Certificate Translation for Optimizing Compilers

Gilles Barthe
IMDEA Software
and
Benjamin Grégoire and César Kunz and Tamara Rezk
INRIA Sophia Antipolis - Méditerranée

Proof Carrying Code provides trust in mobile code by requiring certificates that ensure the code adherence to specific conditions. The prominent approach to generate certificates for compiled code is Certifying Compilation, that automatically generates certificates for simple safety properties.

In this work, we present Certificate Translation, a novel extension for standard compilers that automatically transforms formal proofs for more expressive and complex properties of the source program to certificates for the compiled code.

The article outlines the principles of certificate translation, instantiated for a non optimizing compiler and for standard compiler optimizations in the context of an intermediate RTL Language.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]: Formal methods; F.3.2 [**Semantics of Programming Languages**]: Program analysis; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Logics of programs

General Terms: Languages, Verification, Security

Additional Key Words and Phrases: Proof-carrying Code, Program Verification, Static Analysis, Program Optimizations

1. INTRODUCTION

1.1 Background and Motivation

Program verification environments provide a mean to establish that programs meet their specifications, and are increasingly being used to validate safety-critical or security-critical software [Barnett et al. 2005; Chalin et al. 2006; Burdy et al. 2003; Barthe et al. 2007]. Typically, such program verification environments combine automated techniques such as abstract interpretation and theorem proving with interactive tools such as proof assistants. While automated theorem provers are useful to detect many common programming mistakes and sometimes to establish some simple policies, the use of interactive verification tools might be required for many policies, including basic safety policies for complex software (the complexity of the software may render relatively simple safety policies difficult to verify automatically), and complex policies that involve the functional behavior of software.

This work was partially supported by IST Project MOBIUS.

Most often, program verification environments target high level languages; for example, several program verification environments have been developed for popular programming languages such as Eiffel, Java or C. The focus on high level languages is beneficial for developers, since it enables them to gain increased confidence or feedback directly on their code, without the need to consider the particular runtime environment where the code shall be executed. There are strong arguments to verify compiled code; in particular, reasoning at the level of source code restricts the scope of properties that can be verified, and is also not appropriate in mobile code scenarios where users do not have access to the source programs and require guarantees on compiled code. There are several frameworks to verify low-level code but little work studying the link between reasoning at source and compiled levels, for those properties that can be verified at both levels. The objective of our work is precisely to fill this gap by proposing a mechanism, called *certificate translation*, for bringing the benefits of interactive source code verification to code consumers.

1.2 Proof Carrying Code

In order to transfer evidence from source programs to compiled programs, certificate translation relies on Proof Carrying Code [Necula 1997], a.k.a. PCC, which provides a mean 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. More concretely, certificate translation relies upon a typical PCC architecture, i.e. a formal logic for specifying and verifying policies, a verification condition generator VCGen for producing proof obligations which ensure that the component respects the safety policy, a certificate language to represent proofs, and a proof checker that validates certificates against specifications. While PCC does not preclude generating certificates from interactive verification of source programs, the prominent approach to certificate generation is certifying compilation [Necula and Lee 1998], 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— in order to achieve automatic generation of certificates and, thus, 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, and thus certificate translation must be agnostic about the verification process, and in particular it should applicable to programs that have been annotated and proved interactively with a proof assistant.

1.3 Certificate Translation: informal definition and setting

The primary goal of certificate translation is to transform certificates of source-language programs into certificates of compiled programs. Given a compiler represented by the function $\llbracket \cdot \rrbracket$, a function $\llbracket \cdot \rrbracket_{\mathrm{spec}}$ to transform specifications, and certificate checkers (expressed as a ternary relation "c is a certificate that P adheres to ϕ" and written $c : P \models \phi$), a certificate translator is a function $\llbracket \cdot \rrbracket_{\mathrm{cert}}$ such that for all programs p, policies ϕ, and certificates c,

$$c : p \models \phi \quad \Longrightarrow \llbracket c \rrbracket_{\mathrm{cert}} : \llbracket p \rrbracket \models \llbracket \phi \rrbracket_{\mathrm{spec}}$$

Certificate translators are intrinsically bound to a compilation scheme and to a verification infrastructure, that are captured by the functions $\llbracket\cdot\rrbracket$ and $\llbracket\cdot\rrbracket_{\mathrm{spec}}$, and by the relation $\cdot : \cdot \models \cdot$ respectively.

In this paper, we focus on the problem of certificate translation for an optimizing compiler from a high-level imperative language to an intermediate RTL representation. Compilation proceeds by successive transformations: imperative programs are first translated into RTL programs, then common program optimizations are successively applied to RTL programs in order to yield the target program. For each step, we build an appropriate certificate translator, and combine them to obtain a certificate translator for the complete compilation process.

Our verification infrastructure builds upon verification condition generators (VCGen), which are part of the standard PCC infrastructure and are also used in many interactive verification environments. A VCGen can be seen as an automatic strategy for applying Hoare logic rules; it generates from a program p and a specification ϕ a set of proof obligations $\mathsf{PO}(p, \phi)$ whose validity ensures that the program meets its specification. In this setting, c is a certificate that p satisfies ϕ (denoted $c : p \models \phi$) iff c is a set of (logical) certificates such that for every proof obligation $\psi \in \mathsf{PO}(p, \phi)$, there exists a (logical) certificate $d \in c$ that satisfies $d \models_{\mathrm{po}} \psi$, where $\models_{\mathrm{po}}$ is a binary relation between (logical) certificates and proof obligations.

Of course, certificate translation also depends by definition on the format of certificates, and on the procedure to check that a certificate establishes a property ψ. Nevertheless, we choose not to commit to a particular certificate infrastructure for proof obligations in order to study the existence of certificate translators. Instead, we show the existence of certificate translators for common program optimizations under the assumption that certificates are closed under a few logical rules that include introduction and elimination rules for the $\wedge$ and $\Rightarrow$ connectives and for the $\forall$ quantifier, as well as substitution of equals for equals.

Difficulties. Building a certificate translator for non-optimizing compilation is relatively simple since proof obligations are preserved up to minor differences. Dealing with optimizations is more challenging because:

- *Optimizations that perform arithmetic simplifications* such as constant propagation or common subexpression elimination, do not necessarily preserve verification conditions. Consider the following piece of code to which constant propagation is applied:

$$\begin{array}{ll} r_1 := 1 & r_1 := 1 \\ \{\mathsf{true}\} & \{\mathsf{true}\} \\ r_2 := r_1 & r_2 := 1 \\ \{r_1 = r_2\} & \{r_1 = r_2\} \end{array}$$

  Proof obligations for the assignment instruction to r_2 are $\mathsf{true} \Rightarrow r_1 = r_1$ and $\mathsf{true} \Rightarrow r_1 = 1$ for the original and optimized version respectively. The second proof obligation is unprovable, since this proof obligation is unrelated to the sequence of code containing the assignment $r_1 := 1$.

  The conditions that justify an optimization opportunity must be propagated through every intermediate assertion. Therefore, typical analyzers must be extended into *certifying analyzers*, which justify analyses upon which the optimizations rely by expressing their results in the logic of the PCC architecture, and produce a certificate of the analysis for each program. Then, we define a weaving function that take as argument, in addition to the certificate of the original program, the certificate produced by the certifying analyzer to produce the certificate of the optimized program. The process is presented in Figure 1;

- *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, eliminating nop instructions may lead to delete assertions attached to them, or dead register elimination may eliminate registers that occur in intermediate assertions of the program. Considering the following example, after performing constant propagation

$$\begin{array}{c} x := n; \\ \vdots \\ \{x = n\} \\ y := x; \\ \vdots \\ \{x = y\} \end{array} \longrightarrow \begin{array}{c} x := n; \\ \vdots \\ \{x = n\} \\ y := n; \\ \vdots \\ \{x = y\} \end{array}$$

the variable x is dead. However, we cannot simply remove the first assignment since proof obligations referring to dead registers ($\{x = n\}$ in this case) cannot be proved because all hypotheses about these registers would be lost. 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.

According to the characteristics of their certificate translators, optimizations fall in one of the following categories:

— PPO/IPO (Preservation 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 nonoptimizing compilation and unreachable code elimination;

— IPO (Instantiation of Proof Obligations): 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 inlining;

— 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 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,

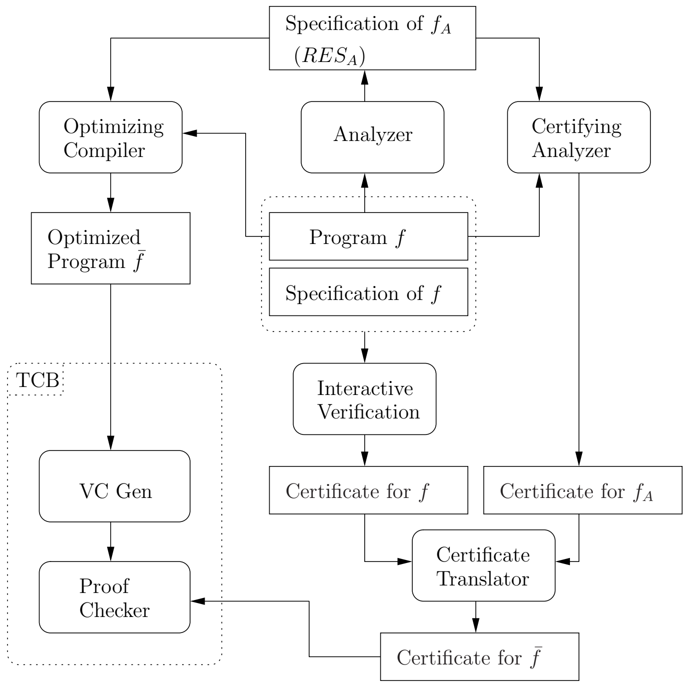

Fig. 1. Overall picture of certificate translation

common subexpression elimination, strength reduction, and other optimizations that rely on arithmetic.

1.4 Contributions

This paper is an extended version of the conference article [Barthe et al. 2006] on certificate translation, considering a broader set of program transformations and providing a detailed fragment of the translation procedure. The main contributions are:

—an introduction of certificate translation as a means to extend significantly the scope of PCC to complex security policies;

—the classification of certificate translation for common optimizations, including constant propagation, loop induction variable strength reduction, dead register elimination, common subexpression elimination, copy propagation, unreachable code elimination, register allocation and function inlining. We present each of the certificate translators for the RTL language.

1.5 Contents

In Section 2 we introduce our proof carrying code setting, including a Register Transfer Language (RTL) and its verification infrastructure. In Section 3, we

$$
\begin{array}{rlll}
\textbf{comparison} & \lhd & ::= & < \mid \leq \mid = \mid \geq \mid > \\
\textbf{expressions} & e & ::= & n \mid r \mid -e \mid e + e \mid e * e \mid \ldots \\
\textbf{assertions} & \phi & ::= & \mathsf{true} \mid e \lhd e \mid \phi \wedge \phi \mid \neg\phi \mid \forall r.\ \phi \mid \ldots \\
\textbf{comparisons} & \mathsf{cmp} & ::= & r\ \lhd\ r \mid r\ \lhd\ n \\
\textbf{operators} & \mathsf{op} & ::= & n \mid r \mid n + r \mid \ldots \\
\textbf{instr. desc.} & \mathsf{ins} & ::= & r_d := \mathsf{op},\ L \\
 & & \mid & r_d := f(\vec{r}),\ L \\
 & & \mid & \mathsf{cmp}\ ?\ L_t : L_f \\
 & & \mid & \mathtt{return}\ r \\
 & & \mid & \mathsf{nop},\ L \\
\textbf{instructions} & I & ::= & (\phi,\ \mathsf{ins}) \mid \mathsf{ins} \\
\textbf{fun. decl} & F & ::= & \{\vec{r};\ \varphi;\ G;\ \psi;\ \lambda;\ \vec{\Lambda}\}
\end{array}
$$

Fig. 2. Syntax of RTL

present certificate translation for nonoptimizing compilers. In Section 4, we describe certificate translation for several standard optimizations. In Section 5, we position our work with respect to other compilation techniques and discuss related work. We conclude in Section 6.

2. PCC 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 Section 2.4), 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, ϕ and ψ range over assertions.

A program p is defined as a function from RTL function identifiers to function declarations. RTL functions return integer values. Every program comes equipped with a special function, namely main, and its declaration. The body of a function is defined as a mapping G from program labels to instructions, and instructions are equipped with explicit successors. Hence, the body of a function constitutes a (closed) pointed graph. A mapping from program points to instructions is preferred, rather than an instruction sequence, to abstract from the details of label updating when modifying the function code. In the sequel, the distinguished label L_{sp} is used as an entry point for every function body. A declaration F for a function f includes its formal parameters $\vec{r}$, a precondition φ, a (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 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 is a certificate that attests the validity of the latter proof obligation and $\vec{\Lambda}$ maps every program point that contains an assertion to the certificate of its associated proof obligation.

Formal parameters are represented as 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 $\mathcal{R}^*$ denote the set $\{r^* \mid r \in \mathcal{R}\}$ and $\vec{r^*}$ denote a sequence of registers in $\mathcal{R}^*$. We also introduce, for specification purposes, 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 respectively the specification of pre and postcondition of the function, and are subject to well-formedness constraints. The precondition of a function f, also referred as $\mathsf{pre}(f)$, is an assertion in which the only registers to occur are the function formal parameters, hereafter denoted $\vec{r_f}$; in other words, the precondition of a function can only talk about the initial values of its parameters. The postcondition of a function f, also referred as $\mathsf{post}(f)$, is an assertion[1] in which the only registers to occur are res and the formal parameters $\vec{r_f}^*$; 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. the application of graph code of f to label L.

An instruction is either an instruction descriptor ins or a pair $(\phi,\ \mathsf{ins})$ consisting of an annotation ϕ and an instruction descriptor ins. 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; and in particular, if L is associated with a `return` instruction, $\mathsf{succ}_f(L) = \emptyset$.

2.2 Operational Semantics

The operational semantics of RTL is standard. In particular, neither proofs nor assertions interfere with the semantics. The semantics is defined in Figure 3 as a big step relation between non terminal states and a terminal state. A non terminal state is defined as a tuple with two elements: the current instruction and a map ρ

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

$$\frac{\langle \mathsf{ins}, \rho\rangle \leadsto_f n}{\langle (\phi,\ \mathsf{ins}), \rho\rangle \leadsto_f n}$$

$$\frac{\langle f[L], [\rho \mid r_d \mapsto [\![\mathsf{op}]\!]_\rho]\rangle \leadsto_f n}{\langle r_d := \mathsf{op},\ L, \rho\rangle \leadsto_f n}$$

$$\frac{\langle f[L_t], \rho\rangle \leadsto_f n}{\langle \mathsf{cmp}\ ?\ L_t : L_f, \rho\rangle \leadsto_f n}\ \text{if } [\![\langle \mathsf{cmp}\rangle]\!]_\rho$$

$$\frac{\langle f[L_f], \rho\rangle \leadsto_f n}{\langle \mathsf{cmp}\ ?\ L_t : L_f, \rho\rangle \leadsto_f n}\ \text{if } \neg[\![\langle \mathsf{cmp}\rangle]\!]_\rho$$

$$\frac{\langle g[L_{\mathsf{sp}}], [\vec{r}_g \mapsto \rho\vec{r}]\rangle \leadsto_g m \quad \langle f[L'], [\rho \mid ret \mapsto m]\rangle \leadsto_f n}{\langle ret := g(\vec{r}),\ L', \rho\rangle \leadsto_f n}$$

$$\frac{}{\langle \mathtt{return}\ r, \rho\rangle \leadsto_f \rho r}$$

Fig. 3. Operational Semantics

from local register to values. The expression $[\rho \mid x \mapsto n]$ stands for the function ρ' s.t. $\rho' y = n$ if $x = y$ and $\rho' y = \rho y$ otherwise.

Let $[\![]\!]$ be a standard interpretation function that takes an assertion and a map from registers to values and returns a logical proposition. For clarity, the interpretation $[\![.]\!]_\rho^{\rho^*}$ refers to two parameters ρ and ρ^* with disjoint domains $\mathcal{R}$ and $\mathcal{R}^*$ respectively. When it is clear from the context that an assertion ϕ does not contain registers in $\mathcal{R}^*$, e.g. when ϕ is a precondition, we may simply write $[\![\phi]\!]_\rho$ instead of $[\![\phi]\!]_\rho^{\rho^*}$.

We say that an assertion is valid, if for every assignments ρ and ρ^*, $[\![\phi]\!]_\rho^{\rho^*}$ is a valid logical proposition. Similarly, we define an interpretation function $[\![.]\!]$ for expressions, that takes the assignments ρ and ρ^* respectively for registers in $\mathcal{R}$ and $\mathcal{R}^*$. When an expression e does not contain registers in $\mathcal{R}^*$, e.g. when it is an expression of the programming language, we may simply write $[\![e]\!]_\rho$ instead of $[\![e]\!]_\rho^{\rho^*}$.

2.3 Verification Condition Generator

Verification condition generators (VCGens) produce the proof obligations that must be discharged in order to guarantee that the program meets its specification. The predicate transformer wp is a partial function that computes, from a sufficiently annotated program, a fully annotated program in which all labels of the program have an explicit precondition attached to them. To ensure the computability of the wp function, its domain is restricted to *well-annotated* programs. This domain can be characterized by an inductive and decidable definition and does not impose any specific structure on programs.

Definition 2.1.

— *The graph of a function f is closed if for every node all its successors are in the graph:*

$$\mathsf{closed}(f) = \forall L \in G_f, L' \in \mathsf{succ}_f(L) \Rightarrow L' \in G_f$$

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

Fig. 4. Verification condition generator

— *A label L' is reachable from a label L in f, if $L = L'$, or if it is the successor of a label reachable from L:*

$$
\begin{array}{l}
L \in \mathsf{reachable}_{f,L} \\
L' \in \mathsf{reachable}_{f,L} \Rightarrow \forall L'' \in \mathsf{succ}_f(L'), L'' \in \mathsf{reachable}_{f,L}
\end{array}
$$

— *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` *instruction (in that case the annotation is the post condition), or if all its immediate successors reach annotated labels. More precisely,* $\mathsf{reachAnnot}_f$ *is defined as the smallest set that satisfies the following conditions:*

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

— *A function f is well annotated if it is closed and every reachable point from the 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 compute an assertion for every label. For each program point, its associated assertion represents the precondition that a state must satisfy to guarantee that the function reaches only final states satisfying the postcondition.

A fully annotated function is computed from a partial annotated function f using the wp_f transformer. The computation 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 $\mathsf{wp}_f(L)$ proceeds by case analysis: if L points to an instruction that carries an assertion ϕ, then $\mathsf{wp}_f(L)$ is set to ϕ; otherwise, $\mathsf{wp}_f(L)$ is computed by the function $\mathsf{wp}^{\mathtt{id}}_f$.

The formal definitions of wp_f and $\mathsf{wp}^{\mathtt{id}}_f$ are given in Figure 4, where the expression $e[{}^{e'}/_{r}]$ stands for the substitution in the expression e of all occurrences of register r by e'. The definition of $\mathsf{wp}^{\mathtt{id}}_f$ 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, $\mathsf{wp}^{\mathtt{id}}_f(r_d := g(\vec{r}),\ L)$ is defined as a conjunction of the precondition of g, where formal parameters are replaced by

$\mathsf{intro}_{\mathsf{true}}$	:	$\mathcal{C}(\Gamma \vdash \mathsf{true})$	
$\mathsf{axiom}(A)$	:	$\mathcal{C}(\Gamma \vdash A)$	if $A \in \Gamma$
ring	:	$\mathcal{C}(\Gamma \vdash n_1 = n_2)$	if $n_1 = n_2$ is a ring equality
$\mathsf{intro}_{\wedge}$	:	$\mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma \vdash B) \to \mathcal{C}(\Gamma \vdash A \wedge B)$	
$\mathsf{elim}^{\mathsf{l}}_{\wedge}$	:	$\mathcal{C}(\Gamma \vdash A \wedge B) \to \mathcal{C}(\Gamma \vdash A)$	
$\mathsf{elim}^{\mathsf{r}}_{\wedge}$	:	$\mathcal{C}(\Gamma \vdash A \wedge B) \to \mathcal{C}(\Gamma \vdash B)$	
$\mathsf{intro}_{\Rightarrow}$	:	$\mathcal{C}(\Gamma; A \vdash B) \to \mathcal{C}(\Gamma \vdash A \Rightarrow B)$	
$\mathsf{elim}_{\Rightarrow}$	:	$\mathcal{C}(\Gamma \vdash A \Rightarrow B) \to \mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma \vdash B)$	
$\mathsf{elim}_{=}$	:	$\mathcal{C}(\Gamma \vdash e_1 = e_2) \to \mathcal{C}(\Gamma \vdash A[^{e_1}/_{r}]) \to \mathcal{C}(\Gamma \vdash A[^{e_2}/_{r}])$	
$\mathsf{subst}\ r\ e$	:	$\mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma[^{e}/_{r}] \vdash A[^{e}/_{r}])$	
weak_{Δ}	:	$\mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma; \Delta \vdash A)$	
$\mathsf{intro}_{\forall}$	:	$\mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma \vdash \forall r.A)$	if r is not in Γ
$\mathsf{elim}_{\forall}$	:	$\mathcal{C}(\Gamma \vdash \forall r.A) \to \mathcal{C}(\Gamma \vdash A)$	

Fig. 5. Proof Algebra

actual parameters, and of the assertion $\forall \mathsf{res}.\ \mathsf{post}(g)[^{\vec{r}}/_{\vec{r}^*_g}] \Rightarrow \mathsf{wp}_f(L)[^{\mathsf{res}}/_{r_d}]$. The second conjunct permits that information in $\mathsf{wp}_f(L)$ about registers different from r_d be propagated to other preconditions. In the remainder of the paper, we shall abuse notation and write $\mathsf{wp}^{\mathtt{id}}_f(L)$ instead of $\mathsf{wp}^{\mathtt{id}}_f(\mathsf{ins})$ if $f[L] = \mathsf{ins}$.

2.4 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. A proof algebra is given by a set-valued function $\mathcal{C}$ over judgments, and by a set of operations, all implicitly quantified in the usual way. The operations are standard (given in Figure 5), 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.

In order to remain at an abstract level, we do not provide an algorithm for checking certificates. Instead, we take $\mathcal{C}(\Gamma \vdash \phi)$ 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. Furthermore, we require the certificate infrastructure to be sound, i.e., if $\mathcal{C}(\Gamma \vdash \phi) \neq \emptyset$ then for all maps ρ, ρ^*, if for every $\psi \in \Gamma$, $[\![\psi]\!]^{\rho^*}_{\rho}$ is valid, then $[\![\phi]\!]^{\rho^*}_{\rho}$ is valid.

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.

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

—*A program is certified if all its functions are.*

2.5 Soundness of PCC Infrastructure

The verification condition generator is sound, in the sense that if a certified program p is called with registers set to values that verify the precondition of the function main, and it terminates normally, then the final state will verify the postcondition of main.

When considering mutually recursive functions, special care must be taken to ensure soundness of the VCGen. In this case it is not hard to achieve since we are only interested in verifying finite executions.

LEMMA 2.1. *Let p be a certified program. Then, for every function f with declaration $\{\vec{r};\ \varphi;\ G;\ \psi;\ \lambda;\ \vec{\Lambda}\}$, any initial mapping ρ^* with domain $\{r_1^*, \dots, r_k^*\}$, any label L in the domain of G_f and any state ρ, if $[\![\mathsf{wp}_f(L)]\!]^{\rho^*}_{\rho}$ and $\langle f[L], \rho\rangle \rightsquigarrow_f n$ then $[\![\psi]\!]^{\rho^*}_{[\mathsf{res}\mapsto n]}$.*

PROOF. *Since wp and $\mathsf{wp}^{\mathsf{ins}}$ are defined each one in terms of the other, we prove the goal of the lemma above simultaneously with a similar goal but under the hypothesis $[\![\mathsf{wp}^{\mathsf{ins}}_f(L)]\!]^{\rho^*}_{\rho}$. The proof follows by rule induction on the derivation of $\langle f[L], \rho\rangle \rightsquigarrow_f n$. For simplicity, we rely in the following standard results (where $FV(\varphi)$ stands for the set of unbound variables in φ):*

i) **(Coincidence Lemma)**. *For all states $\rho_1, \rho_2, \rho_1^*, \rho_2^*$ and assertion φ, if for all x in $FV(\varphi) \cap \mathcal{R}$ and y in $FV(\varphi) \cap \mathcal{R}^*$ we have $\rho_1 x = \rho_2 x$ and $\rho_1^* y = \rho_2^* y$ then $[\![\varphi]\!]^{\rho_1^*}_{\rho_1} = [\![\varphi]\!]^{\rho_2^*}_{\rho_2}$.*

ii) **(Substitution Lemma)**. *For all x, c, φ and ρ, ρ^*, $[\![\phi[{}^{e}/_{x}]]\!]^{\rho^*}_{\rho} = [\![\phi]\!]^{\rho^*}_{[\rho|x\mapsto[\![e]\!]\rho^*\rho]}$ and $[\![\phi[{}^{e}/_{x^*}]]\!]^{\rho^*}_{\rho} = [\![\phi]\!]^{[\rho^*|x^*\mapsto[\![e]\!]\rho^*\rho]}_{\rho}$.*

—*Consider the case s.t. the last rule applied is*

$$\frac{\langle \mathsf{ins}, \rho\rangle \rightsquigarrow_f n}{\langle (\phi,\ \mathsf{ins}), \rho\rangle \rightsquigarrow_f n}$$

then $\mathsf{wp}_f(L) = \phi$ and since f is certified, and the certificate infrastructure is sound, we have that $[\![\phi \Rightarrow \mathsf{wp}^{\mathsf{ins}}_f(L)]\!]^{\rho^*}_{\rho}$ is valid. Hence, from the hypothesis $[\![\mathsf{wp}_f(L)]\!]^{\rho^*}_{\rho}$ and definition of $[\![.]\!]$, we have that $[\![\mathsf{wp}^{\mathsf{ins}}_f(L)]\!]^{\rho^*}_{\rho}$ is valid. By I.H. and the latter condition we get $[\![\psi_f]\!]^{\rho^*}_{[\mathsf{res}\mapsto n]}$. Notice that this is the only case where we need to distinguish the hypothesis $[\![\mathsf{wp}^{\mathsf{ins}}(L)]\!]^{\rho^*}_{\rho}$ from $[\![\mathsf{wp}(L)]\!]^{\rho^*}_{\rho}$; in any other case $\mathsf{wp}(L) = \mathsf{wp}^{\mathsf{ins}}(L)$.

—*Assume the last rule applied is*

$$\frac{\langle f[L'], [\rho \mid r_d \mapsto [\![\mathsf{op}]\!]^{\rho^*}_{\rho}]\rangle \rightsquigarrow_f n}{\langle r_d := \mathsf{op},\ L', \rho\rangle \rightsquigarrow_f n}\ .$$

By hypothesis and definition of wp_f, *we have* $\llbracket \mathsf{wp}_f(L')[{}^{\mathsf{op}}\!/_{r_d}] \rrbracket_{\rho}^{\rho^*}$. *Equivalently, by substitution lemma* $\llbracket \mathsf{wp}_f(L') \rrbracket_{[\rho | r_d \mapsto \llbracket \mathsf{op} \rrbracket \rho^* \rho]}^{\rho^*}$

—*Assume the last rule applied involves a function call, i.e. there is a function* g *s.t.* $f[L] = ret := g(\vec{r}),\ L'$ *and the last rule applied is*

$$\frac{\langle g[L_{\mathsf{sp}}], [\vec{r}_g \mapsto \rho\vec{r}] \rangle \rightsquigarrow_g m \quad \langle f[L'], [\rho \mid ret \mapsto m] \rangle \rightsquigarrow_f n}{\langle ret := g(\vec{r}),\ L', \rho \rangle \rightsquigarrow_f n}$$

for some value m. *Let* φ_g *and* ψ_g *stand for the preconditions and postcondition of* g *respectively. From* $\llbracket \mathsf{wp}_f(L) \rrbracket_{\rho}^{\rho^*}$ *and definition of* $\llbracket . \rrbracket$, *we have* $\llbracket \varphi_g[{}^{\vec{r}}\!/_{\vec{r}_g}] \rrbracket_{\rho}^{\rho^*}$. *By coincidence lemma we have then* $\llbracket \varphi_g[{}^{\vec{r}}\!/_{\vec{r}_g}] \rrbracket_{[r \mapsto \rho r]}^{[r_g^* \mapsto \rho r]}$, *and by substitution lemma* $\llbracket \varphi_g \rrbracket_{[r_g \mapsto \rho r]}^{[r_g^* \mapsto \rho r]}$. *Since* g *is certified, we have a proof for* $\varphi_g \Rightarrow \mathsf{wp}_g(L_{sp})[{}^{\vec{r}_g}\!/_{\vec{r}_g^*}]$ *and therefore* $\llbracket \mathsf{wp}_g(L_{sp})[{}^{\vec{r}_g}\!/_{\vec{r}_g^*}] \rrbracket_{[r_g \mapsto \rho r]}^{[r_g^* \mapsto \rho r]}$. *Again by substitution lemma,* $\llbracket \mathsf{wp}_g(L_{sp}) \rrbracket_{[r_g \mapsto \rho r]}^{[r_g^* \mapsto \rho r]}$. *Therefore, by application of I.H., we know that* $\llbracket \psi_g \rrbracket_{[\mathsf{res} \mapsto m]}^{[r_g^* \mapsto \rho r]}$, *and then by substitution lemma* $\llbracket \psi_g[{}^{m}\!/_{\mathsf{res}}] \rrbracket_{\rho}^{[r_g^* \mapsto \rho r]}$. *From the hypothesis* $\llbracket \mathsf{wp}_f(L) \rrbracket_{\rho}^{\rho^*}$ *and definition of* $\llbracket . \rrbracket$, *we have that* $\llbracket \psi_g[{}^{\vec{r}}\!/_{\vec{r}_g^*}] \rrbracket_{[\rho | \mathsf{res} \mapsto m]}^{\rho^*} \Rightarrow \llbracket \mathsf{wp}_f(L')[{}^{\mathsf{res}}\!/_{ret}] \rrbracket \rho^*[\rho \mid \mathsf{res} \mapsto m]$. *Equivalently by substitution lemma,* $\llbracket \psi_g[{}^{m}\!/_{\mathsf{res}}] \rrbracket_{\rho}^{[r_g^* \mapsto \rho r]} \Rightarrow \llbracket \mathsf{wp}_f(L') \rrbracket_{[\rho | ret \mapsto m]}^{\rho^*}$, *and hence* $\llbracket \mathsf{wp}_f(L') \rrbracket_{[\rho | ret \mapsto m]}^{\rho^*}$. *From the latter condition and I.H., we get the desired result.*

□

As a corollary, we obtain the following theorem:

Theorem Soundness of VCGen. *Suppose that*

a) P *is a certified program containing a function* main, *with precondition* Φ *and postcondition* Ψ,

b) ρ *is such that* $\llbracket \Phi \rrbracket_{\rho}$, *and*

c) $\langle \mathsf{main}[L_{sp}], \rho \rangle \rightsquigarrow n$,

then the interpretation $\llbracket \Psi \rrbracket_{[\mathsf{res} \mapsto n]}^{[\vec{r^*} \mapsto \rho\vec{r}]}$ *is valid.*

3. PRESERVATION OF PROOF OBLIGATIONS

The purpose of this section is to establish preservation of proof obligations for a nonoptimizing compiler from an imperative language with procedures to RTL. This result is inspired from earlier work by Barthe, Rezk and Saabas [Barthe et al. 2005].

In this section, we define a simple and structured high-level language, a standard verification condition generator for this language, and a nonoptimizing compiler to the RTL language defined before. Then, we show that given the same program specification, proof obligations for the source program and for its compiled RTL version coincide.

3.1 Source Language

A program p in the source language is defined as a function from function identifiers to function declarations. We assume that every program comes equipped with a

$$\begin{array}{rcl}
\lhd & ::= & < \mid \leq \mid = \mid \geq \mid > \mid \ \wedge \mid \ \vee \\
e & ::= & x \mid n \mid -e \mid e\ +\ e \mid e\ -\ e \mid e\ *\ e \\
c & ::= & \mathtt{skip} \mid x := e \mid c; c \mid \mathtt{while}\ \{\phi\}\ e \lhd e\ \mathtt{do}\ c \mid \\
 & & \mathtt{if}\ e \lhd e\ \mathtt{then}\ c\ \mathtt{else}\ c \mid y := \mathtt{call}\ f(\vec{x}) \mid \\
 & & \mathtt{return}\ e
\end{array}$$

Fig. 6. Syntax of source language

$$\begin{array}{lcl}
\mathsf{wp}(\mathtt{skip}, \psi) & = & \psi \\
\mathsf{wp}(x := e, \psi) & = & \psi[e/x] \\
\mathsf{wp}(c_1; c_2, \psi) & = & \mathsf{wp}(c_1, \mathsf{wp}(c_2, \psi)) \\
\mathsf{wp}(\mathtt{while}\ \{\phi\}\ e_1 \lhd e_2\ \mathtt{do}\ c_1, \psi) & = & \phi \\
\mathsf{wp}(\mathtt{if}\ e_1 \lhd e_2\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \psi) & = & \langle e_1 \lhd e_2 \rangle \Rightarrow \mathsf{wp}(c_1, \psi) \wedge \neg\langle e_1 \lhd e_2 \rangle \Rightarrow \mathsf{wp}(c_2, \psi) \\
\mathsf{wp}(y := \mathtt{call}\ g(\vec{x}), \psi) & = & \mathsf{pre}(g)[{}^{\vec{x}}/_{\vec{x}_g}] \wedge \\
 & & \forall \mathsf{res}. (\mathsf{post}(g)[{}^{\vec{x}}/_{\vec{x}_g^*}] \Rightarrow \psi[{}^{\mathsf{res}}/_{y}]) \\
\mathsf{wp}(\mathtt{return}\ e, \psi) & = & \psi[{}^{e}/_{\mathsf{res}}]
\end{array}$$

Fig. 7. wp for the source language

$$\begin{array}{lcl}
\mathsf{PO}(\mathtt{skip}, \psi) & = & \emptyset \\
\mathsf{PO}(x := e, \psi) & = & \emptyset \\
\mathsf{PO}(c_1; c_2, \psi) & = & \mathsf{PO}(c_2, \psi) \cup \mathsf{PO}(c_1, \mathsf{wp}(c_2, \psi)) \\
\mathsf{PO}(\mathtt{while}\ \{\phi\}\ e_1 \lhd e_2\ \mathtt{do}\ c_1, \psi) & = & \\
\quad \mathsf{PO}(c_1, \phi) \cup \{\phi \Rightarrow (e_1 \lhd e_2 \Rightarrow \mathsf{wp}(c_1, \phi)) \wedge (\neg(e_1 \lhd e_2) \Rightarrow \psi)\} & & \\
\mathsf{PO}(\mathtt{if}\ e_1 \lhd e_2\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \psi) & = & \mathsf{PO}(c_1, \psi) \cup \mathsf{PO}(c_2, \psi) \\
\mathsf{PO}(y := \mathtt{call}\ g(\vec{x}), \psi) & = & \emptyset \\
\mathsf{PO}(\mathtt{return}\ e, \psi) & = & \emptyset
\end{array}$$

Fig. 8. Proof obligations for the source language

special function identifier, namely main, and its declaration. The declaration of a function f in the source language has the form: $\{\vec{x};\ \varphi;\ c;\ \psi;\ \lambda;\ \vec{\Lambda}\}$, where c is a command whose syntax is shown in Figure 6. Every function returns integer values.

As in RTL programs, a function declaration includes its formal parameters $\vec{x}$, a precondition φ, a postcondition ψ, a certificate λ, and a *set* of certificates $\vec{\Lambda}$. The source language features the same annotation language as RTL. However, the only command that has an annotation is the `while` command. A program is well-annotated if all its `while` commands hold annotations. The definition of the VCGen is made in terms of the function wp, which is overloaded to denote as well a predicate transformer for the source code, and is given in Figure 7. Certificates for source level and RTL programs are represented by the same proof algebra.

Definition 3.1.

—*A function f with declaration $\{\vec{x};\ \varphi;\ c;\ \psi;\ \lambda;\ \vec{\Lambda}\}$ is certified if:*

—λ *is a proof of* $\vdash \varphi \Rightarrow \mathsf{wp}(c, \psi)[\vec{x}/_{\vec{x}^*}]$

—$\vec{\Lambda}$ *contains a proof of* $\vdash \varphi$ *for every proof obligation* φ *in* $\mathsf{PO}(c, \psi)$*, where* PO *is defined in Figure 8.*

—*A program is certified if all its functions are.*

The semantics of the source language is standard and, thus, omitted.

3.2 Compilation

The compilation function to RTL is standard, except for the while command. There are two compilation schemes, sketched in Figure 9. The first one, $L_H : \llbracket e \rrbracket_{r_d, L_E}$, is defined for the compilation of source expressions. The resulting code (graph), that starts at label L_H, evaluates the expression e, stores its result in the register r_d and jumps to the continuation label L_E. When the expression e is a variable or constant the compilation scheme simply accesses the corresponding value and stores the result in r_d. For binary operators of the form $e_1 \diamond e_2$, the expressions e_2 and e_1 are first evaluated in order, and the results are respectively stored in fresh registers r_2 and r_1. Finally, the code execute the last instruction which perform the binary operation, storing the result in r_d, and jumps to the continuation label L_E. It is important to notice that the compilation scheme is global (and imperative); the compiler maintains a set of already used labels and registers, and each time the compiler needs a fresh register (resp. a fresh label) the new register (resp. label) is added to the set and will not be used later. An important point is that in the intermediate RTL representation we consider an infinite number of pseudo-registers, rather than machine registers. In consequence, variables of the source language are directly mapped to a pseudo register of RTL (sharing the same name.)

The second compilation scheme $L_H : \llbracket c \rrbracket_{L_E}$ is defined for commands, L_H is the entry point and L_E the exit point (i.e. the successor label of the last instruction.) The skip command is compiled to the `nop` instruction. The compilation of an assignment simply use the compilation of the right member where the resulting value is stored in the left member (i.e. the variable). The compilation of the binary relation of a conditional statement follows the same scheme than for binary expressions, the two expressions are evaluated and stored in two fresh register r_1 and r_2, then a conditional RTL instruction evaluates the test and jumps to the label L_T, corresponding to the beginning of the true branch, if the evaluation is positive or to L_F otherwise. For the compilation of while loops, the unusual part is the insertion of an assertion just before the beginning of the evaluation of the test.

3.3 Preservation of Proof Obligations

The wp of a source code function is syntactically equivalent to the wp of its compilation, provided the variables of the source language and the registers of RTL are equivalent. For notational convenience, in the following proofs we let the expression $f[L, L']$ stand for the subgraph of nodes reachable from label L without traversing (and not including) the node at label L'.

LEMMA 3.1. *Let* $f[L, L']$ *be a subgraph code of the RTL function* f *given by compilation* $L : \llbracket c \rrbracket_{L'}$ *of a command* c*. Then,* $\mathsf{wp}(c, \psi) = \mathsf{wp}_f(L)$*, where* $\psi = \mathsf{wp}_f(L')$.

Proof. *The proof proceeds by structural induction on the command c. For simplicity, we assume the following result about the compilation of expressions:*

$$\text{if } f[l,l'] = L : \llbracket e \rrbracket_{r,L'} \text{ then } \mathsf{wp}_f(l) = \mathsf{wp}_f(l')[{}^{e}\!/\!_{r}] \tag{1}$$

—*case $c = \texttt{while}\ \{\phi\}\ e_1 \lhd e_2\ \texttt{do}\ c$. In this case $\mathsf{wp}(c,\psi)$ is equal to ϕ, as well as $\mathsf{wp}_f(L)$ by definition of $\llbracket . \rrbracket$.*

—*case $c = x := e$. We have that $\mathsf{wp}(c,\psi)$ is equal to $\psi[{}^{e}\!/\!_{x}]$, and thus, to $\mathsf{wp}_f(L')[{}^{e}\!/\!_{x}]$ by hypothesis. From definition of $\llbracket . \rrbracket$, $f[L,L'] = L : \llbracket e \rrbracket_{r,L'}$, and property (1), we have that $\mathsf{wp}(c,\psi)$ is equal to $\mathsf{wp}_f(L)$.*

—*case $c = c_1; c_2$. By definition of wp, $\mathsf{wp}(c,\psi) = \mathsf{wp}(c_1, \mathsf{wp}(c_2,\psi))$. By definition of $\llbracket c \rrbracket_{L,L'}$ we have $f[L,L''] = L : \llbracket c_1 \rrbracket_{L''}$ and $f[L'',L'] = L'' : \llbracket c_2 \rrbracket_{L'}$, then, by I.H. $\mathsf{wp}(c_2,\psi) = \mathsf{wp}_f(L'')$. Hence, $\mathsf{wp}(c,\psi) = \mathsf{wp}(c_1,\psi')$ where $\psi' = \mathsf{wp}_f(L'')$ and, since again by I.H. we have $\mathsf{wp}(c_1,\psi') = \mathsf{wp}_f(L)$, we get $\mathsf{wp}(c,\psi) = \mathsf{wp}_f(L)$.*

□

Hence, one can prove that proof obligations and certificates are preserved[2] along nonoptimizing compilation.

Lemma 3.2. *Let f with declaration $\{\vec{x};\ \varphi;\ c;\ \psi;\ \lambda;\ \vec{\Lambda}\}$ be a source certified function. Then $\overline{f}$ declared as $\{\vec{x};\ \varphi;\ L_{\mathsf{sp}} : \llbracket c \rrbracket_L;\ \psi;\ \lambda;\ \vec{\Lambda}\}$ is an RTL certified function.*

Proof. *The proof consists on verifying that f and $\overline{f}$ contain exactly the same proof obligations. To this end, consider a subprogram c s.t. $\overline{f}[l,l'] = l : \llbracket c \rrbracket_{l'}$ and $\psi = \mathsf{wp}_{\overline{f}}(l')$. Then, we show by structural induction on c, that the proof obligations induced by assertions in $\overline{f}[l,l']$ correspond to the proof obligations in $\mathsf{PO}(c,\psi)$. We only consider the case $c = \texttt{while}\ \{\phi\}\ e_1 \lhd e_2\ \texttt{do}\ c'$. The proof obligations in c w.r.t. ψ are the proof obligations in $\mathsf{PO}(c',\phi)$ plus*

$$\phi \Rightarrow (e_1 \lhd e_2 \Rightarrow \mathsf{wp}(c',\phi)) \wedge (\neg(e_1 \lhd e_2) \Rightarrow \psi) \ .$$

By definition of $\llbracket . \rrbracket$ we have

$$\begin{aligned} f[l,l'] = \ & l \ : \ (\phi,\ \texttt{nop},\ l_b) \\ & l_b \ : \ \llbracket e_2 \rrbracket_{r_2,l'_b} \\ & l'_b \ : \ \llbracket e_1 \rrbracket_{r_1,l''} \\ & l'' \ : \ r_1 \lhd r_2\ ?\ l_T : l' \\ & l_T \ : \ \llbracket c' \rrbracket_l \end{aligned}$$

Annotations in $f[l,l']$ are ϕ plus the annotations in c'. By I.H., proof obligations in $f[l_T,l']$ are exactly the proof obligations in $\mathsf{PO}(c',\phi)$. The annotation ϕ in l induces the proof obligation $\phi \Rightarrow \mathsf{wp}^{\mathsf{ins}}_f(L_b)$, that after unfolding of wp_f and $\mathsf{wp}^{\mathsf{ins}}_f$ can be shown equal to

$$\phi \Rightarrow (e_1 \lhd e_2 \Rightarrow \mathsf{wp}_f(l_T)) \wedge (\neg(e_1 \lhd e_2) \Rightarrow \mathsf{wp}_f(l'))$$

[2]Strictly speaking, the first $\vec{\Lambda}$ in the source program is a set, whereas the second $\vec{\Lambda}$ in the compiled program is a map, but it is immediate to turn one into the other.

$$\begin{array}{lcl}
L_H : \llbracket x \rrbracket_{r_d,L_E} & = & L_H \; : \; r_d := x,\; L_E \\
L_H : \llbracket n \rrbracket_{r_d,L_E} & = & L_H \; : \; r_d := n,\; L_E \\
L_H : \llbracket -e \rrbracket_{r_d,L_E} & = & L_H : \llbracket e \rrbracket_{r,L} \\
 & & L \; : \; r_d := -r,\; L_E \\
\multicolumn{3}{c}{L \text{ a fresh label, } r \text{ a fresh register}} \\
L_H : \llbracket e_1 \diamond e_2 \rrbracket_{r_d,L_E} & = & L_H : \llbracket e_2 \rrbracket_{r_2,L_2} \\
 & & L_2 : \llbracket e_1 \rrbracket_{r_1,L_1} \\
 & & L_1 \; : \; r_d := r_1 \diamond r_2,\; L_E \\
\multicolumn{3}{c}{L_1\; L_2 \text{ fresh labels, } r_1\; r_2 \text{ fresh registers}} \\
\multicolumn{3}{c}{\text{with } \diamond \; = \; + \mid * \mid -}
\end{array}$$

Compilation of source expressions

$$\begin{array}{lcl}
L_H : \llbracket \mathsf{skip} \rrbracket_{L_E} & = & L_H \; : \; \texttt{nop},\; L_E \\
L_H : \llbracket x := e \rrbracket_{L_E} & = & L_H : \llbracket e \rrbracket_{x,L_E} \\
L_H : \llbracket c_1 ; c_2 \rrbracket_{L_E} & = & L_H : \llbracket c_1 \rrbracket_{L_1} \\
 & & L_1 : \llbracket c_2 \rrbracket_{L_E} \\
\multicolumn{3}{c}{L_1 \text{ a fresh label}} \\
L_H : \llbracket \mathsf{if}\; e_1 \triangleleft e_2 \;\mathsf{then}\; c_1 \;\mathsf{else}\; c_2 \rrbracket_{L_E} & = & L_H : \llbracket e_2 \rrbracket_{r_2,L_2} \\
 & & L_2 : \llbracket e_1 \rrbracket_{r_1,L_1} \\
 & & L_1 \; : \; r_1 \triangleleft r_2 \;?\; L_T : L_F \\
 & & L_T : \llbracket c_1 \rrbracket_{L_E} \\
 & & L_F : \llbracket c_2 \rrbracket_{L_E} \\
\multicolumn{3}{c}{L_1\; L_2\; L_T\; L_F \text{ fresh labels, } r_1\; r_2 \text{ fresh registers}} \\
L_H : \llbracket y := \mathsf{call}\; f(\vec{x}) \rrbracket_{L_E} & = & L_H \; : \; y := f(\vec{x}),\; L_E \\
L_H : \llbracket \mathsf{while}\; \{\phi\}\; e_1 \;\triangleleft\; e_2 \;\mathsf{do}\; c \rrbracket_{L_E} & = & L_H \; : \; (\phi,\; \texttt{nop},\; L_B) \\
 & & L_B : \llbracket e_2 \rrbracket_{r_2,L'_B} \\
 & & L'_B : \llbracket e_1 \rrbracket_{r_1,L} \\
 & & L \; : \; r_1 \triangleleft r_2 \;?\; L_T : L_E \\
 & & L_T : \llbracket c \rrbracket_{L_H} \\
\multicolumn{3}{c}{L_B\; L'_B\; L \text{ fresh labels, } r_1\; r_2 \text{ fresh registers}} \\
L_H : \llbracket \texttt{return}\; e \rrbracket_{L_E} & = & L_H : \llbracket e \rrbracket_{r,L} \\
 & & L \; : \; \texttt{return}\; r \\
\multicolumn{3}{c}{L \text{ a fresh label, } r \text{ a fresh register}}
\end{array}$$

Compilation of source commands

Fig. 9. Compilation of source language to RTL

which, by Lemma 3.1, is equal to

$$\phi \Rightarrow (e_1 \triangleleft e_2 \Rightarrow \mathsf{wp}(c', \phi)) \wedge (\neg(e_1 \triangleleft e_2) \Rightarrow \psi) \;\; .$$

□

We have seen in this section that the first phase of a compiler, that translates a high-level structured program into an RTL representation, preserves verification conditions if no optimization is applied. In the next section we extend this simple compiler with standard optimization phases and for each of them we propose a transformation of the certificate.

4. CERTIFICATE TRANSLATION FOR COMMON OPTIMIZATIONS

This section provides instances of certificate translation 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.

4.1 Overview

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. Consider for example the following annotated piece of code:

$$\begin{array}{rl} & \{\mathsf{true}\} \\ & r_1 := n \\ L: & \{r_1 \geq n\},\ L' \\ L': & r_2 := r_1 \\ & \{r_1 = r_2\} \end{array}$$

An analysis may detect, ignoring annotations, that the register r_1 always stores the value n at program points L and L'. Later, a transformation phase optimizes the code replacing the assignment $r_2 := r_1$ by the more efficient $r_2 := n$.

$$\begin{array}{rl} & \{\mathsf{true}\} \\ & r_1 := n \\ L: & \{r_1 \geq n\},\ L' \\ L': & r_2 := n \\ & \{r_1 = r_2\} \end{array}$$

According to Definition 2.2, a certificate for an optimized function $\bar{f}$ must include a proof that the precondition of $\bar{f}$ implies the precondition of its first instruction, and a proof, for each label L of $\bar{f}$, that the assertion at L implies the precondition of instruction L. In the example, the proof obligations corresponding to the original fragment of code are $\mathsf{true} \Rightarrow n \geq n$ and $r_1 \geq n \Rightarrow r_1 = r_1$. After the transformation we have that proof obligations are $\mathsf{true} \Rightarrow n \geq n$ and $r_1 \geq n \Rightarrow n = r_1$. Not only does one of the proof obligations not coincide with the original one (and hence the original certificate cannot be reused), but it also becomes unprovable.

The above example illustrates that the validity of annotations is not necessarily preserved by program transformations. In order to maintain their validity, the original annotations must be strengthened by assertions that capture in a logical form the results of the analysis that underlies the optimization. Intuitively, the need to strengthen assertions stems from the fact that semantic preservation of program transformations must eventually be justified by the conditions returned by the analysis.

Therefore, optimized programs are defined by augmenting annotations with the information returned by the analysis, expressed as an assertion and denoted $\mathrm{RES}_{\mathcal{A}}(L)$ below.

Definition 4.1. *The optimized graph code of a function f is defined as follows:*

$$G_{\bar{f}}(L) = \begin{cases} (\phi \wedge \mathrm{RES}_{\mathcal{A}}(L),\ \llbracket \mathsf{ins} \rrbracket) & \textit{if } G_f(L) = (\phi,\ \mathsf{ins}) \\ \llbracket \mathsf{ins} \rrbracket & \textit{if } G_f(L) = \mathsf{ins} \end{cases}$$

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

(Note that the above definition is restricted for simplicity to optimizations that do not modify the graph topology.)

Augmenting an assertion ϕ into $\phi \wedge \mathrm{RES}_{\mathcal{A}}(L)$ has two immediate effects. On the negative side, the inserted condition $\mathrm{RES}_{\mathcal{A}}(L)$ requires certificates for the new proof obligations involving the results of the analysis. To solve this issue, an automatic procedure for certification of the analysis, namely a certifying analyzer, is applied as a first step, producing the certified program

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

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 labels L in f.

On the positive side, strengthening the antecedent enables us to build a proof for the transformed proof obligations. We illustrate the benefits of strengthening assertions on the example above, and then elaborate on the transformation of certificates.

Considering the example again, let us add the assertion $r_1 = n$, used to justify the transformation, at program point L. Then, the transformed program is suitably annotated as:

$$\begin{array}{rl} & \{\mathsf{true}\} \\ & r_1 := n \\ L: & \{r_1 \geq n \wedge r_1 = n\},\ L' \\ L': & r_2 := n \\ & \{r_1 = r_2\} \end{array}$$

In this case the proof obligation $r_1 \geq n \wedge r_1 = n \Rightarrow n = r_1$ is provable, but still does not coincide with the original. In such a simple case, one could generate a certificate for the proof obligation without using the certificate of the original proof obligation, but in the general case we will need to build a new certificate from the certificate of the original proof obligation (here $r_1 \geq n \Rightarrow r_1 = r_1$).

A systematic approach to generate certificates for $\bar{f}$ is to define two functions that transform the certificates for f:

$$\begin{array}{ll} T_0: & \mathcal{C}(\vdash \mathsf{pre}(f) \Rightarrow \mathsf{wp}_f(L_{\mathsf{sp}})[\vec{r}/\vec{r^*}]) \to \mathcal{C}(\vdash \mathsf{pre}(\bar{f}) \Rightarrow \mathsf{wp}_{\bar{f}}(L_{\mathsf{sp}})[\vec{r}/\vec{r^*}]) \\ T_\lambda: & \forall L,\ \mathcal{C}(\vdash \phi_L \Rightarrow \mathsf{wp}^{\mathtt{id}}_f(L)) \to \mathcal{C}(\vdash \bar{\phi}_L \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\bar{f}}(L)) \end{array}$$

where ϕ_L is the original assertion at label L, and $\bar{\phi}_L$ is the augmented assertion at label L. Here the function T_0 transforms the proof that the precondition implies the assertion 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 precondition at program point L for f into a proof of the same fact for $\bar{f}$.

The functions T_0 and T_λ can be constructed, independently of the optimization considered, from a function

$$T^{\mathsf{ins}}_L:\ \mathcal{C}(\vdash \mathsf{wp}^{\mathtt{id}}_f(L) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\bar{f}}(L))$$

that associates to every program point L in f, a proof of the following fact: the original annotation of f (i.e. $\mathsf{wp}^{\mathtt{id}}_f(L)$) and the hypothesis obtained from the results of the analysis (i.e. $\mathrm{RES}_{\mathcal{A}}(L)$) imply the annotation of $\bar{f}$ (i.e. $\mathsf{wp}^{\mathtt{id}}_{\bar{f}}(L)$).

Let $\Gamma = [\mathsf{wp}_{\overline{f}}(L)]$ in:
$p_1{:=}\mathsf{axiom}(\mathsf{wp}_{\overline{f}}(L)) : \Gamma \vdash \mathsf{wp}_{\overline{f}}(L)$
$p_2{:=}\mathsf{elim}^{\mathsf{l}}_{\wedge}(p_1) : \Gamma \vdash \mathsf{wp}_f(L)$
$p_3{:=}\mathsf{elim}^{\mathsf{r}}_{\wedge}(p_1) : \Gamma \vdash \mathsf{wp}_{f_A}(L)$
$p_4{:=}\mathsf{weak}_{\Gamma}(\vec{\Lambda}(L)) : \Gamma \vdash \mathsf{wp}_f(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_f(L)$
$p_5{:=}\mathsf{elim}_{\Rightarrow}(p_2, p_4) : \Gamma \vdash \mathsf{wp}^{\mathtt{id}}_f(L)$
$p_6{:=}\mathsf{weak}_{\Gamma}(T^{\mathsf{ins}}_L(L)) : \Gamma \vdash \mathsf{wp}^{\mathtt{id}}_f(L) \Rightarrow \mathsf{wp}_{f_A}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$
$p_7{:=}\mathsf{elim}_{\Rightarrow}(p_5, p_6) : \Gamma \vdash \mathsf{wp}_{f_A}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$
$p_8{:=}\mathsf{elim}_{\Rightarrow}(p_3, p_7) : \Gamma \vdash \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$
$p_9{:=}\mathsf{intro}_{\Rightarrow}(p_8) : \vdash \mathsf{wp}_{\overline{f}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$

Fig. 10. Definition of $T_\lambda(\vec{\Lambda}(L))$ from T^{ins}_L

The function T_λ is defined using the function T^{ins}_L and the certificate of the analysis as shown in Figure 10. To define T_0, i.e. to generate $\overline{\lambda}$ from λ (recall that $\overline{\lambda}$ is a proof of $\mathsf{pre}(\bar{f}) \Rightarrow \mathsf{wp}_{\overline{f}}(L_{\mathsf{sp}})[{}^{\vec{r_g}}/_{\vec{r^*_g}}]$), we reuse λ (i.e. a proof of $\mathsf{pre}(f) \Rightarrow \mathsf{wp}_f(L_{\mathsf{sp}})[{}^{\vec{r_g}}/_{\vec{r^*_g}}]$), since $\mathsf{pre}(f)$ implies $\mathsf{wp}_f(L_{\mathsf{sp}})[{}^{\vec{r_g}}/_{\vec{r^*_g}}]$, and instantiating T^{ins}_L to L_{sp} we get a predicate equivalent to $\mathsf{wp}_f(L_{\mathsf{sp}}) \Rightarrow \mathsf{wp}_{\overline{f}}(L_{\mathsf{sp}})$. The reasoning is valid if we consider that the analysis is computed from the trivial precondition true and that $\mathsf{pre}(\bar{f})$ is defined equal to $\mathsf{pre}(f)$. That is the case for the intraprocedural analyses considered in this paper. To generalize the framework further to include interprocedural analyses, one would need to define $\mathsf{pre}(\bar{f})$ as $\mathsf{pre}(f) \wedge \mathrm{RES}_{\mathcal{A}}(L_{\mathsf{sp}})$.

Whereas the definition of T_0 and T_λ are generic, the function T^{ins}_L must be defined for each program optimization. It turns out that for many program optimizations it is possible to inductively define T^{ins}_L using the definition of $T^{\mathsf{ins}}_{L_1}, \ldots, T^{\mathsf{ins}}_{L_k}$, where $\{L_1, \ldots, L_k\}$ are the successor program points for L. Due to the presence of loops, T^{ins}_L may be not well defined in the general case. However, we only consider well-annotated programs, and the definition of well-annotated programs induces an induction principle with annotated program labels as the base case.

In summary, the definition of a certificate translator for a program optimization requires defining a certifying analyzer, and a function T^{ins}_L with suitable characteristics. In the rest of this section, we show how to define these for many common program optimizations.

4.2 Constant Propagation

4.2.1 *Description.* Constant propagation aims at minimizing run-time evaluation of expressions and access to registers with constant values. It relies on a data flow analysis that returns a function $\mathcal{A}$ with type $\mathcal{PP} \times \mathcal{R} \to \mathbb{Z}_\perp$ ($\mathcal{PP}$ denoting the set of program points) such that $\mathcal{A}(L, r) = n$ indicates that r holds the value n every time execution reaches the label L. If the analysis cannot infer that a register r holds a constant value at label L then we write $\mathcal{A}(L, r) = \perp$.

The definition of constant propagation over a function f can be found in Figure 11. Exploiting the information provided by $\mathcal{A}$, the optimization consists of replacing instructions that read registers by equivalent instructions that read con-

$$
\begin{aligned}
\llbracket(\phi,\ \mathsf{ins})\rrbracket_L &= (\phi \wedge \mathrm{RES}_{\mathcal{A}}(L),\ \llbracket\mathsf{ins}\rrbracket_L^{\mathtt{id}}) \\
\llbracket\mathsf{ins}\rrbracket_L &= \llbracket\mathsf{ins}\rrbracket_L^{\mathtt{id}}
\end{aligned}
$$

$$
\begin{aligned}
\llbracket r_d := \mathsf{op},\ L'\rrbracket_L^{\mathtt{id}} &= r_d := \llbracket\mathsf{op}\rrbracket_L^{\mathsf{op}},\ L' \\
\llbracket\mathsf{cmp}\ ?\ L_t : L_f\rrbracket_L^{\mathtt{id}} &= \begin{cases} \mathsf{nop},\ L_t & \text{when } \llbracket\mathsf{cmp}\rrbracket_L^{\mathsf{cmp}} = \mathsf{true} \\ \mathsf{nop},\ L_f & \text{when } \llbracket\mathsf{cmp}\rrbracket_L^{\mathsf{cmp}} = \mathsf{false} \\ \llbracket\mathsf{cmp}\rrbracket_L^{\mathsf{cmp}}\ ?\ L_t : L_f & \text{otherwise} \end{cases} \\
\llbracket\mathsf{ins}\rrbracket_L^{\mathtt{id}} &= \mathsf{ins} \qquad \text{in any other cases}
\end{aligned}
$$

$$
\begin{aligned}
\llbracket r\rrbracket_L^{\mathsf{op}} &= \begin{cases} n & \text{if } \mathcal{A}(L,r) = n \\ r & \text{otherwise} \end{cases} \\
\llbracket r_1 + r_2\rrbracket_L^{\mathsf{op}} &= \begin{cases} n & \text{if } \mathcal{A}(L,r_i) = n_i \\ & \quad \text{and } n = n_1 + n_2 \\ r_2 & \text{if } \mathcal{A}(L,r_1) = 0 \\ r_1 & \text{if } \mathcal{A}(L,r_2) = 0 \\ n_1 + r_2 & \text{if } \mathcal{A}(L,r_1) = n_1 \\ n_2 + r_1 & \text{if } \mathcal{A}(L,r_2) = n_2 \\ r_1 + r_2 & \text{in any other cases} \end{cases} \\
\llbracket r_1 \lhd r_2\rrbracket_L^{\mathsf{cmp}} &= \begin{cases} \mathsf{true} & \text{if } \mathcal{A}(L,r_i) = n_i \text{ and } n_1 \lhd n_2 \\ \mathsf{false} & \text{if } \mathcal{A}(L,r) = n_i \text{ and } \neg(n_1 \lhd n_2) \\ r_1 \lhd r_2 & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 11. Constant Propagation

stants. Furthermore, in the case of a conditional instruction, if the truth value of an integer comparison can be statically determined, it is replaced by a jump instruction to the corresponding branching point.

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 integer n s.t. $n = n_1 + n_2$. If only one register is known to be equal to 0 the compiler replaces the addition operation by a move instruction. 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 the other arithmetic operations, but are not shown in Figure 11.

The optimized function $\overline{f}$ will be such that $\overline{f}[L] = \llbracket f[L]\rrbracket_L$ for every label L in the domain of G_f. The transformation of an annotated instruction is defined as a transformation of the annotation and the transformation of the instruction descriptor.

4.2.2 *Certifying analyzer.* We have implemented a certifying analyzer for constant propagation as an extension of the standard analysis algorithm. First, we attach to each reachable label L the assertion $\mathrm{RES}_{\mathcal{A}}(L)$:

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

To derive a certificate for the analysis we must, for each reachable label L, generate a proof for the judgment

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

After applying $\mathsf{elim}_{\Rightarrow}$ (i.e. moving hypothesis to the context), and rewriting equalities from the context in the goal, one is left to prove closed equalities of the form $n = n'$ (i.e. n, n' are constants and do not contain variables). If the assertions are correct, then the certificate is obtained by applying reflexivity of equality (an instance of the ring rule).

4.2.3 *Certificate translation.* Suppose we have a certified function f with declaration $\{\vec{r_g};\ \varphi;\ G;\ \psi;\ \lambda;\ \vec{\Lambda}\}$. 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 $\overline{\vec{\Lambda}}$ (the set proof obligations generated by the intermediate assertions), for each instruction of the form $\overline{f}[L] = (\phi \wedge \mathrm{RES}_{\mathcal{A}}(L),\ \mathsf{ins})$ we have to find a proof for $\vdash \phi \wedge \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$. To this end, we rely on an auxiliary function T^{ins}_{L} of type

$$T^{\mathsf{ins}}_{L}\ : \mathcal{C}(\vdash \mathsf{wp}^{\mathtt{id}}_{f}(L) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L))$$

for every label L, and on the function T_{λ} defined in Section 4.1 to construct a certificate for $\vdash \phi \wedge \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$. Furthermore, for every local substitution of op by $[\![\mathsf{op}]\!]^{\mathsf{op}}_{L}$ performed by the optimization, we require a certificate $T_{\mathsf{op}}(\mathsf{op}, L)$ for $\vdash \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \langle \mathsf{op} \rangle = \langle [\![\mathsf{op}]\!]^{\mathsf{op}}_{L} \rangle$.

The certificate T^{ins}_{L} represents the fact that under the hypotheses that the result of the analysis is correct, if a program state satisfies $\mathsf{wp}^{\mathtt{id}}_{f}(L)$ then it will also satisfy $\mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$. The definition of the constructor T^{ins}_{L} is detailed in Fig. 12 for the assignment case. In the figure, the auxiliary function T_L of type $\mathcal{C}(\vdash \mathsf{wp}_{f}(L) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}_{\overline{f}}(L))$ is defined equal to T^{ins}_{L} when $f[L]$ does not contain an assertion. Otherwise, T_L has type $\mathcal{C}(\vdash \phi \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \phi \wedge \mathrm{RES}_{\mathcal{A}}(L))$ for some ϕ and, thus, it is trivially defined.

Example 4.1. *Consider as an example the following program transformation:*

$$\begin{array}{lll}
L_1\ : \{r_2 \geq 1\} & & L_1\ : \{r_2 \geq 1\} \\
L_2\ : r_1 := 1 & & L_2\ : r_1 := 1 \\
L_3\ : \{r_2 \geq r_1 \wedge r_1 \geq 0\} & & L_3\ : \{r_2 \geq r_1 \wedge r_1 \geq 0\} \\
L_4\ : r_3 := r_1 & \longrightarrow & L_4\ : r_3 := 1 \\
L_5\ : \{r_2 \geq r_3 \wedge r_3 = r_1 \wedge r_1 \geq 0\} & & L_5\ : \{r_2 \geq r_3 \wedge r_3 = r_1 \wedge r_1 \geq 0\} \\
L_6\ : r_2 := r_2 + r_3 & & L_6\ : r_2 := r_2 + 1 \\
L_7\ : \mathtt{nop},\ L_3 & & L_7\ : \mathtt{nop},\ L_3
\end{array}$$

We have originally a proof obligation generated at L_3:

$$r_2 \geq r_1 \wedge r_1 \geq 0 \Rightarrow r_2 \geq r_1 \wedge r_1 = r_1 \wedge r_1 \geq 0$$

and a proof obligation at L_5:

$$r_2 \geq r_3 \wedge r_3 = r_1 \wedge r_1 \geq 0 \Rightarrow r_2 + r_3 \geq r_1 \wedge r_1 \geq 0\ .$$

Let

$$\begin{aligned} f[L] &= r := \mathsf{op},\ L' \\ \Gamma &= [\phi_1; \mathrm{RES}_{\mathcal{A}}(L)] \\ \phi_1 &= \mathsf{wp}_f(L')[{}^{\mathsf{op}}\!/_{r_d}] \\ \phi_2 &= \mathrm{RES}_{\mathcal{A}}(L')[{}^{\mathsf{op}}\!/_{r_d}] \\ \phi_3 &= \mathsf{wp}_{\bar{f}}(L')[{}^{\mathsf{op}}\!/_{r_d}] \end{aligned}$$

in:

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

Fig. 12. Definition of T^{ins}_L for assignment case

The first proof obligation becomes $r_2 \geq r_1 \wedge r_1 \geq 0 \Rightarrow r_2 \geq 1 \wedge 1 = r_1 \wedge r_1 \geq 0$ *after the program transformation. Clearly, in order to obtain a proof of it, it is necessary to introduce the condition* $r_1 = 1$ *in the antecedent. This motivates the need for the hypothesis about the result of the analysis, in this case* $r_1 = 1$.

However, this introduction is not always needed. For example, the second condition becomes $r_2 \geq r_3 \wedge r_3 = r_1 \wedge r_1 \geq 0 \Rightarrow r_2 + 1 \geq r_1 \wedge r_1 \geq 0$ *in the optimized code, and the assertion* $r_1 = 1$ *is not necessary at* L_5 *in order to prove the verification condition (unless it is also introduced at* L_3*.)*

4.3 Loop Induction Variable Strength Reduction

4.3.1 *Description.* Loop induction strength reduction aims at reducing the number of multiplication operations inside a loop, which are commonly more costly than addition operations. An *induction register* is a register that is incremented (or decremented) in each iteration of the loop by a fixed constant value. An induction register is defined in the loop by an instruction of the form $r_i := r_i + c$, where c is a constant value. A *derived induction register* is a register that is assigned in each iteration of the loop the value of a linear function on the induction register. A derived induction register is defined in the loop by an instruction of the form $r_d := b * r_i$, where b is a constant value. The optimization consists of replacing any instruction updating a derived induction register r_d, made in terms of the basic induction register r_i, by an increment made only in terms of the previous value of r_d. For example, in

$$\begin{aligned} L_{oop}\ :\ & r_i := r_i + c \\ & r_d := b * r_i \\ & \mathtt{nop},\ L_{oop} \end{aligned}$$

$$\begin{array}{lcl} f'[L''_H] & = & r'_d := b * r_i,\ L_H \\ f'[L_H] & = & f[L_H] \\ f'[L_i] & = & r_i := r_i + c,\ L''_i \\ f'[L''_i] & = & r'_d := r'_d + b * c,\ L'_i \\ f'[L] & = & \begin{cases} f[L] & \text{if } L \text{ is any other label inside the loop} \\ f[L][{}^{L''_H}/_{L_H}] & \text{if } L \text{ is a label outside the loop} \end{cases} \end{array}$$

Fig. 13. Loop Induction: First Transformation Step

r_i is a basic induction register with an increment of c, and r_d is an induction register with coefficient b derived from r_i. The optimization replaces the assignment $r_d := b * r_i$ by the less costly assignment $r_d := r_d + b * c$ and introduces the initialization $r_d := b * r_i$ just before the head of the loop. In addition to reducing the cost of the instruction, the live range of the register r_i is reduced, enabling further optimizations.

In the sequel, we follow the simplifying assumption that the loop body contains a single assignment for each register r_i and r_d. We assume also that the compiler has the ability to detect loops and returns a set of labels $\{L_1, \ldots, L_n\}$ determining the loop body, from which the header label L_H is the unique entry point.

Let $\{\vec{x};\ \varphi;\ G;\ \psi;\ \lambda;\ \vec{\Lambda}\}$ be the declaration for function f. Strength reduction proceeds in two steps. In the first step, an analysis detects inside the loop an induction register r_i and a derived induction register r_d. More precisely, the analysis takes as input a set of labels $\{L_1, \ldots, L_n\}$ (we assume that this set of labels corresponds to the output of a loop analysis, and that the header label is L_H) and provides the following information: an induction register r_i and the label L_i in which it is updated, a derived induction register r_d and the label L_d in which its definition appears, a fresh register name r'_d, two new labels L''_i and L''_H not in the domain of G_f and two constant values b, c that correspond respectively to the coefficient of r_d and the increment of r_i. The first transformation step consists on introducing two assignments to a fresh register r'_d, one immediately before the loop header and the other one immediately after the assignment $f[L_i] = r_i := r_i + c,\ L'_i$. The output function at this transformation step is named f' and is defined in Figure 13. The motivation for this transformation phase is to ensure the invariance of the condition $r'_d = b * r_i$ in the loop body. In the figure, labels L', L'_H, L'_i and L'_d are respectively the successor labels for the instructions at L, L_H, L_i and L_d. To ensure that the instruction at L''_H is always executed before entering the loop, we update the labels outside the loop replacing L_H with L''_H.

In a second step, an analysis determines the invariance of the condition $r'_d = b * r_i$ and consequently an optimization replaces the assignment to the derived induction register $r_d := b * r_i$ by the less costly assignment $r_d := r'_d$. We define the optimized function $\overline{f}$ as $\overline{f}[L_d] = r_d := r'_d,\ L'_d$ and $\overline{f}[L] = f'[L]$ for every $L \neq L_d$. In addition, every annotation inside the loop body is augmented with the condition $r'_d = b * r_i$.

4.3.2 *Certifying analyzer.* Only the analysis of the second step must be certified. To annotate f' with the result of the analysis, we define $\mathrm{RES}_{\mathcal{A}}(L)$ as $r'_d = b * r_i$ if L is in $\{L_1, \ldots, L_n\}$, as $r'_d = b * (r_i - c)$ for $L = L''_i$ and $\mathrm{RES}_{\mathcal{A}}(L) \equiv \mathsf{true}$ in any other case (i.e. when L is a label outside the loop.) Then, we need to create a certificate that the analysis is correct. The definition of $f'_{\mathcal{A}}$ is given by $f'_{\mathcal{A}}[L] = (\mathrm{RES}_{\mathcal{A}}(L),\ \mathsf{ins})$, where $f[L] = \mathsf{ins}$ or $f[L] = (\phi,\ \mathsf{ins})$. Since $\mathrm{RES}_{\mathcal{A}}(L)$ refers only to registers r'_d and r_i, the only interesting proof obligations are those corresponding to program labels L''_H, L_i and L''_i. 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. That is, if the loop analysis is not correct, then the certificate cannot be constructed.

4.3.3 *Certificate translation.* Certificate translation from a certificate for f into a certificate for $\overline{f}$ is also performed in two steps. In the first one, we build a certificate for f' from a certificate for f. Then we build a certificate for $\overline{f}$ from the certificate for f'.

The first translation step is trivial due to preservation of proof obligations. More precisely, we can show that since r'_d is a fresh register and, hence, does not appear in the code nor in assertions, the introduction of assignments targeting register r'_d does not affect the computation of the function $\mathsf{wp}^{\mathtt{id}}_f$. Formally, we can prove by the induction principle induced by the definition of well-annotated programs, that for every $L \notin \{L''_H, L''_i\}$, $\mathsf{wp}^{\mathtt{id}}_{f'}(L) = \mathsf{wp}^{\mathtt{id}}_f(L)$. Then, since no proof obligations are introduced at L''_H or L''_i, the set of proof obligation $\{\phi \Rightarrow \mathsf{wp}^{\mathtt{id}}_{f'}(L) \mid f'[L] = (\phi,\ \mathsf{ins})\}$ corresponds to the original set of proof obligations $\{\phi \Rightarrow \mathsf{wp}^{\mathtt{id}}_f(L) \mid f[L] = (\phi,\ \mathsf{ins})\}$. Therefore, the original certificate can be reused without modifications.

Certificate translation for the transformation from f' to $\overline{f}$ proceeds as with constant propagation or any certificate translation that requires a certifying analyzer. That is, we need to give explicitly a proof of $\mathsf{wp}_{\overline{f}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$ for each instruction of the form $\overline{f}[L] = (\phi \wedge \mathrm{RES}_{\mathcal{A}}(L),\ \mathsf{ins})$. We can avoid repeating the main process since it is the same that was specified in the previous section (constant propagation). However it remains to define T^{ins}_L. Intuitively, the function T^{ins}_L expresses the fact that for every program point L inside the loop, $\mathsf{wp}^{\mathtt{id}}_{f'}(L)$ and $\mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$ are equivalent, provided the condition $r'_d = b * r_i$ is valid. In this case, the annotations for function f' corresponding to labels outside the loop are not modified.

$$T^{\mathsf{ins}}_L\colon\ \mathcal{C}(\vdash \mathsf{wp}^{\mathtt{id}}_{f'}(L) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L))$$

This function T^{ins}_L is constructed using the induction principle attached to the definition of well-annotated programs, and then this result is merged with the original certificate and the certificate of the analysis, to produce a certificate for $\overline{f}$. In Figure 14 we show the definition of T^{ins}_L for the case $L = L_d$.

Let $\Gamma = [\mathsf{wp}^{\mathtt{id}}_{f'}(L_d), \mathrm{RES}_{\mathcal{A}}(L_d)]$ in

$$\begin{aligned}
p_1 &:= T_L(L'_d) : \vdash \mathsf{wp}_{f'}(L'_d) \Rightarrow \mathsf{wp}_{f'_{\mathcal{A}}}(L'_d) \Rightarrow \mathsf{wp}_{\overline{f}}(L'_d)\\
p_2 &:= \mathsf{subst}\ r_d\ b{*}r_i(p_1) : \vdash \mathsf{wp}^{\mathtt{id}}_{f'}(L_d) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{f'_{\mathcal{A}}}(L_d) \Rightarrow \mathsf{wp}_{\overline{f}}(L'_d)[{}^{b*r_i}\!/_{r_d}]\\
p_3 &:= \mathsf{axiom}(\mathsf{wp}^{\mathtt{id}}_{f'}(L_d)) : \Gamma \vdash \mathsf{wp}^{\mathtt{id}}_{f'}(L_d)\\
p_4 &:= \mathsf{axiom}(\mathrm{RES}_{\mathcal{A}}(L_d)) : \Gamma \vdash \mathrm{RES}_{\mathcal{A}}(L_d)\\
p_5 &:= \mathsf{weak}_{\Gamma}(\vec{\Lambda}_{\mathcal{A}}(L_d)) : \Gamma \vdash \mathrm{RES}_{\mathcal{A}}(L_d) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{f'_{\mathcal{A}}}(L_d)\\
p_6 &:= \mathsf{elim}_{\Rightarrow}(p_4, p_5) : \Gamma \vdash \mathsf{wp}^{\mathtt{id}}_{f'_{\mathcal{A}}}(L_d)\\
p_7 &:= \mathsf{elim}_{\Rightarrow}(p_3, \mathsf{weak}_{\Gamma}(p_2)) : \Gamma \vdash \mathsf{wp}^{\mathtt{id}}_{f'_{\mathcal{A}}}(L_d) \Rightarrow \mathsf{wp}_{\overline{f}}(L'_d)[{}^{b*r_i}\!/_{r_d}]\\
p_8 &:= \mathsf{elim}_{\Rightarrow}(p_6, p_7) : \Gamma \vdash \mathsf{wp}_{\overline{f}}(L'_d)[{}^{b*r_i}\!/_{r_d}]\\
p_9 &:= \mathsf{elim}_{=}(p_4, p_8) : \Gamma \vdash \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L_d)\\
T^{\mathsf{ins}}_L(L_d) &:= \mathsf{intro}_{\Rightarrow}(\mathsf{intro}_{\Rightarrow}(p_9)) : \vdash \mathsf{wp}^{\mathtt{id}}_{f'}(L_d) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L_d) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L_d)
\end{aligned}$$

Fig. 14. Definition of T^{ins}_L for the case $L = L_d$

EXAMPLE 4.2. *Consider the following program, where $n \in \mathbb{N}$:*

$$\begin{array}{rl}
 & r := 0\\
 & r_i := 0\\
L_{oop}: & \{r = b * r_i \wedge r_i \leq n\}\\
 & r_i \geq n\ ?\ L_{out}\\
 & r_i := r_i + 1\\
 & r_d := b * r_i\\
 & r := r_d\\
 & \mathtt{nop},\ L_{oop}\\
L_{out}: & \{r = b * n\}
\end{array}$$

One can apply the optimization of this section to reduce the strength of the instruction updating the derived induction register r_d. Focusing on the proof obligation corresponding to the preservation of the loop invariant, we can extract the interesting fragment

$$r = b * r_i \wedge r_i \leq n \wedge r_i < n \Rightarrow b * (r_i + 1) = b * (r_i + 1)\ .$$

If the program is transformed as follows (strength reduction+copy propagation)

$$\begin{array}{rl}
 & r := 0\\
 & r_i := 0\\
 & r'_d := 0\\
L_{oop}: & \{r = b * r_i \wedge r_i \leq n\}\\
 & r_i \geq n\ ?\ L_{out}\\
 & r_i := r_i + 1\\
 & r'_d := r'_d + b\\
 & r := r'_d\\
 & \mathtt{nop},\ L_{oop}\\
L_{out}: & \{r = b * n\}
\end{array}$$

the fragment of the proof obligation becomes

$$r = b * r_i \wedge r_i \leq n \wedge r_i < n \Rightarrow r'_d + b = b * (r_i + 1)\ .$$

The condition $r'_d = b * r_i$ is missing in the antecedent, and adding it in is mandatory to verify the code. At the same time, this extension of the invariant must be verified, a task that corresponds to the proof obligations automatically generated by the certifying analyzer.

4.4 Common Subexpression Elimination

4.4.1 *Description.* Common subexpression elimination (CSE) aims at reducing the number of duplicated computations by reusing previously defined and still available nontrivial expressions: if the same expression is computed in two different program points, CSE eliminates one of the computations, by replacing the second operation by an access to the register containing the result of the first evaluation.

CSE is similar to constant propagation, in the sense that the transformation is triggered by conditions represented by an equality between a register and an expression. In constant propagation this expression corresponds to a constant value, whereas in CSE it may be a more complex expression (commonly involving arithmetic operators). Therefore, the principle behind certificate translation for CSE is very similar to the one for certificate translation for CP. In the following example

$$\begin{array}{l} r_1 := r_2 + r_3 \\ r_4 := r_2 + r_3 \\ r_1 := r_1 + 1 \\ r_5 := r_2 + r_3 \end{array} \quad \longrightarrow \quad \begin{array}{l} r_1 := r_2 + r_3 \\ r_4 := r_1 \\ r_1 := r_1 + 1 \\ r_5 := r_4 \end{array}$$

the assignment updating the register r_4 could be optimized, but after the second assignment over r_1, the availability of the expression $r_2 + r_3$ through r_1 is lost, and then the assignment to r_5 has to be optimized with r_4.

We begin by discussing the certifying analyzer for CSE. The function $\mathcal{A}$ with type $\mathcal{PP} \times \mathcal{R} \to \mathsf{op}_\perp$ associates to a label L and a register r an RTL expression. Similarly to constant propagation, the assertion $\mathrm{RES}_\mathcal{A}$ is defined as:

$$\mathrm{RES}_\mathcal{A}(L) \equiv \bigwedge_{\mathcal{A}(L,r)\neq\perp} r = \langle \mathcal{A}(L,r) \rangle$$

where $\langle \mathcal{A}(L,r) \rangle$ is the interpretation into the language of assertions of the information returned by the analysis.

The certifying analyzer generates automatically a certificate for $f_\mathcal{A}$, where for every label L, the corresponding annotation will be $\mathrm{RES}_\mathcal{A}(L)$.

We briefly describe the transformation for CSE. For each $f[L]$ of the form $r := e$, if there is a register r' such that $r' \neq r$ and $\mathcal{A}(L, r') = e$, we replace the instruction $f[L]$ by $r := r'$.

Finally, certificate translation proceeds exactly as with constant propagation, with the definition of two mutually recursive transfer functions using the induction principle attached to the definition of $\mathsf{reachAnnot}_f$ (Definition 2.1):

$$\begin{array}{l} T_L^{\mathsf{ins}}\colon\ \forall L,\ \mathcal{C}(\vdash \mathsf{wp}_f^{\mathtt{id}}(L) \Rightarrow \mathrm{RES}_\mathcal{A}(L) \Rightarrow \mathsf{wp}_{\overline{f}}^{\mathtt{id}}(L)) \\ T_L\ :\ \forall L,\ \mathcal{C}(\vdash \mathsf{wp}_f(L) \Rightarrow \mathrm{RES}_\mathcal{A}(L) \Rightarrow \mathsf{wp}_{\overline{f}}(L)) \end{array}$$

Then, these two functions are used to merge the original certificate with the certificate of the analysis to build a certificate for $\overline{f}$.

EXAMPLE 4.3. *Consider the previous example with intermediate assertions:*

$$\begin{array}{lcl}
\{0 \leq r_2 \wedge 0 \leq r_3\} & & \{0 \leq r_2 \wedge 0 \leq r_3\} \\
r_1 := r_2 + r_3 & & r_1 := r_2 + r_3 \\
\{0 \leq r_2 \wedge 0 \leq r_3\} & & \{0 \leq r_2 \wedge 0 \leq r_3\} \\
r_4 := r_2 + r_3 & & r_4 := r_1 \\
\{0 \leq r_4 \wedge 0 \leq r_2 \wedge 0 \leq r_3\} & \rightarrow & \{0 \leq r_4 \wedge 0 \leq r_2 \wedge 0 \leq r_3\} \\
r_1 := r_1 + 1 & & r_1 := r_1 + 1 \\
r_5 := r_2 + r_3 & & r_5 := r_4 \\
\{0 \leq r_5\} & & \{0 \leq r_5\}
\end{array}$$

If the instruction $r_4 := r_2 + r_3$ is replaced with $r_4 := r_1$, the lack of information about register r_1 on assertions prevents proving the condition $0 \leq r_4$ of the third assertion. Then, we need to propagate $r_1 = r_2 + r_3$ to the second assertion. Finally, although there is enough information about register r_4 on the second assertion to prove that $0 \leq r_5$ is valid after the assignment, the original proof is done in terms of r_2 and r_3. Hence, the need for a general and automatic technique forces us to introduce the condition $r_4 = r_2 + r_3$.

The $f_{\mathcal{A}}$ and $\overline{f}$ functions are respectively:

$$\begin{array}{ll}
\{\mathsf{true}\} & \{0 \leq r_2 \wedge 0 \leq r_3\} \\
r_1 := r_2 + r_3 & r_1 := r_2 + r_3 \\
\{r_1 = r_2 + r_3\} & \{0 \leq r_2 \wedge 0 \leq r_3 \wedge r_1 = r_2 + r_3\} \\
r_4 := r_2 + r_3 & r_4 := r_1 \\
\{r_4 = r_2 + r_3\} & \{0 \leq r_4 \wedge 0 \leq r_2 \wedge 0 \leq r_3 \wedge r_4 = r_2 + r_3\} \\
r_1 := r_1 + 1 & r_1 := r_1 + 1 \\
r_5 := r_2 + r_3 & r_5 := r_4 \\
\{\mathsf{true}\} & \{0 \leq r_5\}
\end{array}$$

4.5 Copy Propagation

4.5.1 *Description.* Copy propagation aims at reducing the live range of variables defined by move operations, simplifying register allocation at the code generation phase. It is intended to remove the number of auxiliary register copies that may be introduced by other optimizations.

Copy propagation searches for occurrences of instructions of the form $r_1 := r_2$, and replaces every occurrence of r_1 by r_2 (along its successors' path) until either r_1 or r_2 are modified. The original assignment $r_1 := r_2$ can be moved forward in the code until the condition $r_1 = r_2$ gets invalidated.

4.5.2 *Certificate translation.* As with constant propagation, common subexpression elimination and many other optimizations, the translation of the certificate can be made by using a certifying analyzer. First, a special purpose function f_A is generated, fully annotated with the results of the dataflow analysis (e.g. $r_1 = r_2$), and a new certificate is automatically generated validating these auxiliary assertions. Formally, after performing the analysis, the function $\mathcal{A} : \mathcal{PP} \times \mathcal{R} \rightarrow \mathcal{R}_\perp$, such that $\mathcal{A}(L, r_1) = r_2$ if r_1 holds a copy of the value at r_2, is used to define

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

and to generate a certificate for the analysis, defining for each reachable label L the certificate

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

that is later integrated with the certificate of the original program. The function T^{ins}_{L} is defined by case analysis. The proof for T^{ins}_{L} does not represent any further difficulty with respect to that of constant propagation or common subexpression elimination.

4.6 Dead Register Elimination

4.6.1 *Description.* Dead register elimination aims at removing assignments to registers that will not be needed in the future. As mentioned in the introduction, we propose a transformation that removes dead assignments while simultaneously modifying the assertions. Regarding certificate translation, this program transformation is atypical since it does not follow the scheme consisting on a certifying analyzer and a definition of the function T^{ins}_{L}.

Intuitively, a register is live at some execution point if it stores a value that interferes with the subsequent execution steps. The classical definition is intentional and overapproximating: 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 as a parameter of a function call, in a conditional jump, in a `return` instruction, or on the right side of an assignment to a live register r'.

In the following, we represent with the function $\mathcal{L}$ the result of the analysis and write $\mathcal{L}(L, r) = \top$ when a register is live at L.

A live range of a register r is a set of sequences of consecutive input or output edges such that r is live.

The problem with certificate translation for removal of dead registers is that intermediate assertions can specify conditions over local variables that will never be used, and hence are dead in the program. Arguably, we may assume that original assertions only refer to live variables. However, some optimizations can reduce the live range of variables and, therefore, the occurrence of some registers may become redundant after previous optimizations steps. For example in the following optimization

$$\begin{array}{lcl} \{\mathsf{true}\} & & \{\mathsf{true}\} \\ r_1 := 1 & & r_1 := 1 \\ \{r_1 = 1\} & \longrightarrow & \{r_1 = 1\} \\ r_2 := r_1 & & r_2 := 1 \\ \{r_2 = 1\} & & \{r_2 = 1\} \end{array}$$

the register r_1 becomes dead, but the assignment $r_1 := 1$ cannot be removed since we have to ensure that $r_1 = 1$ is a valid intermediate assertion.

In order to deal with assertions, we extend the definition of liveness.

Definition Live in 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 but there is a path from L that reaches a label L' such that r appears in an assertion at L' or where r is used to define a register which is live in an assertion at label L'.*

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

In order to deal with registers that are dead in the code but live in assertions, we rely on the introduction of ghost variables. Ghost variables are expressions in our language of assertions (we assume that the set of ghost variables names and the set of register names $\mathcal{R}$ are disjoint). We introduce, as part of the RTL syntax, non-executable *ghost assignments*, represented as instructions of the form 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 wp in the same way as normal assignments. At the end of this section we discuss the soundness of the verification infrastructure in presence of ghost variables.

In the following, if σ is a function mapping registers to expressions, we denote $\phi\sigma$ the result of substituting every free register r in ϕ by σr.

We propose a transformation, defined by the following equations:

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

where the substitution σ_L maps r to $\hat{r}$ if $\mathcal{L}(L, r) = \top_\phi$ and $\mathsf{dead}_c(L, L') = \{r \mid \mathcal{L}(L, r) = \top \wedge \mathcal{L}(L', r) = \top_\phi\}$ and the transformation function $\mathsf{ghost}_L^{\mathsf{id}}(\mathsf{ins})$ is defined in Figure 15. We use set $\hat{\vec{r}} := \vec{r}$ as syntactic sugar for a sequence of assignments set $\hat{r_i} := r_i$ where for each register r_i in the sequence $\vec{r}$, $\hat{r_i}$ in $\hat{\vec{r}}$ is its corresponding ghost variable. Each instruction of f is transformed into a sequence of instructions for $\overline{f}$. Intuitively, it introduces for every instruction ins (with successor L') at label L, a ghost assignment 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, every instruction of the form $r_d := \mathsf{op}$ is removed if the register r_d is not live at L.

4.6.2 *Certificate translation.* 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 annotations up to substitution of dead variables.

LEMMA 4.1. *After applying ghost variable introduction, the transformed function $\overline{f}$ satisfies the following property:*

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

where σ is the substitution defined above.

PROOF. *The proof is by the induction principle attached to the definition of* reachAnnot *(Definition 2.1). We only consider some representative cases:*

— *Case $f[L] = (\phi,\ \mathsf{ins})$. In this case $\mathsf{wp}_{\overline{f}}(L)$ is equal to $\phi\sigma_L$ by definition of* wp *and ghost variable introduction. But, since ϕ is equal to $\mathsf{wp}_f(L)$, the lemma is satisfied.*

— *Case $f[L] = \mathtt{nop},\ L'$. By definition of* wp*, we have that $\mathsf{wp}_{\overline{f}}(L)$ is equal to $\mathsf{wp}_{\overline{f}}(L')$, which in turn is equal to $\mathsf{wp}_f(L')\sigma_{L'}$ by inductive hypothesis. Since $\mathsf{wp}_f(L') = \mathsf{wp}_f(L)$, it remains to prove that $\sigma_{L'}$ is in fact equal to σ_L, but this is the case since the condition of liveness or liveness in assertion is the same for L*

$$
\begin{array}{lcl}
\mathsf{ghost}_L^{\mathtt{id}}(\mathtt{return}\ r) & = & \mathtt{return}\ r \\
\mathsf{ghost}_L^{\mathtt{id}}(r_d := f(\vec{r}),\ L') & = & \left| \begin{array}{lcl} L & : & r_d := f(\vec{r}),\ L'' \\ L'' & : & \mathsf{set}\ \hat{\vec{t}} := \vec{t},\ L' \end{array} \right. \\
\mathsf{ghost}_L^{\mathtt{id}}(\mathtt{nop},\ L') & = & \mathtt{nop},\ L' \\
\mathsf{ghost}_L^{\mathtt{id}}(\mathsf{cmp}\ ?\ L_1 : L_2) & = & \left| \begin{array}{lcl} L & : & \mathsf{cmp}\ ?\ L'_1 : L'_2 \\ L'_1 & : & \mathsf{set}\ \hat{\vec{t_1}} := \vec{t_1},\ L_1 \\ L'_2 & : & \mathsf{set}\ \hat{\vec{t_2}} := \vec{t_2},\ L_2 \end{array} \right. \\
\mathsf{ghost}_L^{\mathtt{id}}(r_d := \mathsf{op},\ L') & = & \begin{cases} \mathtt{nop},\ L' & \text{if } \mathcal{L}(L', r_d) = \bot \\ \mathsf{set}\ \hat{r}_d := \mathsf{op}\sigma_L,\ L' & \text{if } \mathcal{L}(L', r_d) = \top_\phi \\ \left| \begin{array}{lcl} L & : & r_d := \mathsf{op},\ L'' \\ L'' & : & \mathsf{set}\ \hat{\vec{t}} := \vec{t},\ L' \end{array} \right. & \text{if } \mathcal{L}(L', r_d) = \top \end{cases}
\end{array}
$$

where $\vec{t}$, $\vec{t_1}$ and $\vec{t_2}$ stand respectively for variables in the sets $\mathsf{dead}_c(L, L')$, $\mathsf{dead}_c(L, L_1)$ and $\mathsf{dead}_c(L, L_2)$

Fig. 15. Ghost Variable Introduction-Dead Register Elimination

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 $f[L] = r_d := \mathsf{op},\ L'$ and $\mathcal{L}(L', r_d) = \top_\phi$. By definition of wp and the transformation performed we have that $\mathsf{wp}_{\overline{f}}(L)$ is equal to $\mathsf{wp}_{\overline{f}}(L')[{}^{\mathsf{op}\sigma_L}/_{\hat{r_d}}]$, and by I.H., to $\mathsf{wp}_f(L')\sigma_{L'}[{}^{\mathsf{op}\sigma_L}/_{\hat{r_d}}]$. Now we have to see that*

$$[{}^{\mathsf{op}\sigma_L}/_{\hat{r_d}}] \circ \sigma_{L'} = \sigma_L \circ [{}^{\mathsf{op}}/_{r_d}] \ .$$

To prove this equation we proceed by case analysis. First, consider the application of the substitutions to register r_d, in this case we can 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 $\mathsf{wp}_f(L')$ 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 $\mathsf{wp}_f(L')$, 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 $\mathsf{wp}_f(L')$, then r must be live on label L', which implies that is is also live on label L, and that is a contradiction.

□

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 $\overline{f}$. More precisely, for every label L s.t. $f[L]$ is of the form $(\phi_L,\ \mathsf{ins})$ we can obtain a proof of $\vdash \phi_L\sigma_L \Rightarrow \mathsf{wp}_f^{\mathtt{id}}(L)\sigma_L$ (i.e. of $\vdash \mathsf{wp}_{\overline{f}}(L) \Rightarrow \mathsf{wp}_{\overline{f}(L)}^{\mathtt{id}}$) by applying subst rule of Figure 5 to the original proof for $\vdash \phi_L \Rightarrow \mathsf{wp}_f^{\mathtt{id}}(L)$.

After ghost variable introduction has been applied, every register that occurs free in $\mathsf{wp}_{\overline{f}}(L)$ is live at L.

Soundness of Ghost Variable Introduction. To consider the proposed transformation as an optimizing one, the execution environment must be capable of safely ignoring ghost assignments. Intuitively, the set of ghost variables do not affect the state of any other variable nor the control flow (that is because ghost variables do not appear on ordinary assignments or conditional jumps.) If we define an equivalence relation between steps that only takes into consideration the coincidence on live variables, it should be clear that two executions starting from different but equivalent states always reach equivalent intermediate states. Stated in other words, the domain of ghost variables does not interfere with the domain of standard variables and, hence, it is safe to slice them out of the standard RTL programs.

Discussion. We can avoid introducing ghost variables if we are able to remove dead variables from assertions. However, it is not trivial to determine whether a subassertion can be removed from an invariant. Consider for instance the following example:

$$\begin{array}{rl} & y := 3 \\ & x := y \\ L_{oop}: & \{\phi\} \\ & b\ ?\ L_{out} \\ & x := x + 1 \\ & y := 3,\ L_{oop} \\ L_{out}: & \{x \geq 0\} \end{array}$$

where ϕ is $x \geq 3 \wedge y \geq 3$. Clearly, the assignment $y := 3$ within the loop and the subassertion $y \geq 3$ may sliced out from the program. However, in other examples, it may be the case that the subassertion refers both to dead and live variables, and hence cannot be removed. Consider the example above but suppose now that ϕ is defined as $x \geq y \wedge y \geq 3$. In this case, the condition $y \geq 3$ appearing in the invariant is necessary to prove the invariance of $x \geq y$. Thus, ϕ may not be simplified.

EXAMPLE 4.4. *Taking as input the short piece of code from the introduction, the result of the transformation is:*

$$\begin{array}{l} \{\mathsf{true}\} \\ \mathsf{set}\ \hat{r_1} := 1 \\ \{\hat{r_1} = 1\} \\ r_2 := 1 \\ \{r_2 = 1\} \end{array}$$

4.7 Unreachable Code Elimination

4.7.1 *Description.* Unreachable code elimination aims at removing instructions that are never executed.

The optimization described here simply identifies the connected graph of nodes that can be reached starting from the initial node pointed by L_{sp}, and removes the remaining nodes. The existence of unreachable program points may be originated, for instance, by application of other program optimizations.

We are not considering in this section the detection and removal of redundant conditional branches. Removing redundant conditional branches can be performed in a previous phase, in which the analysis is required to detect the validity of the branching condition. For example in

$$\begin{array}{rl@{\qquad\qquad}rl}
 & r_1 := r_2 & & r_1 := r_2 \\
 & r_1 = r_2 \ ?\ L_t & & \texttt{nop},\ L_t \\
 & [S] & & [S] \\
L_t : & \ldots & L_t : & \ldots
\end{array}$$

the analysis may infer that the fact $r_1 = r_2$ is valid immediately before the conditional branch and, in addition, in order to translate the certificate, this condition must be proved correct. Next, as is common with several optimizations, the function T_L^{ins} is defined and the proof of the analysis is merged with the original proof to translate the certificate. Notice that the fragment of code $[S]$ is not removed since it may be reachable by some other instructions, this transformation is left to a different decision procedure: unreachable code elimination.

The selection of reachable nodes is a straightforward process and translating the certificate does not represent any difficulty.

4.7.2 *Certificate translation.* Our definition of VCGen is such that proof obligations are totally independent from unreachable nodes.

Its clear that for any label L reachable from L_{sp}, $\mathsf{wp}_f(L) = \mathsf{wp}_{f_{|R}}(L)$, where $f_{|R}$ is f with its graph code restricted to the set R of reachable labels. Notice that for the preservation of certificates we require the VCGen to consider only assertions on reachable labels, as is the case in this paper. Otherwise, if the VCGen considers also annotations on unreachable labels, the set of proof obligations of the transformed program is a subset of the proof obligations of the initial program.

4.8 Register Allocation

4.8.1 *Description.* Register allocation is a code generation phase that intends to minimize register usage by storing in a single real machine register two or more temporary registers. That is possible, for instance, for those registers whose live ranges are disjoint (namely nonoverlapping registers).

We restrict our attention to a certificate translator for node coalescing, abstracting from the complexity of determining the mapping between intermediate and final registers. For simplicity, we refrain from considering memory spills. The result of the analysis is represented as a mapping σ with type $\mathcal{R} \mapsto \mathcal{R}$. The code transformation rewrites each instruction and assertion by applying the substitution σ given by the analysis, that maps temporary registers of the intermediate RTL language to a potentially shared register in the object language.

To ensure correctness of the analysis we require that applying the transformation induced by this substitution σ is semantics preserving. More precisely, the following property is desired over the result of the analysis: for any registers r_1 and r_2, if $\sigma r_1 = \sigma r_2$ then the live ranges of r_1 and r_2 are disjoint. In addition, we require that there are no assignments to dead registers. This condition can be fulfilled by applying ghost variable introduction in a previous phase.

4.8.2 *Certificate translation.* If the substitution σ returned by the analysis is correct, i.e. two overlapping registers r_1 and r_2 are not mapped to the same register, then the proof obligations of the original and optimized program coincide up to variable renaming.

LEMMA 4.2. *Suppose we have a function f, such that for any register r and any label L, $\mathcal{L}(L,r) \neq \top_\phi$ (all registers in assertions are live). Assume also that there are no assignments to dead variables. If $\overline{f}$ is the result of applying register allocation, then for any label L*

$$\mathsf{wp}_{\overline{f}}(L) = \mathsf{wp}_f(L)\sigma$$

where σ represents the mapping that joins pairs of nonoverlapping registers.

PROOF.

— *case* $f[L] = (\phi,\ \mathsf{ins})$
By definition of wp *and the register replacement,* $\mathsf{wp}_{\overline{f}}(L)$ *is equal to* $\phi\sigma$*, which is equal to* $\mathsf{wp}_f(L)\sigma$

— *case* $f[L] = \mathtt{nop},\ L'$
In this case $\mathsf{wp}_{\overline{f}}(L)$ *is equal to* $\mathsf{wp}_{\overline{f}}(L')$*, and by inductive hypothesis is equal to* $\mathsf{wp}_f(L')\sigma$*, and then to* $\mathsf{wp}_f(L)\sigma$*.*

— *case* $f[L] = \mathtt{return}\ r$
$\mathsf{wp}_{\overline{f}}(L)$ *is* $\mathsf{post}(f)[{}^{\sigma r}\!/\!_{\mathsf{res}}]$ *by definition, but it is clear that this is also equal to* $\mathsf{post}(f)[{}^{r}\!/\!_{\mathsf{res}}]\sigma$*. Which by definition is* $\mathsf{wp}_f(L)\sigma$*.*

— *case* $f[L] = r_d := g(\vec{r}),\ L'$
By definition of wp*, we have that* $\mathsf{wp}_{\overline{f}}(L)$ *is equal to*

$$\mathsf{pre}(f)[{}^{\sigma\vec{r}}\!/\!_{\vec{r_g}}] \wedge (\forall \mathsf{res}, \mathsf{post}(f)[{}^{\sigma\vec{r}}\!/\!_{\vec{r_g^*}}] \Rightarrow \mathsf{wp}_{\overline{f}}(L')[{}^{\mathsf{res}}\!/\!_{\sigma r_d}])$$

Since σ is the identity for variables occurring in $\mathsf{pre}(f)$ *or* $\mathsf{post}(f)$ *we have*

$$(\mathsf{pre}(f)[{}^{\vec{r}}\!/\!_{\vec{r_g}}])\sigma \wedge (\forall \mathsf{res}, (\mathsf{post}(f)[{}^{\vec{r}}\!/\!_{\vec{r_g^*}}])\sigma \Rightarrow \mathsf{wp}_{\overline{f}}(L')[{}^{\mathsf{res}}\!/\!_{\sigma r_d}])$$

and applying inductive hypothesis on $\mathsf{wp}_{\overline{f}}(L')$ *we get*

$$(\mathsf{pre}(f)[{}^{\vec{r}}\!/\!_{\vec{r_g}}])\sigma \wedge (\forall \mathsf{res}, (\mathsf{post}(f)[{}^{\vec{r}}\!/\!_{\vec{r_g^*}}])\sigma \Rightarrow \mathsf{wp}_f(L')\sigma[{}^{\mathsf{res}}\!/\!_{\sigma r_d}])$$

finally, since $\sigma\mathsf{res} = \mathsf{res}$*, it is clear that the last expression is equal to*

$$(\mathsf{pre}(f)[{}^{\vec{r}}\!/\!_{\vec{r_g}}])\sigma \wedge (\forall \mathsf{res}, (\mathsf{post}(f)[{}^{\vec{r}}\!/\!_{\vec{r_g^*}}])\sigma \Rightarrow (\mathsf{wp}_f(L')[{}^{\mathsf{res}}\!/\!_{r_d}])\sigma)$$

that is exactly $\mathsf{wp}_f(L)\sigma$

— *case* $f[L] = \mathsf{cmp}\ ?\ L_t : L_f$
In this case $\mathsf{wp}_{\overline{f}}(L)$ *is* $\mathsf{cmp}\sigma \Rightarrow \mathsf{wp}_{\overline{f}}(L_t) \wedge \neg\mathsf{cmp}\sigma \Rightarrow \mathsf{wp}_{\overline{f}}(L_f)$ *which by inductive hypothesis is equal to* $\mathsf{cmp}\sigma \Rightarrow \mathsf{wp}_f(L_t)\sigma \wedge \neg\mathsf{cmp}\sigma \Rightarrow \mathsf{wp}_f(L_f)\sigma$*. But this is in fact* $\mathsf{wp}_f(L)\sigma$ *by definition of* wp*.*

— *case* $f[L] = r_d := \mathsf{op},\ L'$
$\mathsf{wp}_{\overline{f}}(L)$ *is by definition of* wp *and the transformation,* $\mathsf{wp}_{\overline{f}}(L')[{}^{\mathsf{op}\sigma}\!/\!_{\sigma r_d}]$*. By inductive hypothesis, this is also equal to* $\mathsf{wp}_f(L')\sigma[{}^{\mathsf{op}\sigma}\!/\!_{\sigma r_d}]$*. If we show that for any variable r in* $\mathsf{wp}_f(L')$*,* $[{}^{\mathsf{op}\sigma}\!/\!_{\sigma r_d}](\sigma r)$ *is equal to* $\sigma([{}^{\mathsf{op}}\!/\!_{r_d}]r)$ *then* $\mathsf{wp}_f(L')\sigma[{}^{\mathsf{op}\sigma}\!/\!_{\sigma r_d}]$ *is equal to* $\mathsf{wp}_f(L')[{}^{\mathsf{op}}\!/\!_{r_d}]\sigma$*, and that is what we want to prove. To prove the equality between the two mappings, suppose first that $r = r_d$, in this case* $[{}^{\mathsf{op}\sigma}\!/\!_{\sigma r_d}](\sigma r)$ *is* $\mathsf{op}\sigma$ *and is clearly the same for* $\sigma([{}^{\mathsf{op}}\!/\!_{r_d}]r)$*. In case $r \neq r_d$, σr must be clearly different to σr_d, as r_d is live in L' and if $r \in FV(\mathsf{wp}_f)(L)$ then r is also live in L'. Under this assumption, both sides of the equation are equal to σr.*

□

$$\begin{array}{lll}
\overline{f}[L_{call}] & \triangleq & y := x,\ L_{call'} \\
\overline{f}[L_{call'}] & \triangleq & (\mathsf{pre}(g) \wedge Q,\ \mathtt{nop},\ L_{sp_g}) \\
\overline{f}[L] & \triangleq & \llbracket g[L] \rrbracket \quad \text{if } L \text{ is in } G_g \text{ domain} \\
\overline{f}[L_{ret'}] & \triangleq & \mathsf{post}(g)[{}^{r_d}\!/\!_{\mathsf{res}}] \wedge Q, \mathtt{nop},\ L_{ret} \\
\text{where:} & & \\
\llbracket (\phi,\ \mathsf{ins}) \rrbracket & \triangleq & (\phi \wedge Q,\ \llbracket \mathsf{ins} \rrbracket) \\
\llbracket \mathtt{return}\ r \rrbracket & \triangleq & r_d := r,\ L_{ret'} \\
\llbracket \mathsf{ins} \rrbracket & \triangleq & \mathsf{ins} \qquad \text{if } \mathsf{ins} \neq \mathtt{return}
\end{array}$$

Fig. 16. Function inlining transformation

Conclusion. As with dead register elimination, the proof transformation is just a renaming using the operator subst to the original certificates, as specified by the substitution function σ.

4.9 Function Inlining

4.9.1 *Description.* An immediate motivation of function inlining is reducing the overhead of the function call. However, its main purpose is to generate new optimization opportunities.

We do not consider here profitability issues, since it is not the purpose of this presentation. Indeed, even when the transformation reduces the call overhead and gives rise to new optimization opportunities, it is possibly undesirable since it may increase the code size and the number of registers in use. The analysis that follows is restricted to translating the certificate and does not focus on its profitability.

Suppose we have a function call $r_d := g(\vec{x}),\ L_{ret}$ as the instruction $f[L_{call}]$. The transformation consists of replacing this statement with an assignment to function g's formal parameters, followed by the body of the function g where every `return` instruction is replaced by a corresponding assignment to the register r_d. In this section we assume for simplicity that the set of registers used by g is disjoint from those of the function where it will be inlined. Similarly, for notational convenience, we assume that the functions under consideration do not share program labels, and that the function g has a unique parameter y.

The transformation is depicted in Figure 16, where Q stands for the assertion $\forall \mathsf{res}.\ \mathsf{post}(g)[{}^{x}\!/\!_{y^*}] \Rightarrow \mathsf{wp}_f(L_{ret})[{}^{\mathsf{res}}\!/\!_{r_d}]$. The graph G_g for function g is inserted, with small changes, in replacement of the function call, and appropriate assignments are inserted before the function call. Notice that not only the return instructions are modified, but also the assertions, which are augmented with conditions (Q) that must be propagated through the complete set of instructions until the end, at label L_{ret}.

4.9.2 *Certificate translation.* The wp function is defined for the function call case propagating the conditions that must be satisfied when the function returns. The problem with function inlining is that this propagation is lost, when inserting an arbitrarily big piece of code that may contain assertions. For this reason and for the purpose of certificate generation, assertions are reinforced with the condition that we want to propagate (Q).

The assertion Q has the desired property that it cannot be modified by any instruction in the body of function g, assuming the disjointness of the set of local registers. In consequence, it is not difficult to prove in this case that the following property is satisfied:

$$\forall L \in \text{ domain of } G_g.\ \mathsf{wp}^{\mathtt{id}}_{g}(L) \wedge Q \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$$

Notice that this property is similar to those we get when using an auxiliary function $f_{\mathcal{A}}$ to prove the results of the analysis, but in this case a certifying analyzer is not necessary since Q is always trivially preserved. Another property that is satisfied is

$$\forall L \in \text{ domain of } G_f.\ \mathsf{wp}_{f}(L) = \mathsf{wp}_{\overline{f}}(L)$$

From these two results and the definition of the transformation, we can reconstruct a new proof for the modified function $\overline{f}$. For every label L in the domain of G_f such that $f[L] = (\phi,\ \mathsf{ins})$, we have that $\overline{f}[L] = (\phi,\ \mathsf{ins})$ and also that $\mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L) = \mathsf{wp}^{\mathtt{id}}_{f}(L)$, so certificates are simply preserved in this range. For labels in the domain of G_g, if $g[L] = (\phi,\ \mathsf{ins})$ then $\overline{f}[L]$ is defined as $(\phi \wedge Q,\ \mathsf{ins})$, and using the proof for $\mathsf{wp}^{\mathtt{id}}_{g}(L) \wedge Q \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$, and the original proof for $\phi \Rightarrow \mathsf{wp}^{\mathtt{id}}_{g}(L)$ we get a certificate for $\phi \wedge Q \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$.

It remains to see the cases for the assertions introduced at labels $L_{ret'}$ and $L_{call'}$. In the latter, the certificate corresponding to the proof obligation related to the precondition of g is used. The former proof obligation is clearly provable as well.

Proof of auxiliary property.

Proof. We sketch here a proof for the properties stated above. The proof is performed by simultaneous induction with the order relation induced by the definition of *well-annotated* programs, for the following goals:

$$\forall L \in \text{ domain of } G_g.\ \mathsf{wp}_{g}(L) \wedge Q \Rightarrow \mathsf{wp}_{\overline{f}}(L)$$

and

$$\forall L \in \text{ domain of } G_g.\ \mathsf{wp}^{\mathtt{id}}_{g}(L) \wedge Q \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)\ .$$

— In case $g[L] = (\phi,\ \mathsf{ins})$ the first property is satisfied by definition and the second one can be proved by case analysis in ins.

— If $g[L] = r_1 := r_2,\ L'$ then, under the hypothesis that r_1 is not a free variable in Q, we get from the inductive hypothesis $\mathsf{wp}_{g}(L') \wedge Q \Rightarrow \mathsf{wp}_{\overline{f}}(L')$, a proof for $\mathsf{wp}^{\mathtt{id}}_{g}(L) \wedge Q \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$ and $\mathsf{wp}_{g}(L) \wedge Q \Rightarrow \mathsf{wp}_{\overline{f}}(L)$.

— If $g[L] = b\ ?\ L_t : L_f$ then we have to prove $\mathsf{wp}_{\overline{f}}(L)$ with the hypothesis $(b \Rightarrow \mathsf{wp}_{g}(L_t) \wedge \neg b \Rightarrow \mathsf{wp}_{g}(L_f))$ and Q. By inductive hypothesis we have that $\mathsf{wp}_{g}(L_t) \wedge Q \Rightarrow \mathsf{wp}_{\overline{f}}(L_t)$ and that $\mathsf{wp}_{g}(L_f) \wedge Q \Rightarrow \mathsf{wp}_{\overline{f}}(L_f)$. Hence, we can construct a proof for $b \Rightarrow \mathsf{wp}_{\overline{f}}(L_t)$ and $\neg b \Rightarrow \mathsf{wp}_{\overline{f}}(L_f)$.

— In case $g[L] = \mathtt{return}\ r$ then $\mathsf{wp}_{\overline{f}}(L)$ is equal to $\mathsf{post}(g)[{}^{r}\!/_{\mathsf{res}}] \wedge Q$. And, since $\mathsf{wp}_{g}(L) = \mathsf{post}(g)[{}^{r}\!/_{\mathsf{res}}]$, the consequence is clearly satisfied.

— Since any other instruction is similar and does not represent interesting difficulties, a special treatment is not deserved.

□

4.10 Summary

In Section 3 we have showed that compiling programs from a high-level language to an RTL representation preserves proof obligations as long as no optimization is performed.

In this section we have studied several standard compiler optimizations, and under each particular case, we have showed that proof obligations may be modified and, thus, that original certificates may not be reused. To solve this difficulty, we have proposed a variety of techniques, some of them based on the existence of a certifying analyzer and some others solved by ad-hoc techniques. Optimizations that rely on a certifying analyzer ranges from constant propagation or copy propagation, to common subexpression elimination or a loop optimization like induction variable strength reduction.

When dealing with dead register elimination we have showed a difficulty that arises when dead registers occur on invariants and we have proposed a transformation on both the program and the specifications called ghost variable introduction. Finally, we prove that, after applying this transformation, certificate translation for the modified proof obligations is quite simple.

We have shown that unreachable code elimination is trivial, and we have also proposed particular techniques to deal with function inlining and register allocation. The latter is in fact a simplified version of the standard phase of code generation, and relies on the simplifying assumption that dead variable elimination has already been performed.

5. OTHER COMPILATION TECHNIQUES AND RELATED WORK

The purpose of this section is to position our work w.r.t. recent developments in compilation, such as certified compilers, translation validation, certifying compilation and provable compilation through sound elementary rules. Furthermore, we also position our work with respect to related work.

5.1 Certifying Compilation

Certifying compilation is an extension of a standard compiler that automatically generates a proof that the compiled code satisfies some specific properties. A certifying compiler is built from three main components: A preliminary step consists of propagating basic information, such as the result of type inference, from the source level to the compiled program. This is in some way convenient since a higher level of abstraction is more adequate for the inference of particular properties. However, the goal is to verify properties on the low-level side, with a more concrete execution environment. Thus, it is not always convenient to limit the analysis over the source program, since essential information may be lost because of this abstraction.

Another step consists of inferring loop invariants, possible reusing information inferred about the source program if preserved by the compilation. Notice that requiring automaticity on loop invariance inference is a significant obstacle that restricts the complexity of properties that can be considered. Finally a special purpose VCGen is applied to the result of the compilation to generate a set of proof obligations, the proof of which ensure that the executable code adheres to some safety policies. To complete the process, these proof obligations are discharged by an automatic theorem prover.

Automatic generation of certificates comes at a cost. As said before, required properties must be restricted to sufficiently simple safety policies. But in addition, a certifying analyzer may fail to generate a certificate for the output code. However, it does not affect the soundness of the approach.

In addition to generate a certificate ensuring the correctness of the output code, certifying compilers may rely on a static analysis phase to remove unnecessary runtime checks and, thus, improving performance.

The Touchstone compiler [Necula and Lee 1998] is a notable example of certifying compiler, which generates type-safety certificates for a fragment of C. In Chapter 6 of his thesis [Necula 1998], Necula studies the impact of a particular set of standard program optimizations on certifying compilation, including some of the optimizations considered in this paper. For each optimization, an informal analysis is made, indicating whether the transformation requires reinforcing the program invariants, or whether the transformation does not change proof obligations. For a particular set of sufficiently simple proof obligations, Necula shows that if the VCGen propagates proof obligations backwards, then the verification conditions are preserved. However, he shows that it is not the case with some more sophisticated transformations like induction variable strength reduction and redundant conditional elimination. The former optimization is a clear example where it becomes necessary to strengthen invariants in order to keep proof obligations provable. Furthermore, Necula shows that in both optimizations the associated program certificate must be modified. While we perform a systematic and implicitly defined transformation of the certificate, Necula relies on the capability of a theorem prover, possibly modulo user-provided hints, to discharge the modified proof obligation. Even if delegating this task to a powerful theorem prover may be feasible for sufficiently simple safety properties, it is not clear how it scales up to arbitrary functional properties.

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. Second, we assume that the program comes with its annotations and certificate, and we have not only to strengthen the annotations, but also to transform the certificate. This is the main difficulty with respect to Necula's work: 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.

5.2 Certified Compilers

Compiler correctness [Guttman and Wand 1995] aims at showing that a compiler preserves the semantics of programs. Traditionally, semantics preservation is understood as the preservation of the input/output behavior of programs; thus most compiler correctness statements are of the form: if running a program p on an initial configuration c returns some final result v, then running the corresponding compiled program $[\![p]\!]$ on the corresponding initial configuration c' shall also return the same final result v.

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. [Bertot et al. 2004; Leroy 2006b; Blazy et al. 2006; Strecker 2002], advocates the use of a proof assistant for machine-checking compiler correctness results. Certified compilation involves modeling formally the source and target languages, their semantics, and providing an executable definition $[\![.]\!]$ of the compiler. Then the correctness statement is expressed in the language of the proof assistant, and proved using the logic of the proof assistant. In proof assistants that support the notion of proof object, the task of turning $[\![.]\!]$ into a certified compiler amounts to build a term H of type:

$$\forall p : \mathsf{Program}, \forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ \langle p, c\rangle \Downarrow v \Longrightarrow \langle [\![p]\!], c\rangle \Downarrow v$$

Certified compilation is not motivated by mobile code, and has not been exploited in the context of PCC architectures. In fact, certified compilation is particularly relevant for the critical software industry, where the code producers and the code consumers belong to the same entity, or trust each other. Nevertheless, it is possible to construct certificate translators from certified compilers, as explained below and suggested independently by Leroy [Leroy 2006b].

Under suitable conditions (evaluation is deterministic, compilation preserves non-termination and programs do not get stuck), one can prove from H that the compiler reflects semantics, i.e. one can build a proof object H' that establishes the dual of preservation of semantics:

$$\forall p : \mathsf{Program}, \forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ \langle [\![p]\!], c\rangle \Downarrow v \Rightarrow \langle p, c\rangle \Downarrow v$$

This proof object may be exploited to transfer evidence from source code programs to compiled programs. Indeed, assume that we want to prove the following property for some program p:

$$\forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ \langle [\![p]\!], c\rangle \Downarrow v \Rightarrow R(c, v)$$

where $R(c, v)$ establishes a formal relation between input configurations and results. Then, if we have constructed the proof object cert_p for

$$\forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ \langle p, c\rangle \Downarrow v \Rightarrow R(c, v)$$

one can build the proof object $\mathsf{cert}_{[\![p]\!]}$ for

$$\begin{array}{l}\lambda c : \mathsf{Config}.\lambda v : \mathsf{Res}.\lambda H_{\mathrm{eval}} : (\langle [\![p]\!], c\rangle \Downarrow v).\ \mathsf{cert}_p(H'\ p\ c\ H_{\mathrm{eval}})\\ \quad : \forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ (\langle [\![p]\!], c\rangle \Downarrow v) \Rightarrow R(c, v)\end{array}$$

Thus it is in principle possible to build certificate translators from certified compilers. There are however some drawbacks:

— the certificate $\mathsf{cert}_{[\![p]\!]}$ of the compiled program $[\![p]\!]$ encapsulates the definition and correctness proof of the compiler, as well as the source code and its certificate. Hence, the certificate $\mathsf{cert}_{[\![p]\!]}$ is very large, and costly to check. Leroy [Leroy 2006b] has suggested that partial evaluation could be used to eliminate much of the proof of correctness of the compiler from the certificate, but the applicability of the method has not been demonstrated;

— traditionally, certified compilers are shown to preserve the input/output behavior of programs, thus the approach rules out properties that are expressed with intermediate assertions or ghost variables. Unfortunately, many interesting properties of programs must be specified using assertions or ghost variables. Thus the approach is restrictive. Leroy [Leroy 2006a] has explored means to extend the scope of compiler correctness results beyond input/output behaviors, but his work does not cover yet preservation of intermediate assertions.

Compcert [Leroy 2006b; Blazy et al. 2006] is a notable example of a certified moderately optimizing compiler. The Compcert compiler generates PowerPC code for a significant subset of the C language, and performs several standard optimizations such as constant propagation, common subexpression elimination, register allocation and branch tunneling. The compiler is almost completely specified in the Coq proof assistant, and is accompanied with a proof, in Coq, of a set of lemmas stating that the generated assembly code is semantically equivalent to the input source program. The compiler is executable and can be translated to Ocaml with the extraction mechanism provided by the Coq tool; the performance of the extracted compiler is very reasonable.

5.3 Translation Validation

Translation validation is an approach to verify program transformation by checking, for each individual translation, a correspondence between the output and input programs. It consists mainly of a common model to abstract the semantics of the input and output programs, and an automatic verification procedure that checks whether the semantics of the generated program simulates the semantics of the source program. Since the validation phase is independent of the compiler, i.e. there is no need to specify and verify the compiler, it does not constrain the evolution of the compiler.

Verifying correctness of the compilation consists of giving, for each input program, a proof that the compiled code preserves the semantics of the original one. The difference with verifying the compiler is that instead of establishing once and for all that every output code will be equivalent to the source program, the compiler provides a proof of this equivalence for each compilation result.

Initial work on translation validation [Pnueli et al. 1998] proposed a tool to verify the correctness for a small set of proof obligations on a restricted subset of target programs. Necula [Necula 2000] extended this work by implementing a translation validation infrastructure in the context of a GNU C compiler. It handles the intermediate phases of a realistic compiler, but the set of optimizations under consideration are reduced to structure preserving transformations. Further results on translation validation have presented a more comprehensive set of program optimizations [Barrett et al. 2005; Zuck et al. 2002]. These include simple structure preserving optimizations as well as non structure-preserving transformations such as loop inversion, loop unrolling, loop fusion and strength reduction. A recent publication [Tristan and Leroy 2008] presents the development of a formally verified translation validator for instruction scheduling optimizations, specified and verified in the Coq proof assistant.

As discussed in Section 5.2, it is also possible to construct certificate translators from translation validation. Again, assuming we have a notion of proof object, instead of a proof term H such that for any program p:

$$H : \forall p : \mathsf{Program}, \forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ \langle p, c\rangle \Downarrow v \Longrightarrow \langle [\![p]\!], c\rangle \Downarrow v$$

we will have, for every successfully compiled program p, a proof term H_p such that

$$H_p : \forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ \langle p, c\rangle \Downarrow v \Longrightarrow \langle [\![p]\!], c\rangle \Downarrow v$$

The same reasoning explained above with respect to certified compilers permits to build certificate translators from translation validation. This approach has the advantage that a certificate does not encapsulate a definition of the compiler. However, since the output program is not verified independently of the source program, the latter must be delivered as a component of the certificate.

5.4 Provable Optimizations through Sound Elementary Rules

Rhodium [Lerner et al. 2005] is a special purpose framework aimed at specifying and proving the correctness of program analyses and optimizations. To this end, Rhodium relies on a domain-specific language for specifying the analysis and transformation by means of local rules. A set of statements define the abstract domain of the analysis, called dataflow facts, and specifies its meaning as a predicate on the execution state. Transfer functions of the analysis are specified by the so called propagation rules, that declare the generation or propagation of dataflow facts through single edges. They are specified for a generic instantiation of variables and are local in the sense that they relate only adjacent program nodes. Analyzing the correctness of each local rule is restricted to verifying it with respect to a single execution step. Rhodium generates, for each local propagation rule, a proof obligation in terms of its semantic interpretation, and then submits them to an automatic prover that attempts to discharge them. Given the validity of every local elementary rule, the validity of the entire analysis follows by a common property of the framework (proved once, by hand, and instantiated for any analysis.)

Finally, a set of transformation rules specifies how this inferred data-flow facts are used to trigger program transformations. In the same spirit, giving a formal proof for each elementary transformation rule is sufficient to guarantee the correctness of the whole transformation.

5.5 Proof Transforming Compilation

The Spec# project [Barnett et al. 2005; Leino and Schulte 2004; Leino 2006] 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. Pavlova and Burdy have followed a similar line of work [Burdy and Pavlova 2006] to define a Bytecode Modeling Language (BML), and a VCGen for annotated bytecode programs. Annotations of the Java Modeling Language are translated into BML, and the generated proof obligations are sent to an automatic theorem prover. They present a partial formalization of the relation of verification conditions at source and bytecode levels, but they do not consider proof transformations. In a recent work, Barthe, Gregoire and Pavlova [Barthe et al. 2008] establish the preservation of proof obligations from source Java programs to compiled code, obtained by nonoptimizing compilation.

In a similar spirit, Bannwart and Müller [Bannwart and Müller 2005], 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 nonoptimizing compilation. Müller and Nordio [Müller and Nordio 2007] have developed a proof transforming procedure for a Java to Java Bytecode compiler, in particular, dealing with abrupt termination. They have explained the complications of including a subset of Java with `try-catch`, `try-finally` and `break` statements, and showed how they may be handled. More recently, Nordio, Müeller and Meyer have formalized [Nordio et al. 2008; Nordio et al. 2008], using the Isabelle proof assistant, a proof transforming procedure from a subset of the Eiffel programming language to Microsoft's Common Intermediate Language.

Furthermore, there are relevant results on transferring evidence from source programs to compiled programs in scenarios that use alternative technologies for establishing program correctness. These results include type-preserving compilation [Barthe et al. 2006; Tarditi et al. 1996], and Rival's method [Rival 2004] to translate program invariants generated at source level using abstract interpretation techniques in order to verify safety properties.

Developing further the approach of describing data-flow analysis as type judgments [Laud et al. 2006], Saabas and Uustalu [Saabas and Uustalu 2008] propose to extend these type-based methods to describe also program transformations. They illustrate the feasibility of the method by explaining in detail two particular transformations: Common subexpression elimination and Dead Variable elimination. They have demonstrated the correctness of both transformations, by derivability of Hoare logic proofs, but also showed a constructive mechanism to transform a Hoare proof of the original program to a Hoare proof of the transformed program. However, it is not clear from the paper how the approach extends to arbitrary program transformations.

Another instance of proof preserving compilation is the work of Shao et al. [Shao et al. 2005]. They define a framework to reason and certify programs beyond simple safety policies but still maintaining properties at a decidable level. In addition, they show that certificates are preserved after applying CPS and closure conversion.

5.6 Proof Producing Analysis

One key component of a certificate translator is the certifying analyzer. For each standard optimization we have shown that defining a certifying analyzer is straightforward due to the simplicity of the generated proof obligations. It is possible to define a generic certifying analyzer under mild assumptions on the relation between the analysis and the verification infrastructure. This is a substantial improvement if we plan to extend certificate translation to arbitrary semantics preserving transformations.

The possibility of defining such certifying analyzers has already been studied independently by Wildmoser, Chaieb and Nipkow [Wildmoser et al. 2005] in the context of a bytecode language and by Seo, Yang and Yi [Seo et al. 2003] in the context of a simple imperative language. Seo et al. propose an algorithm that automatically constructs safety proofs in Hoare logic from abstract interpretation results.

6. CONCLUDING REMARKS

Certificate translation provides a means to transfer the benefits of source code verification to code consumers using PCC architectures, extending the scope of PCC to arbitrarily complex policies.

We have introduced the principles of certificate translation and shown that certificate translators exist for many common optimizations. For concreteness, we focused on a particular programming language and set of optimizations. In a subsequent work [Barthe and Kunz 2008], we use the framework of abstract interpretation to give a set of sufficient conditions for the existence of certifying analyzers and certificate translators. This latter work shows that the concept of certificate translation is potentially applicable in a wide range of settings. To complement these theoretical underpinnings, we would like to illustrate the practicality of certificate translation through prototype implementations and case studies. On a more general perspective, we would also like to consider applications of certificate translation in the component-based development of security-sensitive software.

REFERENCES

Bannwart, F. Y. and Müller, P. 2005. A program logic for bytecode. *Electronic Notes in Theoretical Computer Science 141*, 255–273.

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*. Lecture Notes in Computer Science, vol. 4111. Springer-Verlag.

Barnett, M., Leino, K. R. M., and Schulte, W. 2005. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds. Lecture Notes in Computer Science, vol. 3362. Springer-Verlag, 151–171.

Barrett, C. W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., and Zuck, L. D. 2005. Tvoc: A translation validator for optimizing compilers. In *CAV*, K. Etessami and S. K. Rajamani, Eds. Number 3576 in Lecture Notes in Computer Science. Springer-Verlag, 291–295.

Barthe, G., Burdy, L., Charles, J., Grégoire, B., Huisman, M., Lanet, J.-L., Pavlova, M., and Requet, A. 2007. JACK: A tool for validation of security and behaviour of Java applications. In *Formal Methods for Components and Objects: Revised Lectures from the 5th International Symposium FMCO 2006*. Number 4709 in Lecture Notes in Computer Science. Springer-Verlag, 152–174.

Barthe, G., Grégoire, B., Kunz, C., and Rezk, T. 2006. Certificate translation for optimizing compilers. In *Static Analysis Symposium*, K. Yi, Ed. Number 4134 in Lecture Notes in Computer Science. Springer-Verlag, 301–317.

Barthe, G., Grégoire, B., and Pavlova, M. 2008. Preservation of proof obligations for Java. In *International Joint Conference on Automated Reasoning*. Lecture Notes in Computer Science. Springer-Verlag. To appear.

Barthe, G. and Kunz, C. 2008. Certificate translation in abstract interpretation. In *European Symposium on Programming*. Number 4960 in Lecture Notes in Computer Science. Springer-Verlag, 368–382.

Barthe, G., Naumann, D., and Rezk, T. 2006. Deriving an information flow checker and certifying compiler for Java. In *Symposium on Security and Privacy*. IEEE Press.

Barthe, G., T.Rezk, and Saabas, A. 2005. Proof obligations preserving compilation. In *Workshop on Formal Aspects in Security and Trust*, T. Dimitrakos, F. Martinelli, P. Ryan, and S. Schneider, Eds. Number 3866 in Lecture Notes in Computer Science. Springer-Verlag, 112–126.

Bertot, Y., Grégoire, B., and Leroy, X. 2004. A structured approach to proving compiler optimizations based on dataflow analysis. In *TYPES*, J. Filliâtre, C. Paulin-Mohring, and B. Werner, Eds. Lecture Notes in Computer Science, vol. 3839. Springer, 66–81.

Blazy, S., Dargaye, Z., and Leroy, X. 2006. Formal verification of a c compiler front-end. In *FM*, J. Misra, T. Nipkow, and E. Sekerinski, Eds. Lecture Notes in Computer Science, vol. 4085. Springer, 460–475.

Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K., and Poll, E. 2003. An overview of JML tools and applications. In *Workshop on Formal Methods for Industrial Critical Systems*. Electronic Notes in Theoretical Computer Science, vol. 80. Elsevier, 73–89.

Burdy, L. and Pavlova, M. 2006. Java bytecode specification and verification. In *Symposium on Applied Computing*. ACM Press, 1835–1839.

Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E. 2006. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects*, Springer-Verlag, Ed. Lecture Notes in Computer Science, vol. 4111. 342–363.

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

Laud, P., Uustalu, T., and Vene, V. 2006. Type systems equivalent to data-flow analyses for imperative languages. *Theoretical Computer Science 364,* 3, 292–310.

Leino, K. R. M. 2006. Specifying and verifying programs in spec#. In *Ershov Memorial Conference*, I. Virbitskaite and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 4378. Springer, 20.

Leino, K. R. M. and Schulte, W. 2004. Exception safety for c#. In *SEFM*. IEEE Computer Society, 218–227.

Lerner, S., Millstein, T., Rice, E., and Chambers, C. 2005. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 364–377.

Leroy, X. 2006a. Coinductive big-step operational semantics. In *Programming Languages and Systems: Proceedings of the 15th European Symposium on Programming, ESOP 2006*. Lecture Notes in Computer Science, vol. 3924. Springer-Verlag, 54–68.

Leroy, X. 2006b. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of Programming Languages*, J. G. Morrisett and S. L. P. Jones, Eds. ACM Press, 42–54.

Müller, P. and Nordio, M. 2007. Proof-transforming compilation of programs with abrupt termination. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*. ACM, New York, NY, USA, 39–46.

Necula, G. 1998. Compiling with proofs. Ph.D. thesis, Carnegie Mellon University. Available as Technical Report CMU-CS-98-154.

Necula, G. C. 1997. Proof-carrying code. In *Principles of Programming Languages*. ACM Press, New York, NY, USA, 106–119.

Necula, G. C. 2000. Translation validation for an optimizing compiler. *SIGPLAN Not. 35,* 5, 83–94.

Necula, G. C. and Lee, P. 1998. The design and implementation of a certifying compiler. In *Programming Languages Design and Implementation*. Vol. 33. ACM Press, New York, NY, USA, 333–344.

Nordio, M., Müller, P., and Meyer, B. 2008. Formalizing proof-transforming compilation of eiffel programs. Tech. Rep. 587, ETH Zurich.

NORDIO, M., MÜLLER, P., AND MEYER, B. 2008. Proof-transforming compilation of eiffel programs. In *TOOLS-EUROPE*, R. Paige, Ed. Lecture Notes in Business and Information Processing. Springer-Verlag.

PNUELI, A., SINGERMAN, E., AND SIEGEL, M. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, B. Steffen, Ed. Lecture Notes in Computer Science, vol. 1384. Springer-Verlag, 151–166.

RIVAL, X. 2004. Symbolic Transfer Functions-based Approaches to Certified Compilation. In *Principles of Programming Languages*. ACM Press, 1–13.

SAABAS, A. AND UUSTALU, T. 2008. Program and proof optimizations with type systems. *Journal of Logic and Algebraic Programming 77,* 1–2, 131–154.

SEO, S., YANG, H., AND YI, K. 2003. Automatic Construction of Hoare Proofs from Abstract Interpretation Results. In *Asian Programming Languages and Systems Symposium*, A. Ohori, Ed. Lecture Notes in Computer Science, vol. 2895. Springer-Verlag, 230–245.

SHAO, Z., TRIFONOV, V., SAHA, B., AND PAPASPYROU, N. 2005. A type system for certified binaries. *ACM Trans. Program. Lang. Syst. 27,* 1, 1–45.

STRECKER, M. 2002. Formal Verification of a Java Compiler in Isabelle. In *Conference on Automated Deduction*, A. Voronkov, Ed. Lecture Notes in Computer Science, vol. 2392. Springer-Verlag, London, UK, 63–77.

TARDITI, D., MORRISETT, J. G., CHENG, P., STONE, C., HARPER, R., AND LEE, P. 1996. TIL: A type-directed optimizing compiler for ML. In *Programming Languages Design and Implementation.* Association of Computing Machinery, 181–192.

TRISTAN, J. AND LEROY, X. 2008. Formal verification of translation validators: a case study on instruction scheduling optimizations. *SIGPLAN Not. 43,* 1, 17–27.

WILDMOSER, M., CHAIEB, A., AND NIPKOW, T. 2005. Bytecode analysis for proof carrying code. In *Bytecode Semantics, Verification, Analysis and Transformation*, F. Spoto, Ed. Electronic Notes in Theoretical Computer Science, vol. 141. Elsevier.

ZUCK, L. D., PNUELI, A., FANG, Y., AND GOLDBERG, B. 2002. Voc: A translation validator for optimizing compilers. *Electronic Notes in Theoretical Computer Science 65,* 2.

Received Month Year; revised Month Year; accepted Month Year