletes assignments to registers that are not live. In order to deal with dead registers in assertions, we rely on the introduction of ghost variables. Ghost variables are expressions in our language of assertions (we assume that sets of ghost variables names and $\mathcal{R}$ are non overlapping). We introduce as part of RTL, "ghost assignments"of the form $\mathsf{set}\ \hat{v} := \mathsf{op},\ L$, where $\hat{v}$ is a ghost variable. Ghost assignments do not affect the semantics of RTL, but they affect the calculus of vcg in the same way as normal assignments.

The transformation is shown below where $\sigma_L = \{r \leftarrow \hat{r} \mid \mathcal{L}(L,r) = \top_\phi\}$ and $\mathsf{dead}_c(L,L') = \{r | \mathcal{L}(L,r) = \top \wedge \mathcal{L}(L',r) = \top_\phi\}$.

$$\begin{array}{lcl} \mathsf{ghost}_L((\phi,\ \mathsf{ins})) & = & (\phi\sigma_L,\ \mathsf{ghost}_L^{\mathtt{id}}(\mathsf{ins})) \\ \mathsf{ghost}_L(\mathsf{ins}) & = & \mathsf{ghost}_L^{\mathtt{id}}(\mathsf{ins}) \end{array}$$

The analysis $\mathsf{ghost}_L^{\mathtt{id}}(\mathsf{ins})$ is defined in Figure 8. We use $\mathsf{set}\ \hat{\vec{r}} := \vec{r},$ as syntactic sugar for a sequence of assignments $\mathsf{set}\ \hat{r_i} := r_i,$ where for each register r_i in $\vec{r}$, $\hat{r_i}$ in $\hat{\vec{r}}$ is its corresponding ghost variable. The function ghost transforms each instruction of f into a the set of instructions of $\overline{f}$. Intuitively, it introduces for any instruction ins (with successor L') at label L, a ghost assignment $\mathsf{set}\ \hat{r} := r,\ L'$ immediately after L (at a new label L'') if the register r is live at L but not live at the immediate successor L' of L. In addition, the function ghost_L performs dead register elimination if ins is of the form $r_d := \mathsf{op},$ and the register r_d is not live at L.

Instantiation of proof obligations Certificate translation for dead register elimination falls in the IPO category, i.e. the certificate of the optimized program is an instance of the certificate of the source program. This is shown by proving that ghost variable introduction preserves vcg up to substitution.

Lemma 4.2. $\forall L, \mathsf{vcg}_{\bar{f}}(L) = \mathsf{vcg}_f(L)\sigma_L$

A consequence of this lemma is that if the function f is certified, then it is possible to reuse the certificate of f to certify $\overline{f}$, as from each proof $p :\vdash \phi_L \Rightarrow \mathsf{vcg}_f(L)$ we can obtain a proof $\overline{p} :\vdash \bar{\phi}_L \Rightarrow \mathsf{vcg}_{\bar{f}}(L)$ by applying subst rule of Figure 4 to p with substitution σ_L.

After ghost variable introduction has been applied, registers that occur free in $\mathsf{vcg}_{\overline{f}}(L)$, are live at L, i.e. $\mathcal{L}(L,r) = \top$.

Example 4.3. *In Figure 3, applying first copy propagation to program (b) (replacing r_j with r'_j at label L_6), we can then apply ghost variable introduction to obtain program (c), where $\xi_{dr} \triangleq \xi_{li}$ and $\varphi_{dr} \triangleq \hat{r_j} = \hat{r_1} * r_i \wedge \hat{r_1} \geq b \wedge r_i \geq 0 \wedge\ \hat{r_1} = b \wedge r'_j = b * r_i \wedge r'_j = \hat{r_j}$.*

5 Application scenarios

Certificate translation is potentially useful in a wide range of scenarios, including mobile code, and any other scenario where the reliability and security of software is so crucial to justify its (potentially interactive) formal verification. The purpose of this section is to briefly discuss some potential application scenarios.

Mobile telephony provides a very promising application domain for certificate translation. Currently, mobile phone operators have access to hundreds of midlets, originating from possibly untrusted parties, and awaiting to be made available to customers. Despite the market opportunities

$$
\begin{array}{ll}
\mathsf{ghost}_L^{\mathtt{id}}(\mathsf{nop},\ L') = \mathsf{nop},\ L' & \mathsf{ghost}_L^{\mathtt{id}}(\mathsf{cmp}\ ?\ L_1 : L_2) = \mathsf{cmp}\ ?\ L'_1 : L'_2 \\
\mathsf{ghost}_L^{\mathtt{id}}(\mathsf{return}\ r) = \mathsf{return}\ r & L'_1 \mapsto \mathsf{set}\ \hat{\vec{t_1}} := \vec{t_1},\ L_1 \\
\mathsf{ghost}_L^{\mathtt{id}}(r_d := f(\vec{r}),\ L') = r_d := f(\vec{r}),\ L'' & L'_2 \mapsto \mathsf{set}\ \hat{\vec{t_2}} := \vec{t_2},\ L_2 \\
L'' \mapsto \mathsf{set}\ \hat{\vec{t}} := \vec{t},\ L' & \vec{t_1} = \mathsf{dead}_c(L, L_1) \\
\vec{t} = \mathsf{dead}_c(L, L') & \vec{t_2} = \mathsf{dead}_c(L, L_2)
\end{array}
$$

$$
\begin{array}{rcll}
\mathsf{ghost}_L^{\mathtt{id}}(r_d := \mathsf{op},\ L') & = & \mathsf{nop},\ L' & \text{if } \mathcal{L}(L', r_d) = \bot \\
 & = & \mathsf{set}\ \hat{r}_d := \mathsf{op}\sigma_L,\ L' & \text{if } \mathcal{L}(L', r_d) = \top_\phi \\
 & = & r_d := \mathsf{op},\ L'' & \text{if } \mathcal{L}(L', r_d) = \top \\
 & & L'' \mapsto \mathsf{set}\ \hat{\vec{t}} := \vec{t},\ L' & \\
 & & \vec{t} = \mathsf{dead}_c(L, L') &
\end{array}
$$

Figure 8: Ghost Variable Introduction-Dead Register Elimination

that would result from offering a vast panel of new services to customers, telecom operators are reluctant to release these midlets because of the lack of tools to analyze them, the danger of releasing unverified midlets, and the disastrous consequences of being liable for distributing hostile midlets that would cause havoc to the network. Hence, some major phone operators are considering to use PCC as a solution to this dilemma, by making them available on their portal cryptographically signed midlets (thus engaging their liability) provided they come with a PCC certificate establishing their adherence to the operator policy. In such a scenario, it is likely that the midlet producer will verify the source code, and a certificate translator to produce a certificate for the midlet itself (especially because source code will be compiled in several dozen versions to accommodate the different platforms). Further work is required to assess the benefits of certificate translation in the context of mobile telephony, but there has been previous work using verification techniques based on VCGens to enforce safety and security policies of applications, including exception safety, non-interference, and resource consumption.

Certificate translation could also be useful in the component-based development of security-sensitive software, as the software integrator, who will be liable for the resulting product, could reasonably require that components originating from untrusted third parties are certified against their requirements, and use certificate translation to derive a certificate for the overall software from certificates of each component. The benefits of certificate translation seem highest in situations where integration of components involves advanced compilation techniques, e.g. compilation from Domain-Specific Languages or Aspect-Oriented Languages to conventional languages.

6 Concluding remarks

Certificate translation provides a means to bring the benefits of source code verification to code consumers using Proof Carrying Code architectures based on VCGens, and extends the scope of PCC to complex security policies.

6.1 Relation with certifying and certified compilation

Compiler correctness [5] aims at showing that a compiler preserves the semantics of programs. Because compilers are complex programs, the task of compiler verification can be daunting; in order to tame the complexity of verification and bring stronger guarantees on the validity of compiler correctness proofs, *certified compilation*, see e.g. [7], advocates the use of a proof assistant for machine-checking compiler correctness results. Certified compilation is relevant for embedded software, but cannot be used realistically for certificate translation (because of the size of certificates, of the restrictions on the properties that can be handled, and of the effort of certifying a compiler).

Certifying compilation [9] provides an automatic means to generate certificates for safety policies such as memory safety or type safety. Certifying compilers contribute greatly to the scalability of PCC, because they eliminate the need for verification on the code producer side; however, certifying compilation is by design restricted in terms of expressiveness and precision. Thus, certifying compilation is complementary to certificate translation.

6.2 Other related work

Source code vs. compiled code Our work can be seen as a way to validate a more practical approach, as the Spec# project [2]. The Spec# project defines an extension of C# with annotations, and a compiler from annotated programs to annotated .NET files, which can be run using the .NET platform, and checked against their specifications at run-time or verified statically with an automatic prover. The Spec# project implicitly assumes some relation between source and bytecode levels (otherwise the approach would be meaningless), but we are not aware of any attempt to formalize this relation. Similar work is taking place in the context of Java around ESC/Java2, again without formalizing the relation between verification at source and bytecode levels.

This relation has been studied independently of our work by Bannwart and Müller [1], who provide Hoare-like logics for significant sequential fragments of Java source code and bytecode, and illustrate how derivations of correctness can be mapped from source programs to their corresponding bytecode programs. They do not consider program optimizations.

Furthermore, the issue of transferring evidence from source programs to compiled programs in scenarios that use alternative technologies for establishing program correctness. Relevant results include preservation by compilation of program invariants inferred using abstract interpretation techniques [10] and type-preserving compilation [3, 12].

Certifying analyzers The possibility of defining proof-producing program analyzes has been studied independently by Wildmoser, Chaieb and Nipkow [13] in the context of a bytecode language and by Seo, Yang and Yi [11] in the context of a simple imperative language. However, these works do not propose a general definition to study certifying analyzers, as we do.

Future work The present work establishes the principles of certificate translation, and shows its viability. Further work is needed to make certificate translation a practical tool for mobile code, including:

- extending our results to: i) a mainstream programming language such as C or Java; ii) a complete compilation scheme from source to assembly (which amounts to extend the results of this paper with a translation from RTL to assembly); iii) other common optimizations, including loop unrolling, stack allocation (for object-oriented source languages), enforcing calling conventions, linearizing code, and spilling (for RTL to assembly).
- systematizing our approach to families of program optimizations. Millstein *et al.* [6] provide a domain-specific language Rhodium for declaring a variety of standard optimizations, including many of those studied in this paper, and proving them correct. More precisely, Rhodium generates for each optimization a set of proof obligations from which its correctness follows. It would be very interesting to adapt certificate translation to this setting, by providing a

means to construct interactively a certificate translator from the definition of a Rhodium transformation.

- experimenting with certificate translation in the context of Trusted Personal Devices (smartcards, cell phones, etc) and global computing. The experimentations will also indicate whether certificate translation significantly increases the size of certificates in practice. We conjecture that the increase is moderate, and can be minimized for certificates based on λ-terms, using partial normalization algorithms.

We conclude by pointing out that the relevance of certifying analyzers escapes the scope of certificate translation, and their study deserves further attention. In a forthcoming article, we shall report on the principles and implementations of certifying analyzers for many optimizations, and discuss possible extensions to smarter analyzers that take into account the annotations provided in the program.

References

[1] F. Bannwart and P. Müller. A program logic for bytecode. In F. Spoto, editor, *Proceedings of Bytecode'05*, Electronic Notes in Theoretical Computer Science. Elsevier Publishing, 2005.

[2] M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 50–71. Springer-Verlag, 2005.

[3] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In B. Steffen and G. Levi, editors, *Proceedings of VMCAI'04*, volume 2934 of *Lecture Notes in Computer Science*, pages 2–15. Springer-Verlag, 2004.

[4] C. Colby, P. Lee, G.C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proceedings of PLDI'00*, pages 95–107. ACM Press, 2000.

[5] J. D. Guttman and M. Wand. Special issue on VLISP. *Lisp and Symbolic Computation*, 8(1/2), March 1995.

[6] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of POPL'05*, pages 364–377. ACM Press, 2005.

[7] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of POPL'06*. ACM Press, 2006.

[8] G.C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.

[9] G.C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of PLDI'98*, pages 333–344. ACM Press, 1998.

[10] X. Rival. Symbolic Transfer Functions-based Approaches to Certified Compilation. In *Proceedings of POPL'04*, pages 1–13. ACM Press, 2004.

[11] S. Seo, H. Yang, and K. Yi. Automatic construction of hoare proofs from abstract interpretation results. In A. Ohori, editor, *Proceedings of APLAS'03*, volume 2895 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2003.

[12] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of PLDI'96*, pages 181–192. ACM Press, 1996.

[13] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In F. Spoto, editor, *Proceedings of BYTECODE'05*, Electronic Notes in Theoretical Computer Science. Elsevier Publishing, 2005.

A Soundness of VCGen

In order to prove the soundness of our approach, it is necessary to define formally an execution state and the validity of an assertion over a state. A frame for an implicit function f, is defined as a tuple with four elements: the current instruction, a map ρ from local register to values, an auxiliary map ρ^*, and a return register. A state is defined as a non empty stack of frames.

We say that an assertion is valid in a program state, denoted $\langle \mathsf{ins}, \rho, \rho^*, r\rangle :: fs \models \phi$ if $[\![\phi]\!]\,\rho\,\rho^*$ evaluates to true. The definition of $[\![]\!]$ over assertions is standard, with the difference of an extra argument, motivated by the need of evaluating expressions that contains auxiliary registers that represent the initial value of parameters in a function call. The definition of $[\![]\!]$ over expressions is slightly modified from its standard version to deal with that mentioned set of extra registers, and is formalized in Figure 9.

$$\begin{array}{lcl}
[\![n]\!]\,\rho\,\rho* & = & n \\
[\![r]\!]\,\rho\,\rho* & = & \rho r \\
[\![r^*]\!]\,\rho\,\rho* & = & \rho^* r^* \\
[\![-e]\!]\,\rho\,\rho* & = & -[\![e]\!]\,\rho\,\rho* \\
[\![e_1 + e_2]\!]\,\rho\,\rho* & = & [\![e_1]\!]\,\rho\,\rho* + [\![e_2]\!]\,\rho\,\rho* \\
[\![e_1 - e_2]\!]\,\rho\,\rho* & = & [\![e_1]\!]\,\rho\,\rho* - [\![e_2]\!]\,\rho\,\rho* \\
 & \vdots &
\end{array}$$

Figure 9: Semantics of Expressions

Stack frames are related by $\rightsquigarrow$, which represents an execution step and is defined in figure 10.

The first lemma proves invariance of assertions in a one step basis, while the next theorem using this lemma proves the validity of VCGen over a finite number of steps. Soundness of VCGen for a complete execution follows easily as a corollary of the theorem.

Lemma A.1. *Let p be a certified program. Let S_L and $S_{L'}$ be execution states, if $f[L]$ does not contain a* `return` *instruction, $S_L \models \mathsf{vcg}^{\mathtt{id}}_f(L)$ and $S_L \rightsquigarrow S_{L'}$ then $S_{L'} \models \mathsf{vcg}_g(L')$, where g and L' are the corresponding function and label for $S_{L'}$.*

Proof.

- interesting case $f[L] = ret_g := g(\vec{r}),\ L'$.

$$\begin{array}{lcl}
S_L & = & \langle ret_g := g(\vec{r}),\ L', \rho_f, \rho^*_f, ret_f\rangle :: fs \\
S_{L'} & = & \langle g[L_{\mathsf{sp}}], [\vec{r}_g \mapsto \rho_f\vec{r}], [\vec{r}^*_g \mapsto \rho_f\vec{r}], ret_g\rangle \\
 & & :: \langle f[L'], \rho_f, \rho^*_f, ret_f\rangle :: fs
\end{array}$$

$$\begin{array}{ll}
 & S_L \models \mathsf{vcg}_f(L) \\
\equiv & [\![\mathsf{vcg}_f(L)]\!]\,\rho_f\,\rho^*_f \\
\equiv & [\![\mathsf{pre}(g)\{\vec{r}_g \leftarrow \vec{r}\} \wedge (\forall res \ldots)]\!]\,\rho_f\,\rho^*_f \\
\Rightarrow & [\![\mathsf{pre}(g)\{\vec{r}_g \leftarrow \vec{r}\}]\!]\,\rho_f\,\rho^*_f \\
\equiv & [\![\mathsf{pre}(g)]\!]\,[\vec{r}_g \mapsto \rho_f\vec{r}]\,[\vec{r}^*_g \mapsto \rho_f\vec{r}] \\
\Rightarrow & [\![\mathsf{vcg}_g\{\vec{r}^*_g \leftarrow \vec{r}_g\}]\!]\,[\vec{r}_g \mapsto \rho_f\vec{r}]\,[\vec{r}^*_g \mapsto \rho_f\vec{r}] \\
\equiv & [\![\mathsf{vcg}_g]\!]\,[\vec{r}_g \mapsto \rho_f\vec{r}]\,[\vec{r}^*_g \mapsto \rho_f\vec{r}] \\
\equiv & S_{L'} \models \mathsf{vcg}_g(L_{\mathsf{sp}})
\end{array}$$

$$\frac{\langle \mathsf{ins}, \rho, \rho^*, r' \rangle :: fs \rightsquigarrow fs'}{\langle (\phi,\ \mathsf{ins}), \rho, \rho^*, r' \rangle :: fs \rightsquigarrow fs'}$$

$$\frac{}{\langle r_d := \mathsf{op},\ L, \rho, \rho^*, r' \rangle :: fs \rightsquigarrow \langle f[L], \rho \oplus \{r_d \mapsto [\![\mathsf{op}]\!]\, \rho\, \rho^*\}, \rho^*, r' \rangle :: fs}$$

$$\frac{}{\langle \mathsf{cmp}\ ?\ L_t : L_f, \rho, \rho^*, r' \rangle :: fs \rightsquigarrow \langle f[L_t], \rho, \rho^*, r' \rangle :: fs}\ \text{if } [\![\langle \mathsf{cmp} \rangle]\!]\, \rho\, \rho*$$

$$\frac{}{\langle \mathsf{cmp}\ ?\ L_t : L_f, \rho, \rho^*, r' \rangle :: fs \rightsquigarrow \langle f[L_f], \rho, \rho^*, r' \rangle :: fs}\ \text{if } \neg[\![\langle \mathsf{cmp} \rangle]\!]\, \rho\, \rho$$

$$\frac{}{\langle ret_g := g(\vec{r}),\ L', \rho_f, \rho_f^*, r_f \rangle :: fs \rightsquigarrow \langle g[L_{\mathsf{sp}}], [\vec{r}_g \mapsto \rho_f \vec{r}], [\vec{r}_g^* \mapsto \rho_f \vec{r}], ret_g \rangle :: \langle f[L'], \rho_f, \rho_f^*, r_f \rangle :: fs}$$

$$\frac{}{\langle \mathsf{return}\ r, \rho_g, \rho_g^*, ret_g \rangle :: \langle f[L'], \rho_f, \rho_f^*, ret_f \rangle :: fs \rightsquigarrow \langle f[L'], \rho_f \oplus \{ret_g \mapsto \rho_g r\}, \rho_f^*, ret_f \rangle :: fs}$$

Figure 10: Operational Semantics

- case $f[L] = \mathsf{cmp}\ ?\ L_t : L_e$ and $[\![\mathsf{cmp}]\!]\, \rho\, \rho^*$.

$$\begin{aligned} S_L &= \langle \mathsf{cmp}\ ?\ L_t : L_e, \rho, \rho^*, ret_f \rangle :: fs \\ S_{L'} &= \langle f[L_t], \rho, \rho^*, ret_f \rangle :: fs \end{aligned}$$

$$\begin{aligned} & S_L \models \mathsf{vcg}_f(L) \\ \equiv\ & [\![\mathsf{vcg}_f(L)]\!]\, \rho\, \rho^* \\ \equiv\ & [\![\mathsf{cmp}]\!]\, \rho\, \rho^* \Rightarrow [\![\mathsf{vcg}_f(L_t)]\!]\, \rho\, \rho^* \\ & \wedge \neg[\![\mathsf{cmp}]\!]\, \rho\, \rho^* \Rightarrow [\![\mathsf{vcg}_f(L_e)]\!]\, \rho\, \rho^* \\ \equiv\ & [\![\mathsf{vcg}_f(L_t)]\!]\, \rho\, \rho^* \\ \equiv\ & S_{L'} \models \mathsf{vcg}_f(L_t) \end{aligned}$$

- case $f[L] = r := \mathsf{op},\ L'$.

$$\begin{aligned} S_L &= \langle \mathsf{cmp}\ ?\ L_t : L_e, \rho, \rho^*, ret_f \rangle :: fs \\ S_{L'} &= \langle f[L'], \rho \oplus \{r \mapsto [\![\mathsf{op}]\!]\, \rho\, \rho^*\}, \rho^*, ret_f \rangle :: fs \end{aligned}$$

$$\begin{aligned} & S_L \models \mathsf{vcg}_f(L) \\ \equiv\ & [\![\mathsf{vcg}_f(L)]\!]\, \rho\, \rho^* \\ \equiv\ & [\![\mathsf{vcg}_f(L')\{r \leftarrow \mathsf{op}\}]\!]\, \rho\, \rho^* \\ \equiv\ & [\![\mathsf{vcg}_f(L')]\!]\, \rho \oplus \{r \mapsto [\![\mathsf{op}]\!]\, \rho\, \rho^*\}\, \rho^* \\ \equiv\ & S_{L'} \models \mathsf{vcg}_f(L') \end{aligned}$$

- case $f[L] = \mathsf{nop},\ L'$.

$$\begin{aligned} & S_L \models \mathsf{vcg}_f L \\ \equiv\ & [\![\mathsf{vcg}_f L]\!]\, \rho\, \rho^* \\ \equiv\ & [\![\mathsf{vcg}_f L']\!]\, \rho\, \rho^* \\ \equiv\ & S_{L'} \models \mathsf{vcg}_f L' \end{aligned}$$

$\square$

Theorem 1. *Suppose p a certified program. Let $\{S_1, .., S_n\}$ be a set of execution states such that $S_1 = \langle \mathsf{main}[L_{\mathsf{sp}}], \rho, \rho^*, ret\rangle :: []$, $S_1 \rightsquigarrow S_2 \rightsquigarrow \ldots \rightsquigarrow S_n$ and $S_1 \models \mathsf{vcg}_{\mathsf{main}}(L_{\mathsf{sp}})$, then $S_i \models \mathsf{vcg}_{g_i}(L_{S_i})$, where g_i and L_{S_i} are the corresponding function and label for any $S_i \in \{S_1, \ldots, S_n\}$*

Proof. The proof is by induction on the length of the execution n, being the function call the only interesting inductive step (because otherwise we can apply previous lemma).

Suppose
$$S_n = \langle \mathsf{return}\ r, \rho_g, \rho_g^*, ret_g\rangle :: \langle f[L'], \rho_f, \rho_f^*, ret_f\rangle :: fs$$
and call S_j the last label in f such that
$$S_j = \langle ret_g := g(\vec{r}),\ L', \rho_f, \rho_f^*, ret_f\rangle :: fs$$
and
$$\begin{array}{rcl} S_{j+1} & = & \langle g[L_{\mathsf{sp}}], [\vec{r}_g \mapsto \rho_f\vec{r}], [\vec{r}_g* \mapsto \rho_f\vec{r}], ret_g\rangle \\ & & :: \langle f[L'], \rho_f, \rho_f^*, ret_f\rangle :: fs \end{array}$$
We know by inductive hypothesis that
$$S_n \models \mathsf{vcg}_g(L_n) \tag{1}$$
and
$$S_j \models \mathsf{vcg}_f(L_j) \tag{2}$$
From (1) and the fact that $\rho_g^*\vec{r_g}^* = \rho_f\vec{r}$ it can be proved that
$$[\![\mathsf{post}(g)\{\vec{r}_g^* \leftarrow \vec{r}\}]\!]\ \ \rho_f \oplus \{res \mapsto \rho_g r\}\ \rho_f^*$$
and from (2), for any value v
$$\begin{array}{rl} & [\![\mathsf{post}(g)\{\vec{r}_g^* \leftarrow \vec{r}\}]\!]\ \ \rho_f \oplus \{res \mapsto v\}\ \rho_f^* \\ \Rightarrow & [\![\mathsf{vcg}_f(L')]\!]\ \ \rho_f \oplus \{ret_g \mapsto v\}\ \rho_f^* \end{array}$$
So we can conclude
$$S_{n+1} = \langle f[L'], \rho_f \oplus \{ret_g \mapsto \rho_g r\}, \rho_f^*, ret_f\rangle \models \mathsf{vcg}_f(L')$$

It remains to say what happens if $f[L]$ contains an assertion. Suppose $f[L] = (\phi,\ \mathsf{ins})$ and let
$$\begin{array}{rcl} S_n & = & \langle (\phi,\ \mathsf{ins}), \rho, \rho^*, ret_f\rangle :: fs \\ S_n^{\mathsf{ins}} & = & \langle \mathsf{ins}, \rho, \rho^*, ret_f\rangle :: fs \\ S_{n+1} & = & fs' \end{array}$$
The derivation of $S_n \rightsquigarrow S_{n+1}$ must follow from $S_n^{\mathsf{ins}} \rightsquigarrow S_{n+1}$. As $\mathsf{vcg}_f(L_n) = \phi$ and f is certified, from $\vec{\Lambda}(L_n)$ and $S_n \models \mathsf{vcg}_f(L_n)$ we can prove that $S_n \models \mathsf{vcg}_f^{\mathsf{id}}(L_n)$ or, what is the same, $S_n^{\mathsf{ins}} \models \mathsf{vcg}_f^{\mathsf{id}}(L_n)$ Therefore, if ins is not a return instruction, we can apply Lemma A.1 to conclude $S_{n+1} \models \mathsf{vcg}_f(L_{n+1})$. Otherwise, we can repeat the analysis concerning the return instruction stated above.

Any other inductive step follows easily by application of Lemma A.1.

$\square$

$$
\begin{array}{lll}
\mathsf{intro}_\top & : & \mathcal{P}(\Gamma \vdash \top) \\
\mathsf{axiom} & : & \mathcal{P}(\Gamma; A; \Delta \vdash A) \\
\mathsf{ring} & : & \mathcal{P}(\Gamma \vdash n_1 = n_2) \qquad \text{if } n_1 = n_2 \text{ is a ring equality} \\[1em]
\mathsf{intro}_\wedge & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma \vdash B) \to \mathcal{P}(\Gamma \vdash A \wedge B) \\
\mathsf{elim}^{\mathsf{l}}_\wedge & : & \mathcal{P}(\Gamma \vdash A \wedge B) \to \mathcal{P}(\Gamma \vdash A) \\
\mathsf{elim}^{\mathsf{r}}_\wedge & : & \mathcal{P}(\Gamma \vdash A \wedge B) \to \mathcal{P}(\Gamma \vdash B) \\[1em]
\mathsf{intro}_\Rightarrow & : & \mathcal{P}(\Gamma; A \vdash B) \to \mathcal{P}(\Gamma \vdash A \Rightarrow B) \\
\mathsf{elim}_\Rightarrow & : & \mathcal{P}(\Gamma \vdash A \Rightarrow B) \to \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma \vdash B) \\[1em]
\mathsf{elim}_= & : & \mathcal{P}(\Gamma \vdash e_1 = e_2) \to \\
 & : & \quad \mathcal{P}(\Gamma \vdash A\{r \leftarrow e_1\}) \to \mathcal{P}(\Gamma \vdash A\{r \leftarrow e_2\}) \\[1em]
\mathsf{subst} & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma\{r \leftarrow e\} \vdash A\{r \leftarrow e\}) \\[1em]
\mathsf{weak} & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma; \Delta \vdash A) \\[1em]
\mathsf{intro}_\forall & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma \vdash \forall r.A) \qquad \text{if } r \text{ is not in } \Gamma \\[1em]
\mathsf{elim}_\forall & : & \mathcal{P}(\Gamma \vdash \forall r.A) \to \mathcal{P}(\Gamma \vdash A)
\end{array}
$$

Figure 11: Proof Algebra

B Certificate Translation

Certificates provide a formal representation of proofs, and are used to verify that the proof obligations generated by the VCGen hold. For the purpose of certificate translation, we do not need to commit to a specific format for certificates. Instead, we assume that certificates are closed under specific operations on certificates, which are captured by an abstract notion of proof algebra.

Recall that a judgment is a pair consisting of a list of assertions, called context, and of an assertion, called goal. Then a *proof algebra* is given by a set-valued function $\mathcal{P}$ over judgments, and by a set of operations, all implicitly quantified in the obvious way, and given in Figure 11. The operations are standard, to the exception perhaps of the substitution operator that allows to substitute selected instances of equals by equals, and of the operator ring, which establishes all ring equalities that will be used to justify the optimizations.

B.1 Certificate Translation for *constant propagation*

B.1.1 Description of the transformation

The definition of the optimization of instructions over a function f can be found in Figure 12. The optimized function $\overline{f}$ will be such that $\overline{f}[L] = [\![f[L]]\!]_L$ for every label L in the domain of G_f. The transformation of an annotated instruction is, as previously defined, the transformation of the annotation and the transformation of the descriptor. The main transformations are for branching instructions which are replaced (whenever the result of the test is known at compile time) by nop instructions with successors pointing to the corresponding branch. Then operations are also modified w.r.t. the result of the analysis. For example, if both arguments of an addition operation are known to be equal to n_1 and n_2, the operation is directly replaced by an immediate load of the

$$
\begin{array}{rcl}
[\![(\phi,\ \mathsf{ins})]\!]_L & = & (\phi \wedge \mathrm{EQ}_{\mathcal{A}}(L),\ [\![\mathsf{ins}]\!]_L^{\mathtt{id}}) \\
[\![\mathsf{ins}]\!]_L & = & [\![\mathsf{ins}]\!]_L^{\mathtt{id}} \\
\\
[\![r_d := \mathsf{op},\ L']\!]_L^{\mathtt{id}} & = & r_d := [\![\mathsf{op}]\!]_L^{\mathtt{op}},\ L' \\
[\![\mathsf{cmp}\ ?\ L_t : L_f]\!]_L^{\mathtt{id}} & = & \begin{cases} \mathsf{nop},\ L_t & \text{when } [\![\mathsf{cmp}]\!]_L^{\mathtt{cmp}} = \top \\ \mathsf{nop},\ L_f & \text{when } [\![\mathsf{cmp}]\!]_L^{\mathtt{cmp}} = \bot \\ [\![\mathsf{cmp}]\!]_L^{\mathtt{cmp}}\ ?\ L_t : L_f & \text{else} \end{cases} \\
[\![\mathsf{ins}]\!]_L^{\mathtt{id}} & = & \mathsf{ins} \qquad \text{in any other cases} \\
\\
[\![r_1 + r_2]\!]_L^{\mathtt{op}} & = & \begin{cases} n & \text{if } \mathcal{A}(L, r_i) = n_i \text{ and } n = n_1 + n_2 \\ r_2 & \text{if } \mathcal{A}(L, r_1) = 0 \\ r_1 & \text{if } \mathcal{A}(L, r_2) = 0 \\ n_1 + r_2 & \text{if } \mathcal{A}(L, r_1) = n_1 \\ n_2 + r_1 & \text{if } \mathcal{A}(L, r_2) = n_2 \\ r_1 + r_2 & \text{in any other cases} \end{cases}
\end{array}
$$

Figure 12: Constant Propagation

integer n s.t. $n = n_1 + n_2$. Instead of this, if only one register is known to be equal to 0 the compiler replaces the addition operation by a move of a second argument. If one register is known but not equal to 0 then the compiler uses an immediate addition operation. Similar kind of optimizations are done for other operations in the language, and are not shown in Figure 12.

B.1.2 Description of certificate translation

Suppose we have a certified function f with declaration $\{\vec{x};\ \varphi;\ c;\ \psi;\ \lambda;\ \vec{\Lambda}\}$. After applying constant propagation we get a function $\overline{f}$ and we are interested on building its corresponding certificates $\overline{\lambda}$ and $\overline{\vec{\Lambda}}$.

To build $\overline{\vec{\Lambda}}$, for each instruction of the form $\overline{f}[L] = (\phi \wedge \mathrm{EQ}_{\mathcal{A}},\ \mathsf{ins})$ we have to find a proof for $\vdash (\phi \wedge \mathrm{EQ}_{\mathcal{A}}) \Rightarrow \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L)$. To this end, we use an auxiliary function T_L^{ins} of type $\forall L,\ \mathcal{P}(\vdash \mathsf{vcg}_f^{\mathtt{id}}(L) \Rightarrow \mathsf{vcg}_{f_A}(L) \Rightarrow \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L))$ that will be defined later. Let $\Gamma = [\mathsf{vcg}_{\overline{f}}(L)]$ in:

$$
\begin{array}{l}
p_1 := \mathsf{axiom}(\Gamma, \mathsf{vcg}_{\overline{f}}(L)) :\ \Gamma \vdash \mathsf{vcg}_{\overline{f}}(L) \\
p_2 := \mathsf{elim}^{\mathsf{l}}_{\wedge}(p_1) :\ \Gamma \vdash \mathsf{vcg}_f(L) \\
p_3 := \mathsf{elim}^{\mathsf{r}}_{\wedge}(p_1) :\ \Gamma \vdash \mathsf{vcg}_{f_A}(L) \\
p_4 := \mathsf{weak}(\Gamma, \vec{\Lambda}(L)) :\ \Gamma \vdash \mathsf{vcg}_f(L) \Rightarrow \mathsf{vcg}_f^{\mathtt{id}}(L) \\
p_5 := \mathsf{elim}_{\Rightarrow}(p_2, p_4) :\ \Gamma \vdash \mathsf{vcg}_f^{\mathtt{id}}(L) \\
p_6 := \mathsf{weak}(\Gamma, T_L^{\mathsf{ins}}(L)) :\ \Gamma \vdash \mathsf{vcg}_f^{\mathtt{id}}(L) \Rightarrow \mathsf{vcg}_{f_A}(L) \Rightarrow \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L) \\
p_7 := \mathsf{elim}_{\Rightarrow}(p_5, p_6) :\ \Gamma \vdash \mathsf{vcg}_{f_A}(L) \Rightarrow \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L) \\
p_8 := \mathsf{elim}_{\Rightarrow}(p_3, p_7) :\ \Gamma \vdash \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L) \\
p_9 := \mathsf{intro}_{\Rightarrow}(p_8) : \vdash \mathsf{vcg}_{\overline{f}}(L) \Rightarrow \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L)
\end{array}
$$

In the case of $\overline{\lambda}$ we can reuse λ, as φ implies $\mathsf{vcg}_f(L_{\mathsf{sp}})$, and instantiating T_L^{ins} to L_{sp} it can be shown that $\mathsf{vcg}_f(L_{\mathsf{sp}}) \Rightarrow \mathsf{vcg}_{\overline{f}}(L_{\mathsf{sp}})$, as the correctness of the certifying analyzer implies that $\mathsf{vcg}_{f_A}(L_{\mathsf{sp}})$ must be equivalent to true.

The constructor T_L^{ins}, is a complex function that semantically represents the fact that under the hypotheses that the result of the analysis is correct, if a program state satisfies $\mathsf{vcg}_f(L)$ then it will also satisfy $\mathsf{vcg}_{\overline{f}}(L)$.

Next, we show in detail its construction process:

case $f[L] = \mathsf{cmp}\ ?\ L_t : L_e$

Suppose $\mathrm{EQ}_{\mathcal{A}}(L) \Rightarrow \langle\mathsf{cmp}\rangle$ is a ring axiom. Note that $\mathsf{vcg}_{\overline{f}}(L) = \mathsf{vcg}_{\overline{f}}(L_t)$ and let

$$\begin{aligned} &p_{\mathsf{cmp}} : \vdash \mathrm{EQ}_{\mathcal{A}}(L) \Rightarrow \langle\mathsf{cmp}\rangle \\ &\Gamma = [\mathsf{vcg}_f(L), \mathsf{vcg}_{f_A}(L)] \end{aligned}$$

in:

$$\begin{aligned} p_1 &:= T_L(L_t) : \vdash \mathsf{vcg}_f(L_t) \Rightarrow \mathsf{vcg}_{f_A}(L_t) \Rightarrow \mathsf{vcg}_{\overline{f}}(L) \\ p_2 &:= \mathsf{axiom}(\Gamma, \mathsf{vcg}_f(L)) :\ \Gamma \vdash \mathsf{vcg}_f(L) \\ p_3 &:= \mathsf{elim}^{\mathsf{r}}_{\wedge}(p_2) :\ \Gamma \vdash \langle\mathsf{cmp}\rangle \Rightarrow \mathsf{vcg}_f(L_t) \\ p_4 &:= \mathsf{axiom}(\Gamma, \mathsf{vcg}_{f_A}(L)) :\ \Gamma \vdash \mathsf{vcg}_{f_A}(L) \\ p_5 &:= \mathsf{weak}(\Gamma, P_{\mathcal{A}}(L)) :\ \Gamma \vdash \mathrm{EQ}_{\mathcal{A}}(L) \Rightarrow \mathsf{vcg}^{\mathtt{id}}_{f_{\mathcal{A}}}(L) \\ p_6 &:= \mathsf{elim}^{\mathsf{r}}_{\wedge}(\mathsf{elim}_{\Rightarrow}(p_4, p_5)) :\ \Gamma \vdash \langle\mathsf{cmp}\rangle \Rightarrow \mathsf{vcg}_{f_A}(L_t) \\ p_7 &:= \mathsf{elim}_{\Rightarrow}(p_4, \mathsf{weak}(\Gamma, p_{\mathsf{cmp}})) :\ \Gamma \vdash \langle\mathsf{cmp}\rangle \\ p_8 &:= \mathsf{elim}_{\Rightarrow}(p_7, p_3) :\ \Gamma \vdash \mathsf{vcg}_f(L_t) \\ p_9 &:= \mathsf{elim}_{\Rightarrow}(p_7, p_6) :\ \Gamma \vdash \mathsf{vcg}_{f_A}(L_t) \\ p_{10} &:= \mathsf{elim}_{\Rightarrow}(p_9, \mathsf{elim}_{\Rightarrow}(p_8, \mathsf{weak}(\Gamma, p_1))) :\ \Gamma \vdash \mathsf{vcg}_{\overline{f}}(L) \\ p_{11} &:= \mathsf{intro}_{\Rightarrow}(\mathsf{intro}_{\Rightarrow}(p_{10})) : \vdash \mathsf{vcg}_f(L) \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}(L) \Rightarrow \mathsf{vcg}_{\overline{f}}(L) \end{aligned}$$

case $f[L] = r := \mathsf{op},\ L'$

Let

$$\begin{aligned} &\Gamma = [\phi_1; \mathrm{EQ}_{\mathcal{A}}(L)] \\ &\phi_1 = \mathsf{vcg}_f(L')\{r_d \leftarrow \mathsf{op}\} \\ &\phi_2 = \mathrm{EQ}_{\mathcal{A}}(L')\{r_d \leftarrow \mathsf{op}\} \\ &\phi_3 = \mathsf{vcg}_{\overline{f}}(L')\{r_d \leftarrow \mathsf{op}\} \end{aligned}$$

in:

$$\begin{aligned} p_1 &:= T_L(L') : \vdash \mathsf{vcg}_f(L') \Rightarrow \mathrm{EQ}_{\mathcal{A}}(L') \Rightarrow \mathsf{vcg}_{\overline{f}}(L') \\ p_2 &:= \mathsf{weak}(\Gamma, \mathsf{subst}(r_d, \langle\mathsf{op}\rangle, p_1)) : \Gamma \vdash \phi_1 \Rightarrow \phi_2 \Rightarrow \phi_3 \\ p_3 &:= \mathsf{axiom}(\Gamma, \phi_1) : \Gamma \vdash \phi_1 \\ p_4 &:= \mathsf{weak}(\Gamma, \vec{\Lambda}_{\mathcal{A}}(L))\Gamma \vdash \mathrm{EQ}_{\mathcal{A}}(L) \Rightarrow \phi_2 \\ p_5 &:= \mathsf{axiom}(\Gamma, \mathrm{EQ}_{\mathcal{A}}(L)) : \Gamma \vdash \mathrm{EQ}_{\mathcal{A}}(L) \\ p_6 &:= \mathsf{elim}_{\Rightarrow}(p_5, p_4) : \Gamma \vdash \phi_2 \\ p_7 &:= \mathsf{elim}_{\Rightarrow}(p_6, \mathsf{elim}_{\Rightarrow}(p_3, p_2)) : \Gamma \vdash \phi_3 \\ p_8 &:= \mathsf{weak}(\Gamma, T_{\mathsf{op}}(L, \mathsf{op})) : \Gamma \vdash \mathrm{EQ}_{\mathcal{A}}(L) \Rightarrow \langle\mathsf{op}\rangle = \langle[\![\mathsf{op}]\!]^{\mathtt{op}}_L\rangle \\ p_9 &:= \mathsf{elim}_{\Rightarrow}(p_5, p_8) : \Gamma \vdash \langle\mathsf{op}\rangle = \langle[\![\mathsf{op}]\!]^{\mathtt{op}}_L\rangle \\ p_{10} &:= \mathsf{elim}_{=}(p_9, \phi_3) : \Gamma \vdash \mathsf{vcg}_{\overline{f}}(L')\{r_d \leftarrow \langle[\![\mathsf{op}]\!]^{\mathtt{op}}_L\rangle\} \\ p_{11} &:= \mathsf{intro}_{\Rightarrow}(\mathsf{intro}_{\Rightarrow}(p_{10})) : \vdash \mathsf{vcg}^{\mathtt{id}}_f(L) \Rightarrow \mathrm{EQ}_{\mathcal{A}}(L) \Rightarrow \mathsf{vcg}^{\mathtt{id}}_{\overline{f}}(L) \end{aligned}$$

case $f[L] = r_d := g(\vec{r}),\ L'$

Let

$$\begin{aligned} p&{=}\mathsf{pre}(g)\{\vec{r_g} \leftarrow \vec{r}\}\\ q_f&{=}\mathsf{vcg}_f(L')\{r_d \leftarrow res\}\\ q_{f_\mathcal{A}}&{=}\mathsf{vcg}_{f_\mathcal{A}}(L')\{r_d \leftarrow res\}\\ q_{\overline{f}}&{=}\mathsf{vcg}_{\overline{f}}(L')\{r_d \leftarrow res\}\\ s&{=}\mathsf{post}(g)\{\vec{r_g} \leftarrow \vec{r}\}\\ \Gamma&{=}[\mathsf{vcg}_f(L), \mathsf{vcg}_{f_\mathcal{A}}(L)]\\ \Delta&{=}[s \Rightarrow q_f, s \Rightarrow q_{f_\mathcal{A}}, s] \end{aligned}$$

in

$$\begin{aligned} p_1&{:=}\mathsf{axiom}(\Gamma, \mathrm{EQ}_\mathcal{A}(L)) : \Gamma \vdash \mathrm{EQ}_\mathcal{A}(L)\\ p_2&{:=}\mathsf{elim}_\Rightarrow(p_1, \mathsf{weak}(\vec{\Lambda}_\mathcal{A}(L))) : \Gamma \vdash \mathsf{vcg}^{\mathtt{id}}_{f_\mathcal{A}}(L)\\ p_3&{:=}\mathsf{elim}^{\mathsf{r}}_\wedge(p_2) : \Gamma \vdash p\\ p_4&{:=}\mathsf{elim}^{\mathsf{l}}_\wedge(p_2) : \Gamma \vdash \forall res.(s \Rightarrow q_{f_\mathcal{A}})\\ p_5&{:=}\mathsf{elim}_\forall(p_4) : \Gamma \vdash s \Rightarrow q_{f_\mathcal{A}}\\ p_6&{:=}\mathsf{axiom}(\Gamma, \mathsf{vcg}_f(L)) : \Gamma \vdash \mathsf{vcg}_f(L)\\ p_7&{:=}\mathsf{elim}_\forall(\mathsf{elim}^{\mathsf{l}}_\wedge(p_6)) : \Gamma \vdash s \Rightarrow q_f\\ p_8&{:=}\mathsf{subst}(r_d, res, T_L(L')) : \vdash q_f \Rightarrow q_{f_\mathcal{A}} \Rightarrow q_{\overline{f}}\\ p_9&{:=}\mathsf{weak}(\Delta, p_8) : \Delta \vdash q_f \Rightarrow q_{f_\mathcal{A}} \Rightarrow q_{\overline{f}}\\ p_{10}&{:=}\mathsf{axiom}(\Delta, s) : \Delta \vdash s\\ p_{11}&{:=}\mathsf{axiom}(\Delta, s \Rightarrow q_f) : \Delta \vdash s \Rightarrow q_f\\ p_{12}&{:=}\mathsf{axiom}(\Delta, s \Rightarrow q_{f_\mathcal{A}}) : \Delta \vdash s \Rightarrow q_{f_\mathcal{A}}\\ p_{13}&{:=}\mathsf{elim}_\Rightarrow(p_{10}, p_{11}) : \Delta \vdash q_f\\ p_{14}&{:=}\mathsf{elim}_\Rightarrow(p_{10}, p_{12}) : \Delta \vdash q_{f_\mathcal{A}}\\ p_{15}&{:=}\mathsf{elim}_\Rightarrow(p_{14}, \mathsf{elim}_\Rightarrow(p_{13}, p_9)) : \Delta \vdash q_{\overline{f}}\\ p_{16}&{:=}\mathsf{intro}_\Rightarrow(\mathsf{intro}_\Rightarrow(\mathsf{intro}_\Rightarrow(p_{15}))) : \vdash (s \Rightarrow q_f) \Rightarrow (s \Rightarrow q_{f_\mathcal{A}}) \Rightarrow (s \Rightarrow q_{\overline{f}})\\ p_{17}&{:=}\mathsf{elim}_\Rightarrow(p_5, \mathsf{elim}_\Rightarrow(p_7, \mathsf{weak}(\Gamma, p_{16}))) : \Gamma \vdash s \Rightarrow q_{\overline{f}}\\ p_{18}&{:=}\mathsf{intro}_\wedge(p_3, \mathsf{intro}_\forall(p_{17})) : \Gamma \vdash p \wedge \forall res.s \Rightarrow q_{\overline{f}}\\ p_{19}&{:=}\mathsf{intro}_\Rightarrow(\mathsf{intro}_\Rightarrow(p_{18})) : \vdash \mathsf{vcg}_f(L) \Rightarrow \mathsf{vcg}_{f_\mathcal{A}}(L) \Rightarrow \mathsf{vcg}_{\overline{f}}(L) \end{aligned}$$

case $f[L] = \mathsf{return}\ r$

Note that, in this case, $\mathsf{vcg}^{\mathtt{id}}_f(L)$ and $\mathsf{vcg}^{\mathtt{id}}_{\overline{f}}(L)$ are both equal to $\mathsf{post}(f)\{res \leftarrow r\}$.

Let

$$\Gamma{=}[\mathsf{vcg}^{\mathtt{id}}_f(L), \mathsf{vcg}_{f_\mathcal{A}}(L)]$$

in

$$\begin{aligned} p_1&{:=}\mathsf{axiom}(\Gamma, \mathsf{vcg}^{\mathtt{id}}_f(L)) : \Gamma \vdash \mathsf{vcg}^{\mathtt{id}}_f(L)\\ p_2&{:=}\mathsf{intro}_\Rightarrow(\mathsf{intro}_\Rightarrow(p_1)) : \vdash \mathsf{vcg}^{\mathtt{id}}_f(L) \Rightarrow \mathsf{vcg}_{f_\mathcal{A}}(L) \Rightarrow \mathsf{vcg}^{\mathtt{id}}_{\overline{f}}(L) \end{aligned}$$

case $f[L] = \mathsf{nop},\ L'$

Trivially, $T_L(L) = T_L(L')$

B.2 Certificate Translation for *loop induction variable strength reduction*

Let $\{\vec{x};\ \varphi;\ c;\ \psi;\ \lambda;\ \vec{\Lambda}\}$ be the declaration for function f and let $\overline{f}$ be the resulting function after applying the optimization.

Again, we need to give explicitly a proof of $\vdash \mathsf{vcg}_{\overline{f}}(L) \Rightarrow \mathsf{vcg}^{\mathsf{id}}_{\overline{f}}(L)$ for each instruction of the form $\overline{f}[L] = (\phi \wedge \mathrm{EQ}_{\mathcal{A}}(L),\ \mathsf{ins})$. We can avoid repeating the main process as it is the same that was specified in the previous section (constant propagation). However it remains to define T^{ins}_{L} as the transformation performed is clearly different. To this end, we will need also to define two other auxiliary functions. In summary:

$$\begin{aligned}
&T_L : \forall L,\ \mathcal{P}(\vdash \mathsf{vcg}_f(L) \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}(L) \Rightarrow \mathsf{vcg}_{\overline{f}}(L))\\
&T^{\mathsf{ins}}_{L} : \forall L, \forall \mathsf{ins},\ \mathcal{P}(\vdash \mathsf{vcg}^{\mathsf{id}}_{f}(\mathsf{ins}, L) \Rightarrow \mathsf{vcg}^{\mathsf{id}}_{f_{\mathcal{A}}}(\mathsf{ins}, L) \Rightarrow \mathsf{vcg}^{\mathsf{id}}_{\overline{f}}(\mathsf{ins}, L))\\
&T^{\overline{\mathsf{ins}}}_{L} : \forall L,\ \mathcal{P}(\vdash \mathsf{vcg}^{\mathsf{id}}_{f}(L) \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}(L) \Rightarrow \mathsf{vcg}^{\mathsf{id}}_{\overline{f}}(L))
\end{aligned}$$

Proofs of T_L, T^{ins}_{L} and $T^{\overline{\mathsf{ins}}}_{L}$ must be derived simultaneously. If $f[L]$ does not contain an assertion, $T_L(L)$ is exactly $T^{\overline{\mathsf{ins}}}_{L}(L)$, otherwise $T_L(L)$ can be derived from $T^{\overline{\mathsf{ins}}}_{L}(L)$ and application of the certificate of $f_{\mathcal{A}}$. As T^{ins}_{L} can be shown equivalent to T_L, it remains to show how to build $T^{\overline{\mathsf{ins}}}_{L}$.

If $\overline{f}[L]$ is equal to $f[L]$ then it is not difficult to get $T^{\overline{\mathsf{ins}}}_{L}(L)$ from $T_L(L)$, therefore, we focus only on the following cases of $T^{\overline{\mathsf{ins}}}_{L}$:

Proof.

- Case $f[L_H]$.

$$\begin{aligned}
&p_1{:=}T^{\mathsf{ins}}_{L}(\mathsf{ins}, L_{H'})\\
&\qquad\qquad :\vdash \mathsf{vcg}^{\mathsf{id}}_{f}(L_H) \Rightarrow \mathsf{vcg}^{\mathsf{id}}_{f_{\mathcal{A}}}(L_H) \Rightarrow \mathsf{vcg}^{\mathsf{id}}_{\overline{f}}(L_{H''})\\
&p_2{:=}\mathsf{axiom}(\Gamma, \mathsf{vcg}^{\mathsf{id}}_{f}(L_H)) :\ \Gamma \vdash \mathsf{vcg}^{\mathsf{id}}_{f}(L_H)\\
&p_3{:=}\mathsf{axiom}(\Gamma, \mathsf{vcg}_{f_{\mathcal{A}}}(L_H)) :\ \Gamma \vdash \mathsf{vcg}_{f_{\mathcal{A}}}(L_H)\\
&p_4{:=}\mathsf{weak}(\Gamma, \vec{\Lambda}_{\mathcal{A}}(L_H)) :\ \Gamma \vdash \mathsf{vcg}_{f_{\mathcal{A}}}(L_H) \Rightarrow \mathsf{vcg}^{\mathsf{id}}_{f_{\mathcal{A}}}(L_H)\\
&p_5{:=}\mathsf{elim}_{\Rightarrow}(p_3, p_4) :\ \Gamma \vdash \mathsf{vcg}^{\mathsf{id}}_{f_{\mathcal{A}}}(L_H)\\
&p_6{:=}\mathsf{elim}_{\Rightarrow}(p_2, \mathsf{weak}(\Gamma, p_1))\\
&\qquad\qquad\qquad : \Gamma \vdash \mathsf{vcg}^{\mathsf{id}}_{f_{\mathcal{A}}}(L_H) \Rightarrow \mathsf{vcg}^{\mathsf{id}}_{\overline{f}}(L_{H''})\\
&p_7{:=}\mathsf{elim}_{\Rightarrow}(p_5, p_6) :\ \Gamma \vdash \mathsf{vcg}^{\mathsf{id}}_{\overline{f}}(L_{H''})\\
&p_8{:=}\mathsf{subst}(r'_j, b * r_i, p_7) :\ \Gamma \vdash \mathsf{vcg}^{\mathsf{id}}_{\overline{f}}(L_{H''})\{r'_j \leftarrow b * r_i\}\\
&p_9{:=}\mathsf{intro}_{\Rightarrow}(\mathsf{intro}_{\Rightarrow}(p_8))\\
&\qquad\qquad :\vdash \mathsf{vcg}^{\mathsf{id}}_{f}(L_H) \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}(L_H) \Rightarrow \mathsf{vcg}^{\mathsf{id}}_{\overline{f}}(L_H)
\end{aligned}$$

- Case $f[L_i] = r_i := r_i + c,\ L'_i$.

Let $n = b * c$ in:

$$\begin{array}{l}
p_1{:=}T_L(L_i') : \vdash \mathsf{vcg}_f(L_i') \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}(L_i') \Rightarrow \mathsf{vcg}_{\overline{f}}(L_i') \\
p_2{:=}\mathsf{subst}(r_j', r_j'{+}n, p_1) \\
\qquad\qquad : \vdash \mathsf{vcg}_f(L_i') \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}(L_i'') \Rightarrow \mathsf{vcg}_{\overline{f}}(L_i'') \\
p_3{:=}\mathsf{subst}(r_i, r_i{+}c, p_2) \\
\qquad\qquad : \vdash \mathsf{vcg}_f^{\mathtt{id}}(L_i) \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}^{\mathtt{id}}(L_i) \Rightarrow \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L_i) \\
p_4{:=}\mathsf{axiom}(\Gamma, \mathsf{vcg}_f^{\mathtt{id}}(L_i)) : \Gamma \vdash \mathsf{vcg}_f^{\mathtt{id}}(L_i) \\
p_5{:=}\mathsf{axiom}(\Gamma, \mathsf{vcg}_{f_{\mathcal{A}}}(L_i)) : \Gamma \vdash \mathsf{vcg}_{f_{\mathcal{A}}}(L_i) \\
p_6{:=}\mathsf{elim}_{\Rightarrow}(p_5, \mathsf{weak}(\Gamma, \vec{\Lambda}_{\mathcal{A}}(L_i))) : \Gamma \vdash \mathsf{vcg}_{f_{\mathcal{A}}}^{\mathtt{id}}(L_i) \\
p_7{:=}\mathsf{weak}(\Gamma, p_3) \\
\qquad\qquad : \Gamma \vdash \mathsf{vcg}_f^{\mathtt{id}}(L_i) \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}^{\mathtt{id}}(L_i) \Rightarrow \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L_i) \\
p_8{:=}\mathsf{elim}_{\Rightarrow}(p_4, p_7) : \Gamma \vdash \mathsf{vcg}_{f_{\mathcal{A}}}^{\mathtt{id}}(L_i) \Rightarrow \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L_i) \\
p_9{:=}\mathsf{elim}_{\Rightarrow}(p_6, p_8) : \Gamma \vdash \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L_i) \\
p_{10}{:=}\mathsf{intro}_{\Rightarrow}(\mathsf{intro}_{\Rightarrow}(p_9)) : \vdash \mathsf{vcg}_f^{\mathtt{id}}(L_i) \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}(L_i) \Rightarrow \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L_i)
\end{array}$$

- Case $f[L_j] = r_j := b * r_i,\ L_j'$.

  Let $\phi_1 = \mathsf{vcg}_{\overline{f}}(L_j')\{r_j \leftarrow b * r_i\}$.

$$\begin{array}{l}
p_1{:=}T_L(L_j') : \vdash \mathsf{vcg}_f(L_j') \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}(L_j') \Rightarrow \mathsf{vcg}_{\overline{f}}(L_j') \\
p_2{:=}\mathsf{subst}(r_j, b{*}r_i, p_1) : \vdash \mathsf{vcg}_f^{\mathtt{id}}(L_j) \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}^{\mathtt{id}}(L_j) \Rightarrow \phi_1 \\
p_3{:=}\mathsf{axiom}(\Gamma, \mathsf{vcg}_f^{\mathtt{id}}(L_j)) : \Gamma \vdash \mathsf{vcg}_f^{\mathtt{id}}(L_j) \\
p_4{:=}\mathsf{axiom}(\Gamma, \mathsf{vcg}_{f_{\mathcal{A}}}(L_j)) : \Gamma \vdash \mathsf{vcg}_{f_{\mathcal{A}}}(L_j) \\
p_5{:=}\mathsf{weak}(\Gamma, \vec{\Lambda}(L_j)) : \Gamma \vdash \mathsf{vcg}_{f_{\mathcal{A}}}(L_j) \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}^{\mathtt{id}}(L_j) \\
p_6{:=}\mathsf{elim}_{\Rightarrow}(p_4, p_5) : \Gamma \vdash \mathsf{vcg}_{f_{\mathcal{A}}}^{\mathtt{id}}(L_j) \\
p_7{:=}\mathsf{elim}_{\Rightarrow}(p_3, \mathsf{weak}(\Gamma, p_2)) : \Gamma \vdash \mathsf{vcg}_{f_{\mathcal{A}}}^{\mathtt{id}}(L_j) \Rightarrow \phi_1 \\
p_8{:=}\mathsf{elim}_{\Rightarrow}(p_6, p_7) : \Gamma \vdash \mathsf{vcg}_{\overline{f}}(L_j')\{r_j \leftarrow b * r_i\} \\
p_9{:=}\mathsf{elim}_{=}(p_4, p_8) : \Gamma \vdash \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L_j) \\
p_{10}{:=}\mathsf{intro}_{\Rightarrow}(\mathsf{intro}_{\Rightarrow}(p_9)) : \vdash \mathsf{vcg}_f^{\mathtt{id}}(L_j) \Rightarrow \mathsf{vcg}_{f_{\mathcal{A}}}(L_j) \Rightarrow \mathsf{vcg}_{\overline{f}}^{\mathtt{id}}(L_j)
\end{array}$$

□

B.3 Certificate translation for *Ghost Variable Introduction*

Lemma B.1. *After applying* Ghost Variable Introduction, *the resulting function $\overline{f}$ satisfies this following property:*

$$\forall L, \mathsf{vcg}_{\overline{f}}(L) = \mathsf{vcg}_f(L)\sigma_L$$

where σ is the substitution defined in Figure 8.

Proof. The proof is by induction principle attached to the definition of reachAnnot.

- case $f[L] = (\phi,\ \mathsf{ins})$

  In this case $\mathsf{vcg}_{\overline{f}}(L)$ is equal to $\phi\sigma_L$ by definition of vcg and *ghost variable introduction.* But, as ϕ is equal to $\mathsf{vcg}_f(L)$, it is clear that the lemma is satisfied.

- case $f[L] = \mathsf{nop},\ L'$

  By definition of vcg, we have that $\mathsf{vcg}_{\overline{f}}(L)$ is equal to $\mathsf{vcg}_{\overline{f}}(L')$, which in turn is equal to $\mathsf{vcg}_f(L')\sigma_{L'}$ by inductive hypothesis. As $\mathsf{vcg}_f(L') = \mathsf{vcg}_f(L)$, it remains to prove that $\sigma_{L'}$ is in fact σ_L, but this is the case as the condition of liveness or liveness in assertion is the same for L and L' for any register.

  For example, if $\mathcal{L}(L', r) = \top$, that means that r is read at label L' or there is a path π that reaches label L'' such that r is never assigned. In the first case, we can propose $L \to L'$ as a path from L that reaches L' where r is read. And in the second case we can construct the desired path appending $L \to L'$ to π.

- case $f[L] = \mathsf{return}\ r$

  The term $\mathsf{vcg}_{\overline{f}}(L)$ is $\mathsf{post}(\overline{f})\{res \leftarrow r\}$ by definition of vcg, but as the postcondition does not change after the optimization, it can be replaced by $\mathsf{post}(f)\{res \leftarrow r\}$. In addition, σ_L does not affect r (r is live at program label L) or any other variable in $\mathsf{post}(f)$. So finally, it can be rewritten as $\mathsf{post}(f)\{res \leftarrow r\}\sigma_L$.

- case $f[L] = \mathsf{cmp}\ ?\ L_1 : L_2$

  By definition, $\mathsf{vcg}_{\overline{f}}(L)$ is $\mathsf{cmp} \Rightarrow \mathsf{vcg}_{\overline{f}}(L_1)\sigma_1 \wedge \neg\mathsf{cmp} \Rightarrow \mathsf{vcg}_{\overline{f}}(L_2)\sigma_2$ where σ_i corresponds to the substitutions generated by the series of inserted assignments on the branch that follows the label L_i. In this case, $\sigma_i = \{\hat{r} \leftarrow r \mid \mathcal{L}(L_i, r) = \top_\phi \wedge \mathcal{L}(L, r) = \top\}$. W.l.o.g. consider branch L_1. By inductive hypothesis, $\mathsf{vcg}_{\overline{f}}(L_1)\sigma_1$ is equal to $\mathsf{vcg}_f(L_1)\sigma_{L_1}\sigma_1$. If we prove that $\sigma_1 \circ \sigma_{L_1}$ is equal to σ_L for the variables that appear in $\mathsf{vcg}_f(L_1)$, and using the fact that σ_L is the identity for cmp, that implies the lemma we are trying to prove. Using the definitions of σ_1 and σ_{L_1} we know that $\sigma_1 \circ \sigma_{L_1}$ is $\{r \leftarrow \hat{r} | \mathcal{L}(L_1, r) = \top_\phi \wedge \mathcal{L}(L, r) \neq \top\}$. To see that this last expression is equal to σ_L for any variable r occurring free in $\mathsf{vcg}_f(L_1)$, first suppose that $\mathcal{L}(L_1, r) = \top_\phi$ and $\mathcal{L}(L, r) \neq \top$, in that case it is clear that $\mathcal{L}(L, r) = \top_\phi$. In the other sense, if $\mathcal{L}(L, r) = \top_\phi$ then it must be the case that $\mathcal{L}(L_1, r) = \top_\phi$, because if r appears on $\mathsf{vcg}_f(L_1)$ then $\mathcal{L}(L_1, r) = \top_\phi$ or $\mathcal{L}(L_1, r) = \top$, but this last option must be rejected. To see why it must rejected, suppose $\mathcal{L}(L_1, r) = \top$ then, again, there are two possiblities. Register r is read at label L_1 or there is a path π from L_1 to L', such that r is read in L' and is not assigned before label L'. In the first case, the path $L \to L'$ proves $\mathcal{L}(L, r) = \top$, and in the second case $L \to L'$ can be prefixed to π to build a new path and prove $\mathcal{L}(L, r) = \top$.

- case $f[L] = r_d := \mathsf{op},\ L'$ and $\mathcal{L}(L', r_d) = \bot$

  $\mathsf{vcg}_{\overline{f}}(L)$ is equal to $\mathsf{vcg}_{\overline{f}}(L')$ by the transformation performed and definition of vcg and then, by IH, equal to $\mathsf{vcg}_f(L')\sigma_{L'}$. As $\mathcal{L}(L', r_d) = \bot$ implies $r_d \notin FV(\mathsf{vcg}_f(L'))$, we can write this result as $\mathsf{vcg}_f(L')\{r_d \leftarrow \mathsf{op}\}\sigma_{L'}$, which in fact is $\mathsf{vcg}_f(L)\sigma_{L'}$ by definition of vcg. To finish the proof, it remains to show that $\sigma_{L'}$ is equal to σ_L. But this is clear, because as $\mathcal{L}(L', r_d) = \bot$, then, for any variable, the condition of live in assertion does not change through the execution of the assignment $r_d := \mathsf{op},\ L'$.

- case $f[L] = r_d := \mathsf{op},\ L'$ and $\mathcal{L}(L', r_d) = \top_\phi$

  By definition of vcg and the transformation performed we have that $\mathsf{vcg}_{\overline{f}}(L)$ is equal to $\mathsf{vcg}_{\overline{f}}(L')\{\hat{r_d} \leftarrow \mathsf{op}\sigma_L\}$, and by IH, to $\mathsf{vcg}_f(L')\sigma_{L'}\{\hat{r_d} \leftarrow \mathsf{op}\sigma_L\}$. Now we have to see that

$$\{\hat{r_d} \leftarrow \mathsf{op}\sigma_L\} \circ \sigma_{L'} = \sigma_L \circ \{r_d \leftarrow \mathsf{op}\}$$

  To prove this equation we proceed by case analysis. First, suppose that we apply both functions to register r_d, in this case it is easy to see that both expressions are equivalent to

$\mathsf{op}\sigma_L$. Now suppose that we apply them to a register $r \neq r_d$. In this case, if r is in $\mathsf{vcg}_f(L')$ then it is live or live in assertion on label L', in which case, it will also be live or live in assertion, respectively, on label L. If r is live in assertion on label L, and r occurs free in $\mathsf{vcg}_f(L')$, then it must also be the case that r is live in assertion on label L', because if it is not the case, and r occurs in $\mathsf{vcg}_f(L')$, then r must be live on label L', which implies that is is also live on label L, and that is a contradiction.

- case $f[L] = r_d := \mathsf{op},\ L'$ and $\mathcal{L}(L', r_d) = \top$

  Starting with $\mathsf{vcg}_{\overline{f}}(L)$ and applying definition of vcg, we get $\mathsf{vcg}_{\overline{f}}(L'')\{r_d \leftarrow \mathsf{op}\}$, where in label L'' the transformation inserts a series of ghost assignments. If we continue with the application of the definition of vcg, we get $\mathsf{vcg}_{\overline{f}}(L')\sigma_1\{r_d \leftarrow \mathsf{op}\}$, where σ_1 represents the substitutions performed by the ghost assignments. By inductive hypothesis this result is in fact equal to $\mathsf{vcg}_f(L')\sigma_{L'}\sigma_1\{r_d \leftarrow \mathsf{op}\}$ and using the fact that

  $$\{r_d \leftarrow \mathsf{op}\} \circ \sigma_1 \circ \sigma_{L'} = \sigma_L \circ \{r_d \leftarrow \mathsf{op}\}$$

  we can get $\mathsf{vcg}_f(L')\{r_d \leftarrow \mathsf{op}\}\sigma_L$ that is what we want to prove. It remains to see that the last equation is correct.

  In case we apply this mappings to r_d, we get the expression op in both sides of the equation. But in the case of $r \neq r_d$, if $r \notin \mathsf{op}$ then $\{r_d \leftarrow \mathsf{op}\}(\sigma_1(\sigma_{L'}r))$ is equal to $\sigma_1(\sigma_{L'}r)$, which is equal to $\sigma_L r$, and $\sigma_L(\{r_d \leftarrow \mathsf{op}\}r)$. And if $r \in \mathsf{op}$ then $\mathcal{L}(L, r) = \top$ and $\sigma_L r = r$, but in this case if $\sigma_{L'} r = \hat{r}$ because of $\mathcal{L}(L', r) = \top_\phi$ then $\sigma_1(\sigma_{L'} r) = r$.

- case $f[L] = r_d := g(r),\ L'$

  By definition of vcg, we have that $\mathsf{vcg}_{\overline{f}}(L)$ is equal to

  $$\mathsf{pre}(g)\{\vec{r}_g \leftarrow \vec{r}\} \wedge (\forall res.\mathsf{post}(g)\{\vec{r}_g^{\,*} \leftarrow r\} \Rightarrow \mathsf{vcg}_{\overline{f}}(L'')\{r_d \leftarrow res\})$$

  For simplicity, we will focus on the subterm $\mathsf{vcg}_{\overline{f}}(L'')\{r_d \leftarrow res\}$. Ghost variable introduction inserts on label L'' some ghost assignments such that this last expression is equal to $\mathsf{vcg}_{\overline{f}}(L')\sigma_1\{r_d \leftarrow res\}$, where $\sigma_1 = \{\hat{t} \leftarrow t | \mathcal{L}(L, t) = \top \wedge \mathcal{L}(L', t) = \top_\phi\}$. By inductive hypothesis this is also equal to $\mathsf{vcg}_f(L')\sigma_{L'}\sigma_1\{r_d \leftarrow res\}$. Now he have to use again the property that says that $\sigma_1 \circ \sigma_{L'}$ is equal to σ_L for any variable occurring in $\mathsf{vcg}_f(L')$, and its proof can be found in cases above. Therefore our expression is equivalent to $\mathsf{vcg}_f(L')\sigma_L\{r_d \leftarrow res\}$ and, as $\mathcal{L}(L, r_d) \neq \top_\phi$, that is also equal to $\mathsf{vcg}_f(L')\{r_d \leftarrow res\}\sigma_L$. In conclusion the whole proposition is equal to

  $$\mathsf{pre}(g)\{\vec{r}_g \leftarrow \vec{r}\} \wedge (\forall res.\mathsf{post}(g)\{\vec{r}_g^{\,*} \leftarrow r\} \Rightarrow \mathsf{vcg}_f(L')\{r_d \leftarrow res\}\sigma_L)$$

  and finally if we consider that the only registers in this phrase that are in the domain of σ_L are $\vec{r}$, using the fact that $\mathcal{L}(L, \vec{r}) = \top$, we can move σ_L to the outermost place of the expression to get what we were looking for.

$\square$