Certificate Translation for Optimizing Compilers*

(Extended Abstract)

Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk

INRIA Sophia-Antipolis, Project EVEREST
{Gilles.Barthe, Benjamin.Gregoire, Cesar.Kunz, Tamara.Rezk}@sophia.inria.fr

Abstract. Certifying compilation provides a means to ensure that untrusted mobile code satisfies its functional specification. A certifying compiler generates code as well as a machine-checkable "certificate", i.e. a formal proof that establishes adherence of the code to specified properties. While certificates for safety properties can be built fully automatically, certificates for more expressive and complex properties often require the use of interactive code verification. We propose a technique to provide code consumers with the benefits of interactive source code verification. Our technique, certificate translation, extends program transformations by offering the means to turn certificates of functional correctness for programs in high-level languages into certificates for executable code. The article outlines the principles of certificate translation, using specifications written in first order logic. This translation is instantiated for standard compiler optimizations in the context of an intermediate RTL Language.

1 Introduction

Program verification environments provide a means to establish that programs meet their specifications, and are increasingly being used to validate safety-critical or security-critical software. Most often, such environments target high-level languages. However it is usually required to achieve correctness guarantees for compiled programs, especially in the context of mobile code—because code consumers may not have access to the source program or, if they do, may not trust the compiler. Yet there is currently no mechanism for bringing the benefits of interactive source code verification to code consumers. The objective of our work is precisely to propose such a mechanism, called *certificate translation*, which allows us to transfer evidence from source programs to compiled programs.

The starting point of our work is Proof Carrying Code (PCC) [9], which provides a means to establish trust in a mobile code infrastructure, by requiring that mobile code is sent along with a formal proof (a.k.a. certificate) showing its adherence to a property agreeable by the code consumer. While PCC does not

* This work is partially funded by the IST European Project Mobius.

K. Yi (Ed.): SAS 2006, LNCS 4134, pp. 301–317, 2006.

preclude generating certificates from interactive verification of source programs, the prominent approach to certificate generation is *certifying compilation* [11], which constructs automatically certificates for safety properties such as memory safety or type safety. Certifying compilation is by design restricted to a specific class of properties and programs—a deliberate choice of the authors [11] whose primary goal was to reduce the burden of verification on the code producer side. In contrast, certificate translation is by design very general and can be used to enforce arbitrary properties on arbitrary programs. Of course, generality comes at the cost of automation, so we must assume that programs have been annotated and proved interactively.

Thus the primary goal of certificate translation is to transform certificates of original programs into certificates of compiled programs. Given a compiler $[\![.]\!]$, a function $[\![.]\!]_{\text{spec}}$ 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 programs p, policies ϕ, and certificates c,

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

The paper outlines the principles of certificate translation, and illustrates its mechanisms in the context of an an optimizing compiler for a Register Transfer Language (RTL). The compiler proceeds in a step by step fashion. For each optimization 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 a non-optimizing compiler is relatively simple to construct since proof obligations are preserved (up to minor differences). Dealing with optimizations is more challenging. The major difficulty arises from the fact that certificate translators for optimizations often take as argument, in addition to the certificate of the original program, a certificate of the results of the analysis that justifies the optimization. In order to enable such an aggregation, one must therefore express the results of the analysis in the logic of the PCC architecture, and enhance the analyzer so that it produces a certificate of the analysis for each program. The overall architecture of a certificate translator is given in Figure 1.

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). In section 5 we compare our work with recent related developments. We conclude in Section 6 with future work.

2 Setting

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

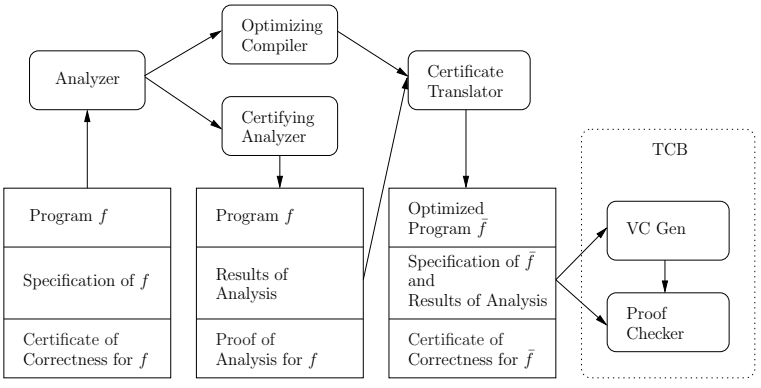

Fig. 1. Overall picture of certificate translation

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

Fig. 2. Syntax of RTL

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), 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 $\boldsymbol{r}$, a precondition φ, a (closed) graph code G, a postcondition ψ, a certificate λ, and a function $\boldsymbol{\Lambda}$ 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.

As will be defined below, the VCGen generates one proof obligation for each program point containing an annotation and one proof obligation for the entry point L_{sp}. The component λ certifies the latter proof obligation and $\boldsymbol{\Lambda}$ maps every program point that contains and assertion to the proof of its related proof obligation.

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 $\boldsymbol{r}$ 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 $\boldsymbol{r}^*$ denote the set $\{r^* \in \mathcal{R} \mid r \in \boldsymbol{r}\}$. 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 $\boldsymbol{r}$; 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[1] in which the only registers to occur are res and registers from $\boldsymbol{r}^*$; 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, $\mathsf{succ}_f(L) = \emptyset$.

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 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*

[1] Notice that a postcondition is not exactly an assertion in the sense that it uses register names from $\boldsymbol{r}^*$, which must not appear in preconditions or annotations of the program.

the annotation is the post condition), or if all its immediate successors reach annotated labels:

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

– *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 cases: 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 $\mathsf{vcg}_f^{\mathtt{id}}$.

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

Fig. 3. Verification condition generator

The formal definitions of vcg_f and $\mathsf{vcg}_f^{\mathtt{id}}$ 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 $\mathsf{vcg}_f^{\mathtt{id}}$ 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, $\mathsf{vcg}_f^{\mathtt{id}}(r_d := g(\boldsymbol{r}),\ L)$ 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 $\forall res.\ \mathsf{post}(g)\{\boldsymbol{r}_g^* \leftarrow \boldsymbol{r}\} \Rightarrow \mathsf{vcg}_f(L)\{r_d \leftarrow res\}$. The second conjunct permits that information in $\mathsf{vcg}_f(L)$ about registers different from r_d is propagated to other preconditions. In the remainder 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.

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 (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.

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

Fig. 4. Proof Algebra (excerpts)

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.

Definition 2 (Certified Program)

- *A function f with declaration $\{\boldsymbol{r};\ \varphi;\ G;\ \psi;\ \lambda;\ \boldsymbol{\Lambda}\}$ is certified if:*
  - *λ is a proof of $\vdash \varphi \Rightarrow \mathsf{vcg}_f(L_{\mathsf{sp}})\{\boldsymbol{r}^* \leftarrow \boldsymbol{r}\}$,*
  - *$\boldsymbol{\Lambda}(L)$ 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.*

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.

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.

Certificate translation is tightly bound to the optimizations considered. According to different optimizations, certificate translators fall in one of the three categories:

- PPO/IPO (Preservation/Instantiation of Proof Obligations): PPO deals with transformations for which the annotations are not rewritten, and where the proof obligations (for the original and transformed programs) coincide. This category covers transformations such as non-optimizing compilation and unreachable code elimination. IPO deals with transformations where the annotations and proof obligations for the transformed program are instances of annotations and proof obligations for the original program, thus certificate translation amounts to instantiating certificates. This category covers dead register elimination and register allocation;
- 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;
- CTCA (Certificate Translation with Certifying Analyzers): CTCA deals with transformations for which the annotations need to be rewritten, and for which certificate translation relies on having certified previously the analysis results used by the transformation. This category covers constant propagation, common subexpression elimination, loop induction, and other optimizations that rely on arithmetic.

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:

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

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 $\bar{f}$.

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

Example 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:

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

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

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

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 5. If we do not rewrite assertions, that is we let $\xi_{cp} = \xi$ and $\varphi_{cp} = \varphi$ then 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. *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

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

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 generate the certificate post-analysis. The merits of both approaches will be reported elsewhere; here we have followed the second approach.

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.

(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 :	$r_i := 0,\ L_2$
$L_2 : \xi_{cp},$	$r_1 := b,\ L_3$	$L_2 : \xi_{li},$	$r_1 := b,\ L_3$	$L_2 : \xi_{dr},$	$\mathsf{set}\ \hat{r_1} := b,\ L_3$
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 :	$\mathsf{set}\ \hat{r_j} := r'_j,\ L_5$
$L_5 : \varphi_{cp},$	$(r_i = n)\ ?\ L_6 : L_3$	$L_5 : \varphi_{li},$	$(r_i = n)\ ?\ L_6 : L'_3$	$L_5 : \varphi_{dr},$	$(r_i = n)\ ?\ L_6 : L'_3$
L_6 :	$\mathsf{return}\ r_j$	L_6 :	$\mathsf{return}\ r_j$	L_6 :	$\mathsf{return}\ r'_j$

Fig. 5. Example of different optimizations

4 Instances of Certificate Translation

This section provides instances of certificate translations 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 Constant Propagation

Goal. Constant propagation aims at minimizing 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 $\mathcal{A} : \mathcal{PP} \times \mathcal{R} \to \mathbb{Z}_\perp$ ($\mathcal{PP}$ denoting the set of program points) 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 $\mathrm{RES}_{\mathcal{A}}(L)$):

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

To derive a certificate for the analysis we have to prove that, for each reachable label L,

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

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^{ins}_L is defined by case analysis, using the fact that the transformation of operations is correct relative to the results of the analysis:

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

The expression $\langle [\![\mathsf{op}]\!]^{\mathsf{op}}_L \rangle$ represents the substitution of variables by constants in op. 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 2. Recall function f, defined in Example 1. Using the compiler and transforming the assertions as explained before, we obtain the optimized program shown in Figure 5 (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.

4.2 Loop Induction

Goal. Loop induction register strength reduction 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.

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

Fig. 6. Loop Induction

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:

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

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 6 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 3. Applying loop induction to program (a) in Figure 5, we obtain program (b) where m denotes the result of the product $b * c$ and $\xi_{li} \triangleq \xi_{cp}$ and $\varphi_{li} \triangleq \varphi_{cp} \wedge r'_j = b * r_i$.

4.3 Dead Register Elimination

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 disjoint). 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 $\mathsf{dead}_c(L, L') = \{r | \mathcal{L}(L, r) = \top \wedge \mathcal{L}(L', r) = \top_\phi\}$.

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

The analysis $\mathsf{ghost}_L^{\mathtt{id}}(\mathsf{ins})$ is defined in Figure 7. We use $\mathsf{set}\ \hat{\boldsymbol{r}} := \boldsymbol{r},$ as syntactic sugar for a sequence of assignments $\mathsf{set}\ \hat{r_i} := r_i,$ where for each register r_i in $\boldsymbol{r}$, $\hat{r_i}$ in $\hat{\boldsymbol{r}}$ 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.

$$
\begin{array}{ll}
\mathsf{ghost}^{\mathsf{id}}_L(\mathsf{return}\ r) & = \mathsf{return}\ r \\
\mathsf{ghost}^{\mathsf{id}}_L(r_d := f(\boldsymbol{r}),\ L') & = \left|\begin{array}{l} L\ : r_d := f(\boldsymbol{r}),\ L'' \\ L'' : \mathsf{set}\ \hat{\boldsymbol{t}} := \boldsymbol{t},\ L' \quad \text{for each t } \in \mathsf{dead}_c(L, L') \end{array}\right. \\
\mathsf{ghost}^{\mathsf{id}}_L(\mathsf{nop},\ L') & = \mathsf{nop},\ L' \\
\mathsf{ghost}^{\mathsf{id}}_L(\mathsf{cmp}\ ?\ L_1 : L_2) & = \left|\begin{array}{l} L\ : \mathsf{cmp}\ ?\ L_1' : L_2' \\ L_1' : \mathsf{set}\ \hat{\boldsymbol{t}}_1 := \boldsymbol{t}_1,\ L_1 \quad \text{where} \boldsymbol{t_1} = \mathsf{dead}_c(L, L_1) \\ L_2' : \mathsf{set}\ \hat{\boldsymbol{t}}_2 := \boldsymbol{t}_2,\ L_2 \quad \text{where } \boldsymbol{t_2} = \mathsf{dead}_c(L, L_2) \end{array}\right. \\
\mathsf{ghost}^{\mathsf{id}}_L(r_d := \mathsf{op},\ L') & = \mathsf{nop},\ L' \quad \text{if } \mathfrak{L}(L', r_d) = \bot \\
& = \mathsf{set}\ \hat{r}_d := \mathsf{op}\sigma_L,\ L' \qquad \text{if } \mathfrak{L}(L', r_d) = \top_\phi \\
& = \left|\begin{array}{ll} L & : r_d := \mathsf{op},\ L'' \qquad\qquad \text{if } \mathfrak{L}(L', r_d) = \top \\ L'' & \mapsto \mathsf{set}\ \hat{\boldsymbol{t}} := \boldsymbol{t},\ L' \\ & \quad \text{where } \boldsymbol{t} = \mathsf{dead}_c(L, L') \end{array}\right.
\end{array}
$$

Fig. 7. Ghost Variable Introduction-Dead Register Elimination

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

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. $\mathfrak{L}(L, r) = \top$.

Example 4. In Figure 5, applying first copy propagation to program (b), 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 Related Work

Certified Compilation. Compiler correctness [6] 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* [8] advocates the use of a proof assistant for machine-checking compiler correctness results. Section 2 of [8] shows that it is theoretically possible to derive certificate translation from certifying compilation. However, we think that the approach is restrictive and unpractical:

- certificates encapsulate the definition of the compiler and its correctness proof on the one hand, and the source code and its certificate on the other hand. Thus certificates are large and costly to check;

- with the above notion of certified compiler, the approach is necessarily confined to properties about the input/output behavior of programs, and rules out interesting properties involving intermediate program points that are expressed with assertions or ghost variables;
- and a further difficulty with this approach is that it requires that the source code is accessible to the code consumer, which is in general not the case.

For similar reasons, it is not appropriate to take as certificates of optimized programs pairs that consist of a certificate for the unoptimized program and of a proof that the optimizations are semantics preserving.

Certifying Compilation. Certifying compilation is concerned with generating automatically safety certificates. The Touchstone compiler [11] is a notable example of certifying compiler, which generates type-safety certificates for a fragment of C. In Chapter 6 of [10], Necula studies the impact of program optimizations on certifying compilation. For most standard optimizations an informal analysis is made, indicating whether the transformation requires reinforcing the program invariants, or whether the transformation does not change proof obligations.

There are many commonalities between his work and ours, but also some notable differences. First, the VCGen used by Necula propagates invariants backwards, whereas ours generates a proof obligation for each invariant. This has subtle implications on the modifications required for the invariant. A main difference is that we not only have to strengthen invariants, but also transform the certificate; further, when he observes that the transformation produces a logically equivalent proof obligation, we have to define a function that maps proofs of the original proof obligation into proofs of the new proof obligation after optimization.

Provable Optimizations through Sound Elementary Rules. Rhodium [7] is a domain-specific language for declaring and proving correct program optimizations. The domain-specific language is used to declare local transformation rules and to combine them into the optimization. Transformations written in Rhodium are given a semantic interpretation that is used to generate sufficient conditions for the correctness of the transformation. The proof obligations are in turn submitted to an automatic prover that attempts to discharge them automatically. The idea also underlies the work of Benton [4], who proposes to use a relational Hoare logic to justify transformation rules from which optimizations can be built. The perspective of decomposing optimizations through sound elementary rules is appealing, but left for future work.

Spec# and BML Project. The Spec# project [2] 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, but does not attempt to formalize this relation. There is no notion

of certificate, and thus no need to transform them. A similar line of work for Java was pursued independently by Pavlova and Burdy [5] who define a Bytecode Modeling Language into which annotations of the Java Modeling Language and a VCGen for annotated bytecode programs; the generated proof obligations are sent to an automatic theorem prover. They partially formalize the relation between proof obligations at source code and bytecode level, but they do not consider certificates.

In a similar spirit, Bannwart and Müller [1], 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 bytecode programs obtained by non-optimizing compilation.

Certifying Analyzers. Specific instances of certifying analyzers have been studied independently by Wildmoser, Chaieb and Nipkow [13] in the context of a bytecode language and by Seo, Yang and Yi [12] in the context of a simple imperative language. Seo, Yang and Yi propose an algorithm that automatically constructs safety proofs in Hoare logic from abstract interpretation results.

6 Concluding Remarks

Certificate translation provides a means to bring the benefits of source code verification to code consumers using PCC architectures. Certificate translation significantly extends the scope of PCC in that it allows to consider complex security policies and complex programs— at the cost of requiring interactive verification. The primary motivation for certificate translation are mobile code scenarios, possibly involving with several code producers and intermediaries, where security-sensitive applications justify interactive verification. One important constraint for these scenarios (which originate from mobile phone industry) is that only the code after compilation and optimization is available to the code consumer or a trusted third party: this assumption makes it impossible to use ideas from certified compilation, or to use as certificates for optimized programs a pair consisting of a certificate of the unoptimized program, and a proof of correctness of the optimizations.

There are many directions for future work, including:

- On a side, we would like to build a generic certificate translation, instead of developing a translator per optimization. One natural approach would be to describe standard program optimizations as a combination of more elementary transformations in the style of Rhodium.
- On a practical side, we have developed a prototype certificate translator for our RTL language. This prototype generates proof obligations for the initial program that are sent to the Coq theorem prover. Once the proofs obligations are solved, the proofs are sent to the certificate translator that automatically optimizes the program and transforms the proofs. In the medium term, we intend to extend our prototype to a mainstream programming language such as C or Java to an assembly language.

- On an experimental side, we would like to gather metrics about the size of certificates—which is an important issue, although not always central in the scenarios we have in mind. Preliminary experiments using λ-terms as certificates indicate that their size does not explode during translation, provided we perform after certificate translation a pass of reduction that eliminates all the redexes created by the translation. For example, the size of certificates remains unchanged for dead register elimination. For constant propagation, the size of certificates grows linearly w.r.t. the size of the code. There are other opportunities to reduce certificate size; in particular, not all annotations generated by certifying analyzers are used to build the certificate for the optimized program, so we could use enriched analyses with dependency information to eliminate all annotations that are not used to prove the optimized program, i.e. annotations that are not directly used to justify an optimization, and annotations that are not used (recursively) to justify such annotations;
- On an applicative side, we would like to experiment with certificate translation in realistic settings. E.g. certificate translation could 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 to conventional languages.

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, T.Rezk, and A. Saabas. Proof obligations preserving compilation. In *Proceedings of FAST'05*, volume 3866 of *Lecture Notes in Computer Science*, pages 112–126. Springer-Verlag, 2005.
4. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of POPL'04*, pages 14–25. ACM Press, 2004.
5. L. Burdy and M. Pavlova. Annotation carrying code. In *Proceedings of SAC'06*. ACM Press, 2006.
6. J. D. Guttman and M. Wand. Special issue on VLISP. *Lisp and Symbolic Computation*, 8(1/2), March 1995.
7. 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.

8. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of POPL'06*, pages 42–54. ACM Press, 2006.
9. G.C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.
10. G.C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.
11. 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.
12. 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.
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.