An introduction to certificate translation*

Gilles Barthe and César Kunz

IMDEA Software, Spain

Abstract. In a Proof-Carrying Code scenario, certificate generation remains a challenging problem. Typically, it is implemented as a compiler module that targets low-level executable code. Hence, since automatic, the properties under verification are limited to very simple safety policies. Discharging verification conditions automatically for arbitrarily complex properties is unfeasible. Therefore, it requires the support of tool-based interactive verification, which commonly targets high-level structured code. To connect source code verification and compiled code certification we have proposed a technique to build, from a certificate of the source program, a certificate for the result of its compilation. In this tutorial, we illustrate the principles of this technique, certificate translation, in the context of a certified quicksort algorithm. For each transformation step that defines the compiler, we explain the corresponding transformation of the certificate.

1 Introduction

Certificate translation [2, 4] is a general method that reconciles interactive verification of source programs with automated verification of compiled programs, using certificates as a means to convey evidence about program correctness. More precisely, certificate translation offers the possibility of generating certificates for complex properties of compiled programs—with the potential cost of interactive verification of source programs—and thus provides an alternative to certifying compilation, which is used in the context of Proof-Carrying Code [19] to generate automatically certificates that compiled programs respect simple policies.

Certificate translation primarily focuses on the interplay between compilation and program correctness: given a compiler $[\![.]\!]$ from a source language $\mathcal{S}$ to a target language $\mathcal{T}$, and a compiler $[\![.]\!]_{\text{spec}}$ from a specification language $\mathsf{Spec}_{\mathcal{S}}$ for source programs to a specification language $\mathsf{Spec}_{\mathcal{T}}$ for target programs, certificate translation is concerned with the following two intimately related questions:

1. for every program p and specification ϕ, does the correctness of p w.r.t. ϕ entail the correctness of $[\![p]\!]$ w.r.t. $[\![\phi]\!]_{\text{spec}}$?
2. for every program p and specification ϕ, is there a method to transform evidence of the correctness of p w.r.t. ϕ into evidence $[\![p]\!]$ w.r.t. $[\![\phi]\!]_{\text{spec}}$?

* Partially funded by the EU project MOBIUS IST-15905.

Answering these questions requires making precise the notion of program correctness, and to a lesser extent on the notion of evidence. To ensure compatibility with typical Proof-Carrying Code architectures, we base our infrastructure for verifying program correctness on generators of proof obligations (a.k.a. verification conditions) from annotated programs. On the other hand, we do not need to commit to a particular format for certificates, and assume instead the existence of a binary judgment $c :\vdash \phi$ stating that c is a certificate for ϕ, and of a set of operations for making some basic manipulations on certificates.

Thus, a program p satisfies a specification ϕ iff the set of proof obligations $\mathsf{PO}(p, \phi) = \{\phi_1, \ldots, \phi_n\}$ is provable, and evidence that p satisfies ϕ takes the form of a set of certificates $\mathsf{Cert}(p, \phi) = \{c_1, \ldots, c_n\}$ such that $c_1 :\vdash \phi_1$ and ... and $c_n :\vdash \phi_n$. Then, the problem tackled by certificate translation is to find a function $[\![\,]\!]_{\mathrm{cert}} : \forall p\ \phi,\ \mathsf{Cert}(p, \phi) \to \mathsf{Cert}([\![p]\!], [\![\phi]\!]_{\mathrm{spec}})$, i.e., a procedure that transforms a set certificates for the source program into a set of certificates for the result of the compilation.

The purpose of this tutorial is to illustrate the principles and effects of certificate translation on the example of the quicksort function. We start from an interactive proof of the quicksort function in a small imperative language with procedures and arrays; the code is given in Figure 1. We assume that the **quicksort** function is certified to satisfy the specification

$$\{\mathsf{Pre} : 0 \leq \texttt{start} \leq \texttt{end} \leq |\texttt{vec}|\}$$
$$\textbf{quicksort}(\texttt{start}, \texttt{end})$$
$$\{\mathsf{Post} : \forall k.\ \texttt{start} \leq k < \texttt{end} \Rightarrow \texttt{vec}[k] \leq \texttt{vec}[k+1]\}$$

where `vec` is a global array variable. That is, if the values held by the parameters `start` and `end` are within the bounds of the array `vec`, after the execution of **quicksort**, `vec` holds increasing values in the range [`start`, `end`].

The certificate transformation process follows the overall structure of a classical compiler, which operates on the input program in successive and independent transformation steps. For each program compilation step, we transform the specification and the certificates accordingly. An overall scheme of the compiler under consideration can be found in Figure 2, together with the corresponding certificate translation steps. In these transformation steps, the code is gradually transformed towards its final executable representation. First, the high-level structured code is transformed to a low-level intermediate program representation (RTL). In this intermediate representation, the compiler proceeds with successive optimizing transformations. The final step transforms the intermediate representation into stack-based code.

Outline. Section 2 provides an informal review of the principles of Proof Carrying Code. In Section 3, we define the source programming language and a corresponding verification framework. In Section 4, we describe the intermediate RTL program representation. In this setting, a verification framework for RTL is defined, and a short verification example is provided. In Section 5, we deal with certificate transformation along compiler phases, including non-optimizing

```
quicksort(start, end){
  if (start < end) {
    p = partition(start, end);
    quicksort(start, p);
    quicksort(p+1, end);
  }
  return;
}

swap(i, j){
  t = vec[i];
  vec[i] = vec[j];
  vec[j] = t;
  return;
}

partition(start, end){
  pivot = vec[start];
  i = start;
  j = end;
  while (i < j) {
    while (vec[i] ≤ pivot ∧ i < j)
      i++;
    while(pivot < vec[j] ∧ i < j)
      j--;
    if (i < j) swap(i, j);
  }
  swap(start, i-1);
  return i-1;
}
```

Fig. 1. Quicksort Algorithm

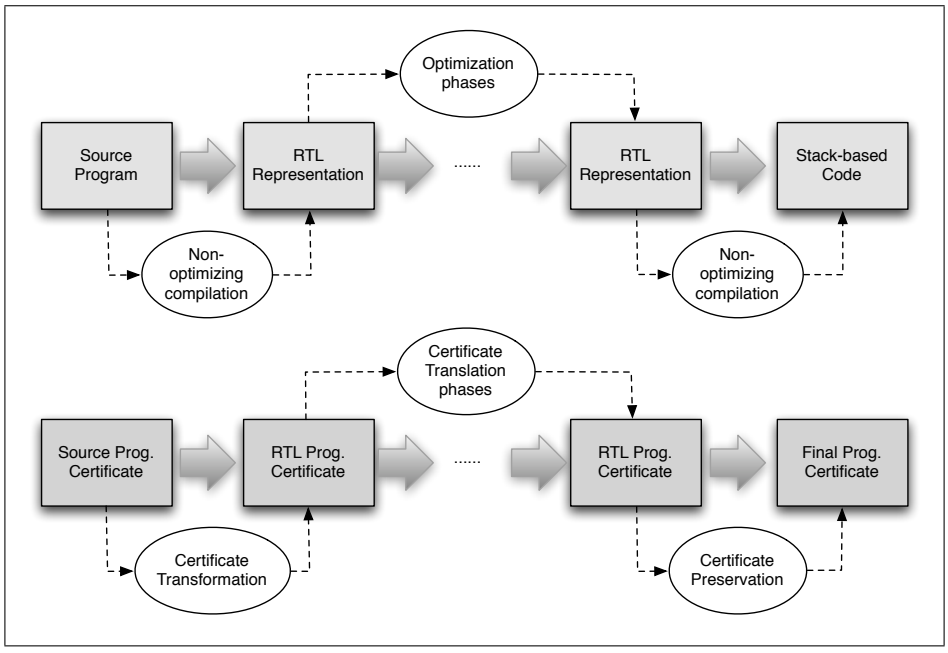


Fig. 2. Overall Compiler and Certificate Translation Phases

compilation, loop-induction strength reduction, dead variable elimination loop unrolling, and redundant conditional elimination. In Section 5.8, we show preservation of proof obligations in the generation of the final stack-based code. We conclude in Section 6.

2 A brief and informal review of Proof Carrying Code

Proof Carrying Code (PCC) [16] provides a general framework for protecting end-users against malicious mobile code. PCC promotes trust via verifiable evidence, and requires mobile code being distributed together with certificates which attest its adherence to the end-user policies. Certificates help dispensing code consumers from the high cost of proving that the code respects their policies; instead, code consumers merely have to check that the incoming certificate is a correct proof, a process that can be fully automated.

A PCC infrastructure is composed of several elements. Figure 2 shows a scheme of the client side of a PCC architecture. We briefly describe each component:

- A formal logic in which the expected behavior of the program is specified. Commonly, PCC adopts first-order or higher-order logic to both specify and verify the program.
- A verification condition generator that automatically produces a set of proof obligations for the code and its specification. The validity of the generated proof obligations ensures that the code complies with its specification.
- A formal representation of proofs, a.k.a. certificates, that provides efficiently verifiable evidence of the validity of proof obligations.
- A proof checker that verifies that the certificate does indeed establish the proof obligations.

Proof Carrying Code benefits from a number of distinctive features that make it a very appropriate basis for security architectures for global computers, and in particular for addressing the security issues highlighted above.

Proof Carrying Code is based on verification rather than trust. Indeed, Proof Carrying Code focuses on the behavior of downloaded components rather than on its origins. In particular, it does not require the existence of a global trust infrastructure (although it can be used in combination with cryptographic based trust infrastructures), for a further discussion see [1].

Proof Carrying Code is transparent for end users. While Proof Carrying Code builds upon ideas from program verification, which in its full generality requires interactive proofs, the PCC architecture does not require the code consumers to build proofs. Rather, it requires code consumers to check proofs, which is fully automatic.

Proof Carrying Code is general. The only restriction on the security policy is that it should be expressible in the formal logic, which is often very expressive.

Proof Carrying Code is flexible and configurable. The same architecture can be used for different policies. In particular, the VCGen and the proof checker are independent of the policy, while the certificate generation can in principle be adapted to different safety properties.

Proof Carrying Code does not sacrifice performance to security. PCC technology advocates for static verification, and therefore does not incur in the overhead cost inherent to dynamic techniques based on monitoring.

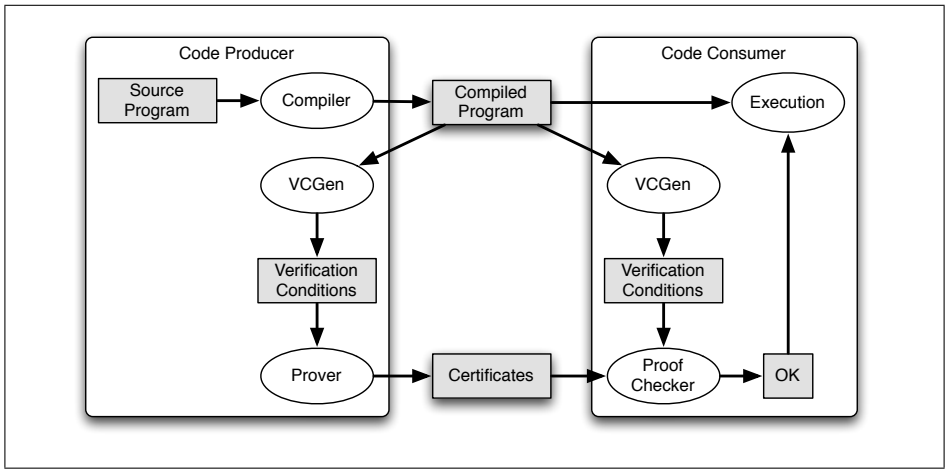


Fig. 3. PCC Scheme - Code Client Side.

3 Tool Based Source Code Verification

3.1 Programming Language Setting

In this section, we define the high-level imperative language for writing source programs. A source program is defined as a collection of procedures, each of them consisting of its formal parameters and the statement that defines its body. Expressions and statements are described in Figure 4. $\mathcal{V}$ and $\mathcal{A}$ represent the set of scalar and array variables, respectively. Most of the constructions in the grammar of the figure are standard, $\oplus$ stands for an integer operation and $a[e]$ stands for the integer value stored in the array a at position e. Statements include assignments to array structures, $a[e]{:=}e$, and procedure invocations of the form $x{:=}f(e)$. The statement **if** b **then** c stands for **if** b **then** c **else skip**.

For simplicity, we assume source programs to be well-formed, in the sense that every execution path reaches a **return** statement. The following definition formalizes this requirement:

integer expressions	$e ::= e \ \oplus \ e \mid n \mid x \mid a[e]$
boolean expressions	$b ::= \mathsf{true} \mid \mathsf{false} \mid e \bowtie e \mid b \wedge b \mid \ldots$
statements	$c ::= \mathbf{skip} \mid x{:=}e \mid a[e]{:=}e \mid c; c \mid \mathbf{return}\ e$
	$\mid f(e) \mid x{:=}f(e)$
	$\mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{while}\ b\ \mathbf{do}\ c$

Fig. 4. Source Programs

Definition 1. *We define well-formed programs as the minimum set of statements* wf *that satisfy the following conditions:*

$$\mathsf{wf}(\mathbf{return})$$
$$\mathsf{wf}(c_2) \Rightarrow \mathsf{wf}(c_1; c_2)$$
$$\mathsf{wf}(c_1) \wedge \mathsf{wf}(c_2) \Rightarrow \mathsf{wf}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)$$
$$\mathsf{wf}(c) \Rightarrow \mathsf{wf}(\mathbf{while}\ b\ \mathbf{do}\ c)$$

In this chapter, we consider scalar and array variables as allocated in separate stores. In particular, scalar variables are local to the execution of a procedure body, and array variables are global to the whole program. Let $\Sigma_{\mathcal{V}}$ and $\Sigma_{\mathcal{A}}$ represent the set of partial functions from program variables to integer values $\mathcal{V} \to \mathbb{Z}$ and from array variables to array values $\mathcal{A} \to (\mathbb{N} \rightharpoonup \mathbb{Z})$, respectively. We denote with Σ the set of elements in $\Sigma_{\mathcal{V}} \times \Sigma_{\mathcal{A}}$.

The semantics of the programming language described above is standard. It is defined in Figure 5 by a relation $\Rightarrow \subseteq (\mathsf{Prog} \times \Sigma) \times (\Sigma + \Sigma_F)$, where Σ_F denotes the set of final states composed of a final value and a final execution state: $\Sigma_F = \mathbb{Z} \times \Sigma_{\mathcal{A}}$. In the figure, σ represents an element in Σ, and $\sigma_{\mathcal{V}}$ and $\sigma_{\mathcal{A}}$ the first and second projection of the pair σ. For a scalar state $\sigma_{\mathcal{V}} \in \Sigma_{\mathcal{V}}$, scalar variable x and $n \in \mathbb{Z}$, $[\sigma_{\mathcal{V}} : x \mapsto n]$ stands for the function that maps x to n, and any other variable y to $\sigma_{\mathcal{V}}\, y$. For an array state $\sigma_{\mathcal{A}} \in \Sigma_{\mathcal{A}}$, array variable a and $b \in \mathbb{N} \to \mathbb{Z}$, $[\sigma_{\mathcal{A}} : a \mapsto b]$ stands for the function that maps a to b, and any other array variable a' to $\sigma_{\mathcal{A}}\, a'$. The expression $[x \mapsto n]$ denotes the function that maps x to n and is undefined for every other variable.

For the integer and boolean expressions e and b, $[\![e]\!]_\sigma$ and $[\![b]\!]_\sigma$ stands for their standard interpretation in the state σ. In the presence of out-of-bounds array accesses, the interpretation function is undefined, and the program execution gets stuck. We denote x_f the formal parameter of a procedure f. Since array variables are considered global to the whole program, x_f is necessarily a scalar variable.

3.2 Verification Setting

Logical verification techniques have been widely studied and used from the early 70's, pioneered by the work of Floyd [11] and Hoare [12]. There is currently a variety of program verification tools, most of them focused on high-level imperative programming languages [7, 6, 10, 15].

$$\frac{}{\langle \mathbf{skip}, \sigma\rangle \Rightarrow \sigma} \qquad \frac{[\![e]\!]_\sigma = n \in \mathbb{Z}}{\langle x{:=}e, \sigma\rangle \Rightarrow [\sigma : x \mapsto [\![e]\!]_\sigma]}$$

$$\frac{0 \leq [\![e_1]\!]_\sigma < |a|}{\langle a[e_1]{:=}e_2, \sigma\rangle \Rightarrow [\sigma : a \mapsto [a : [\![e_1]\!]_\sigma \mapsto [\![e_2]\!]_\sigma]]}$$

$$\frac{\langle c_1, \sigma\rangle \Rightarrow \sigma' \qquad \langle c_2, \sigma'\rangle \Rightarrow s}{\langle c_1; c_2, \sigma\rangle \Rightarrow s}$$

$$\frac{\langle c_1, \sigma\rangle \Rightarrow s \qquad s \in \Sigma_F}{\langle c_1; c_2, \sigma\rangle \Rightarrow s} \qquad \frac{[\![e]\!]_\sigma = n \in \mathbb{Z}}{\langle \mathbf{return}\ e, \sigma\rangle \Rightarrow \langle n, \sigma_{\mathcal{A}}\rangle}$$

$$\frac{[\![e]\!]_\sigma = n \in \mathbb{Z} \qquad c \text{ body of } f \qquad \langle c, \langle [x_f \mapsto n], \sigma_{\mathcal{A}}\rangle\rangle \Rightarrow \langle m, \sigma'_{\mathcal{A}}\rangle}{\langle x := f(e), \langle \sigma_{\mathcal{V}}, \sigma_{\mathcal{A}}\rangle\rangle \Rightarrow \langle [\sigma_{\mathcal{V}} : x \mapsto m], \sigma'_{\mathcal{A}}\rangle}$$

$$\frac{\langle c; \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \Rightarrow s}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \Rightarrow s}\ [\![b]\!]_\sigma \qquad \frac{}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \Rightarrow \sigma}\ [\![\neg b]\!]_\sigma$$

$$\frac{\langle c_1, \sigma\rangle \Rightarrow s}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle \Rightarrow s}\ [\![b]\!]_\sigma \qquad \frac{\langle c_2, \sigma\rangle \Rightarrow s}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle \Rightarrow s}\ [\![\neg b]\!]_\sigma$$

Fig. 5. Source Program Semantics

One distinctive goal of tool based verification is automating the process as much as possible. In general, a verification tool extracts from a program and its logical specification a set of first-order formulae, namely the verification conditions, that must be discharged in order to prove the program correct. Requiring the verification process to be automatic makes weakest precondition based verification preferable to using Hoare-clauses. In addition, such verification tools feed an automatic theorem prover with the verification conditions. Those verification conditions that fail to be automatically discharged must be proved interactively by the user of the verification tool.

In the rest of this section, we formalize a weakest-precondition based verification method for simple imperative programs, we prove the method sound with respect to the program semantics defined above, and we show the extraction of verification conditions in the example of the quicksort algorithm.

Specification language. As a specification language we use first-order formulae as defined in Figure 6. Most of the syntactic constructions are standard, except perhaps for the special purpose variable `res` that refers to the value returned by a procedure, and the scalar and array variables $x^\star$ and $a^\star$ that refer to the initial value of the scalar and array variables x and a, respectively. We

let $\mathcal{V}^\star$ and $\mathcal{A}^\star$ stand for the sets of variables $\{x^\star \mid x \in \mathcal{V}\}$ and $\{x^\star \mid x \in \mathcal{A}\}$, respectively.

$$\bar{e} ::= n \mid x \mid x^\star \mid a[\bar{e}] \mid a^\star[\bar{e}] \mid \bar{e} \oplus \bar{e} \mid \texttt{res}$$
$$\varphi ::= \mathsf{true} \mid \mathsf{false} \mid \bar{e} \bowtie \bar{e} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \Rightarrow \varphi \mid \forall x.\ \varphi$$

Fig. 6. Specification Language

The validity of an assertion in a particular execution state $\sigma \in \Sigma$ is standard. In particular, an assertion that contains the expression $a[e]$ is invalid in those execution states in which $a[e]$ is not well defined, i.e. in those states in which e is out of the bounds of the array a. We assume a relation $\models$ to denote that an assertion $\varphi \in \mathcal{A}$ is valid when interpreted in the state $\sigma \in \Sigma$, written $\models \sigma : \varphi$.

The specification of a procedure consists of a tuple $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$, where Pre and Post specify the procedure pre and postcondition. The verification setting only considers partial correctness, i.e., it only ensures the correctness of terminating executions. The partial function $\mathsf{annot} : \mathcal{L} \rightharpoonup \mathcal{A}$ maps any program loop at label k to the corresponding loop invariant $\mathsf{annot}(k)$. Some restrictions apply to the assertions Pre and Post. Any array variable may appear in the assertion Pre, but the only scalar variables that appear in Pre are the procedure arguments. Similarly, Post can refer to the current and initial value of any array variable, the special return variable `res`, and the initial values of the procedure arguments. The invariants specified by the partial function annot can refer to the initial and current value of any scalar and array variable, but not to the variable `res`.

For notational convenience, we associate labels $k \in \mathcal{L}$ to loop statements, denoted $\mathbf{while}^k\ b\ \mathbf{do}\ c$. In order to be able to extract verification conditions automatically, we require procedure specifications to annotate every program loop, as stated in the following definition.

Definition 2 (Well-annotated Source Program). *A procedure p with specification $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$ is well-annotated if $k \in \mathsf{dom}(\mathsf{annot})$, for every loop statement $\mathbf{while}^k\ b\ \mathbf{do}\ c$ in P. A program is well-annotated if all its procedures are well annotated.*

In the rest of the chapter, we only consider well-annotated programs.

A VCgen for source programs is defined by the set of proof obligations PO, in terms of the function WP, as shown in Figure 7. In the figure, the expression $\phi[{}^{\vec{V}}/_{\vec{V}^\star}]$ represents the result of substituting in ϕ every array and scalar variable $x^\star$ in $\mathcal{V}^\star$ or $\mathcal{A}^\star$ by x.

One desirable property of a verification framework is its soundness with respect to the program semantics. The following lemma formalizes this result:

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

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

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

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

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

$$\frac{\mathsf{WP}(c, \mathsf{annot}(k)) = \langle \phi_1, \theta \rangle \qquad \Phi \doteq \mathsf{annot}(k) \Rightarrow (b \Rightarrow \phi_1) \wedge (\neg b \Rightarrow \phi)}{\mathsf{WP}(\mathbf{while}^{k}\ b\ \mathbf{do}\ c, \phi) = \langle \mathsf{annot}(k), \{\Phi\} \cup \theta \rangle}$$

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

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

Fig. 7. Source Code VCgen Rules

Lemma 1 (Source Code VCGen Soundness). *Let the statement c be the body of the procedure p with specification* $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$. *Let σ represent an initial state that satisfies $\models \sigma : \mathsf{Pre}$, and that every proof obligation in $\mathsf{PO}(p)$ is valid. Then, every reachable final state satisfies the assertion* Post. *Formally, if $\langle c, \sigma \rangle \Rightarrow \langle n, \sigma'_{\mathcal{A}} \rangle$, then $\models \langle [\mathtt{res} \mapsto n], \sigma'_{\mathcal{A}} \rangle : \mathsf{Post}$.*

Example 1. To illustrate a verification process of a simple algorithm, consider the procedure **partition** shown in Figure 8. Figure 9, provides the specifications for the running example, including the procedures **quicksort** and **swap**. Every procedure is specified with a pre and postcondition. A partial function annot is defined only for the procedure **partition**, since it is the only one that contains loop statements.

Consider for instance the procedure `swap`. Since it contains no loops, the VCgen returns a single proof obligation. From the definition of the WP function, the proof obligation we obtain is:

$$\mathsf{inBound}(\mathtt{i}) \wedge \mathsf{inBound}(\mathtt{j}) \Rightarrow \mathsf{swapped}(\mathtt{i}, \mathtt{j})[^{[\mathtt{vec:j} \mapsto \mathtt{t}]}/_{\mathtt{vec}}][^{[\mathtt{vec:i} \mapsto \mathtt{vec[j]}]}/_{\mathtt{vec}}][^{\mathtt{vec[i]}}/_{\mathtt{t}}]$$

After computing the set of proof obligations for the whole program, one can see that they are all valid formulae. In the rest of this paper we assume that a certificate is provided for each of these proof obligations.

```
partition(start, end){
  pivot = vec[start];
  i = start;
  j = end;
  while[l_a] (i < j) {
    while[l_b] (vec[i] ≤ pivot ∧ i < j)
      i++;
    while[l_c](pivot < vec[j] ∧ i < j)
      j--;
    if (i < j) swap(i, j);
  }
  swap(start, i-1);
  return i-1;
}
```

Fig. 8. Quicksort Algorithm - Procedure **partition**

4 RTL Verification and Certification

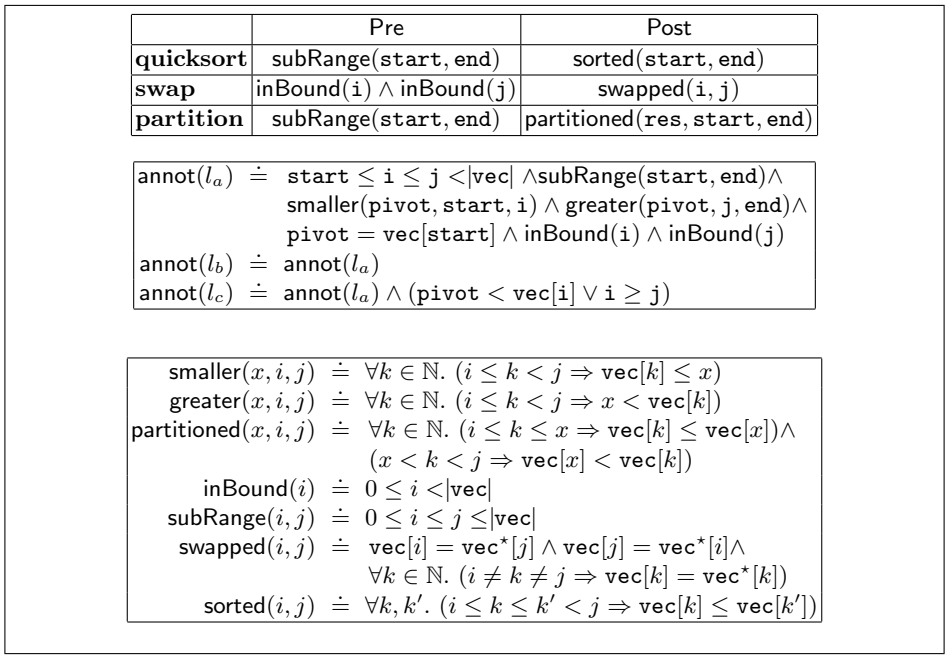
4.1 Programming Language Setting

In this section, we provide a definition of an intermediate RTL program representation. Commonly, most of the compiler optimizations are applied after the program is transformed into this RTL representation.

We define the body of an RTL procedure as a directed graph, where nodes represent program points and edges represent the execution of a statement or a conditional jump. The following definition states this formally.

Definition 3. *The body of an* RTL *procedure is defined by a tuple* $\langle \mathcal{N}, \mathcal{E}, G \rangle$*, where* $\mathcal{N} \subseteq \mathcal{L}$ *is a subset of labels that represents the program points, the relation* $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ *defines the execution flow, and* $G : \mathcal{E} \rightarrow (\mathsf{Instr} + \mathsf{B})$ *maps every edge to instructions or boolean expressions, defined in Figure 10. An* RTL *program* P *is defined as a collection of* RTL *procedures.*

As can be seen in Figure 10, boolean conditions are defined as integer comparisons between two variables. Similarly, instructions involve at most one array access or two program variables. In the figure, e represents an integer expression (one array access or an arithmetic operation between at most two scalar variables).



	Pre	Post
quicksort	$\mathsf{subRange}(\mathtt{start}, \mathtt{end})$	$\mathsf{sorted}(\mathtt{start}, \mathtt{end})$
swap	$\mathsf{inBound}(\mathtt{i}) \wedge \mathsf{inBound}(\mathtt{j})$	$\mathsf{swapped}(\mathtt{i}, \mathtt{j})$
partition	$\mathsf{subRange}(\mathtt{start}, \mathtt{end})$	$\mathsf{partitioned}(\mathtt{res}, \mathtt{start}, \mathtt{end})$

$$
\begin{array}{rcl}
\mathsf{annot}(l_a) & \doteq & \mathtt{start} \leq \mathtt{i} \leq \mathtt{j} < |\mathtt{vec}| \wedge \mathsf{subRange}(\mathtt{start}, \mathtt{end}) \wedge \\
 & & \mathsf{smaller}(\mathtt{pivot}, \mathtt{start}, \mathtt{i}) \wedge \mathsf{greater}(\mathtt{pivot}, \mathtt{j}, \mathtt{end}) \wedge \\
 & & \mathtt{pivot} = \mathtt{vec}[\mathtt{start}] \wedge \mathsf{inBound}(\mathtt{i}) \wedge \mathsf{inBound}(\mathtt{j}) \\
\mathsf{annot}(l_b) & \doteq & \mathsf{annot}(l_a) \\
\mathsf{annot}(l_c) & \doteq & \mathsf{annot}(l_a) \wedge (\mathtt{pivot} < \mathtt{vec}[\mathtt{i}] \vee \mathtt{i} \geq \mathtt{j})
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{smaller}(x,i,j) & \doteq & \forall k \in \mathbb{N}.\ (i \leq k < j \Rightarrow \mathtt{vec}[k] \leq x) \\
\mathsf{greater}(x,i,j) & \doteq & \forall k \in \mathbb{N}.\ (i \leq k < j \Rightarrow x < \mathtt{vec}[k]) \\
\mathsf{partitioned}(x,i,j) & \doteq & \forall k \in \mathbb{N}.\ (i \leq k \leq x \Rightarrow \mathtt{vec}[k] \leq \mathtt{vec}[x]) \wedge \\
 & & (x < k < j \Rightarrow \mathtt{vec}[x] < \mathtt{vec}[k]) \\
\mathsf{inBound}(i) & \doteq & 0 \leq i < |\mathtt{vec}| \\
\mathsf{subRange}(i,j) & \doteq & 0 \leq i \leq j \leq |\mathtt{vec}| \\
\mathsf{swapped}(i,j) & \doteq & \mathtt{vec}[i] = \mathtt{vec}^\star[j] \wedge \mathtt{vec}[j] = \mathtt{vec}^\star[i] \wedge \\
 & & \forall k \in \mathbb{N}.\ (i \neq k \neq j \Rightarrow \mathtt{vec}[k] = \mathtt{vec}^\star[k]) \\
\mathsf{sorted}(i,j) & \doteq & \forall k, k'.\ (i \leq k \leq k' < j \Rightarrow \mathtt{vec}[k] \leq \mathtt{vec}[k'])
\end{array}
$$

Fig. 9. Quicksort Algorithm Specification

For every $l \in \mathcal{N}$, we denote $\mathsf{succ}(l)$ the set of successors of node l, i.e., $\{l' \in \mathcal{N} \mid \langle l, l' \rangle \in \mathcal{E}\}$.

In the rest of the chapter, we use the subscript p to make explicit that the representation $\langle \mathcal{N}_p, \mathcal{E}_p, G_p \rangle$ belongs to a procedure p. We omit, however, the subscript p when it is clear from the context.

In order to define the semantics of RTL programs, we need to define a notion of well-formed code representation.

Definition 4 (Well-formed Program). *A procedure representation $\langle \mathcal{N}, \mathcal{E}, G \rangle$ is well-formed if*

- *$l_{\mathsf{in}}, l_{\mathsf{out}} \in \mathcal{L}$, representing the initial and final label, respectively, are in $\mathcal{N}$. Furthermore, $\{l \mid l_{\mathsf{in}} \in \mathsf{succ}(l)\} = \mathsf{succ}(l_{\mathsf{out}}) = \emptyset$.*
- *The graph is closed. Formally, for every $l \in \mathcal{N}$, we have that $\mathsf{succ}(l) \subseteq \mathcal{N}$.*

A program is well-formed if all its procedures are well-formed.

In the rest of the chapter we consider only well-formed RTL programs. Furthermore, to ensure determinism, we assume that for every node $l \in \mathcal{N}$, only one of the following situations arise:

- there are exactly two outgoing edges $\langle l, l_t \rangle$ and $\langle l, l_f \rangle$, and they are mapped by G to boolean conditions such that $G[\langle l, l_f \rangle] = \neg G[\langle l, l_f \rangle]$, or

$$
\begin{array}{rl}
\text{(B)} & b ::= v_1 \bowtie v_2 \mid \neg(v_1 \bowtie v_2) \\
\textbf{(expressions)} & e ::= n \mid v \mid n \oplus v \mid v \oplus v \mid a[v] \\
\text{(Instr)} & ins ::= \textbf{nop} \mid v := e \mid a[v] := v \\
 & \quad \mid \textbf{invoke}\ f\ (\vec{x}) \mid \textbf{return}\ v
\end{array}
$$

Fig. 10. RTL Instructions

– there is a single outgoing edge $\langle l, l' \rangle$, such that $G[\langle l, l' \rangle] \in \mathsf{Instr}$.

The RTL semantics and verification setting consider non-deterministic RTL programs. However, we can restrict our attention to deterministic RTL programs, since the conditions above are satisfied by the result of compiling a high-level program into the RTL representation.

Let Σ and Σ_F the set of intermediate and final states defined as in previous section. The semantics of well-formed RTL programs is defined by a relation $\rightsquigarrow_p : \mathcal{L} \times \Sigma \to \Sigma_F$, where p denotes the procedure under execution.

$$
\frac{G_p[\langle l, l_{\mathsf{out}} \rangle] = \textbf{return}\ v}{\langle l, \sigma \rangle \rightsquigarrow_p \langle [\![v]\!]_\sigma, \sigma_{\mathcal{A}} \rangle} \qquad \frac{G_p[\langle l, l' \rangle] = \textbf{nop} \qquad \langle l', \sigma \rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, \sigma \rangle \rightsquigarrow_p s}
$$

$$
\frac{\begin{array}{c} G_p[\langle l, l' \rangle] = v := \textbf{invoke}\ p'\ x \qquad \langle l_{\mathsf{in}}, \langle [x_{p'} \mapsto [\![x]\!]_\sigma], \sigma_{\mathcal{A}} \rangle \rangle \rightsquigarrow_{p'} \langle n, \langle \sigma'_{\mathcal{V}}, \sigma'_{\mathcal{A}} \rangle \rangle \\ \langle l', \langle [\sigma_{\mathcal{V}} : v \mapsto n], \sigma'_{\mathcal{A}} \rangle \rangle \rightsquigarrow_p s \qquad s \in \Sigma_F \end{array}}{\langle l, \sigma \rangle \rightsquigarrow_p s}
$$

$$
\frac{G_p[\langle l, l' \rangle] = b \qquad b \in \{(v_1 \bowtie v_2), \neg(v_1 \bowtie v_2)\} \qquad [\![b]\!]_\sigma \qquad \langle l', \sigma \rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, \sigma \rangle \rightsquigarrow_p s}
$$

$$
\frac{G_p[\langle l, l' \rangle] = v := e \qquad [\![e]\!]_\sigma = n \in \mathbb{Z} \qquad \langle l', \langle [\sigma_{\mathcal{V}} : v \mapsto n], \sigma_{\mathcal{A}} \rangle \rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, \langle \sigma_{\mathcal{V}}, \sigma_{\mathcal{A}} \rangle \rangle \rightsquigarrow_p s}
$$

$$
\frac{\begin{array}{c} G_p[\langle l, l' \rangle] = a[v_1] := v_2 \\ 0 \leq [\![v_1]\!]_\sigma < |a| \qquad \langle l', \langle \sigma_{\mathcal{V}}, [\sigma_{\mathcal{A}} : a \mapsto [a : [\![v_1]\!]_\sigma \mapsto [\![v_2]\!]_\sigma]] \rangle \rangle \rightsquigarrow_p s \qquad s \in \Sigma_F \end{array}}{\langle l, \langle \sigma_{\mathcal{V}}, \sigma_{\mathcal{A}} \rangle \rangle \rightsquigarrow_p s}
$$

Fig. 11. Semantics of RTL Programs

Example 2. As an example of the intermediate program representation, consider the RTL graph of the procedure **partition** shown in Figure 15.

4.2 Verification Setting

Specification language. A specification of an RTL procedure is defined by a tuple $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$, where annot maps program labels in $\mathcal{L}$ to intermediate specifications. The first-order formulae that define the specification follow the same restrictions as in previous section.

As before, we require every cycle of the control flow graph of the program to be annotated, so that a full program annotation can be generated from the partial annotation annot.

Definition 5 (Well-annotated RTL Program). *A procedure with specification* $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$ *is well-annotated if every loop in the procedure body contains at least one annotation. That is, for every cyclic path* $\langle l_1, l_2\rangle ... \langle l_k, l_1\rangle$ *we require* $\{l_1, \ldots, l_k\} \cap \mathsf{dom}(\mathsf{annot}) \neq \emptyset$*. A program is well-annotated if all its procedures are well-annotated.*

Given a cycle of the directed graph, a first-order formulae annotating one of its labels can be interpreted as a loop invariant. In fact, as we state later, the result of compiling a well-annotated program is a well-annotated RTL program, in which the original loop invariants annotate every cycle of the control-flow graph.

Notice that the definition of well-annotated code provides an induction principle on the set of labels, with $\mathsf{dom}(\mathsf{annot})$ as the set of base cases and $\mathcal{E} \cap (\mathcal{L} \times (\mathcal{L} \setminus \mathsf{dom}(\mathsf{annot})))$ an order relation with no infinite chains.

Example 3. Let $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$ be the specification for the code in Figure 15. Due to the presence of a cycle in the graph, at least one of the loop labels must be annotated. For instance, as in the source program, one may define the invariant at the loop header l as $\mathsf{annot}(l) \doteq y \geq 1 \wedge c * x^y = x^{\star y^\star}$. One can see that this specification is sufficient for the program to be well-annotated.

Given a well-annotated RTL program, a VCgen is defined by extracting a set of proof obligations from each well-annotated procedure p:

$$\begin{aligned}\mathsf{po}(p) = \{&\mathsf{Pre}_p \Rightarrow \mathsf{wp}_p(l_{\mathsf{in}})[^{V}/_{V^\star}]\} \cup \\ &\{\mathsf{annot}_p(l) \Rightarrow \textstyle\bigwedge_{l' \in \mathsf{succ}(l)} \mathsf{wpi}(G_p[\langle l, l'\rangle], \mathsf{wp}_p(l')) \mid l \in \mathsf{dom}(\mathsf{annot}_p)\}\end{aligned}$$

where the predicate transformers wpi and wp_p are defined in Figure 12. In the figure, a represents every array variable that may get modified by p. The assertion $\varphi[^{V^\star}/_{V}]$ stands for the substitution in φ of every array variable $x^\star \in V^\star$ by $x \in V$.

Lemma 2 (VCgen Soundness). *Consider a well-annotated* RTL *program* P*. Assume that for every procedure* p *of* P*,* $\mathsf{po}(p)$ *is a valid set of proof obligations. Then, for every procedure* p*, if* $\langle l_{\mathsf{in}}, \sigma\rangle \rightsquigarrow_p \langle n, \sigma'\rangle$ *then* $\models [\sigma' : \mathtt{res} \mapsto n] : \mathsf{Post}$*.*

4.3 Certificate Infrastructure

In general, a program certificate can be defined as a mathematical object that provides efficiently verifiable evidence of the validity of logical formulae. There

$$
\begin{aligned}
&\mathsf{wpi}(\mathbf{nop}, \varphi) = \varphi \\
&\mathsf{wpi}(v := e, \varphi) = \varphi[{}^{e}/_{v}] \\
&\mathsf{wpi}(a[v_1] := v_2, \varphi) = \varphi[{}^{[a:v_1 \mapsto v_2]}/_{a}] \\
&\mathsf{wpi}(v_1 \bowtie v_2, \varphi) = (v_1 \bowtie v_2) \Rightarrow \varphi \\
&\mathsf{wpi}(\neg(v_1 \bowtie v_2), \varphi) = \neg(v_1 \bowtie v_2) \Rightarrow \varphi \\
&\mathsf{wpi}(\mathbf{invoke}\ f\ x, \varphi) = \mathsf{Pre}_f[{}^{x}/_{x_f}] \wedge \\
&\qquad\qquad \forall_{\mathtt{res},a'} \mathsf{Post}_f[{}^{a',a}/_{a,a^\star}][{}^{x_f{}^\star}/_{x}] \Rightarrow \varphi[{}^{a'}/_{a}] \\
&\mathsf{wpi}(\mathbf{return}\ v, \varphi) = \varphi[{}^{v}/_{\mathtt{res}}]
\end{aligned}
$$

$$
\mathsf{wp}_p(l) = \begin{cases} \mathsf{Post}_p & \text{if } l = l_{\mathsf{out}} \\ \mathsf{annot}(l) & \text{if } l \in \mathsf{dom}(\mathsf{annot}) \\ \bigwedge_{l' \in \mathsf{succ}(l)} \mathsf{wpi}(G_p[\langle l, l' \rangle], \mathsf{wp}_p(l')) & \text{otherwise} \end{cases}
$$

Fig. 12. RTL VCgen rules

are several formal representation of certificates, depending on competing criteria such as ease of generation and transformation, ease of checking, and the size of certificates. One notion of certificate representation are proof scripts, a sequence of logical deduction statements in the language of a proof-assistant. More commonly, certificates are represented as terms of the λ-calculus, as suggested by the Curry-Howard isomorphism [13].

The development of the certificate transformations depends strongly on the representation of certificates. To provide a generic presentation of the proof transformations, we prefer to abstract from the actual implementation of certificates. Instead, we assume a set of operations over proofs, formalized by an abstract proof algebra, shown in Figure 13.

For instance, an actual implementation of these operations in a λ-term representation of certificates would define the $\mathsf{intro}_\wedge$ operation of the proof algebra as the λ-term $\lambda f.\ \lambda g.\ \lambda a.\ \langle fa, ga \rangle$.

5 Certificate Translation

In general, verification conditions are not preserved by program transformations. Consequently, a priori, the certificates that are used to attest the verification of a source program cannot be reused to certify the transformed program. Furthermore, the original specification can become unprovable. In most cases, the transformation of the certificate is closely dependent on the first step of the optimization, in which the compiler gathers static information about the execution of the program. Indeed, in order to preserve the soundness of the specification, several optimizations require that invariants are strengthened with the result of the analysis that justifies the optimization. Intuitively, this comes as a need to

$$
\begin{array}{lll}
\mathsf{intro}_{\mathsf{true}} & : & \Gamma \vdash \mathsf{true} \\
\mathsf{axiom}\ A & : & \Gamma \vdash A \qquad \text{if } A \in \Gamma \\
\mathsf{ring} & : & \Gamma \vdash n_1 = n_2 \quad \text{if } n_1 = n_2 \text{ is a ring equality} \\
\\
\mathsf{intro}_{\wedge} & : & \Gamma \vdash A \rightarrow \Gamma \vdash B \rightarrow \Gamma \vdash A \wedge B \\
\mathsf{elim}_{\wedge,l} & : & \Gamma \vdash A \wedge B \rightarrow \Gamma \vdash A \\
\mathsf{elim}_{\wedge,r} & : & \Gamma \vdash A \wedge B \rightarrow \Gamma \vdash B \\
\\
\mathsf{intro}_{\Rightarrow} & : & \Gamma; A \vdash B \rightarrow \Gamma \vdash A \Rightarrow B \\
\mathsf{elim}_{\Rightarrow} & : & \Gamma \vdash A \Rightarrow B \rightarrow \Gamma \vdash A \rightarrow \Gamma \vdash B \\
\\
\mathsf{elim}_{=} & : & \Gamma \vdash e_1 = e_2 \rightarrow \Gamma \vdash A[{}^{e_1}\!/_{r}] \rightarrow \Gamma \vdash A[{}^{e_2}\!/_{r}] \\
\\
\mathsf{weak}_{\Delta} & : & \Gamma \vdash A \rightarrow \Gamma; \Delta \vdash A \\
\\
\mathsf{intro}_{\forall} & : & \Gamma \vdash A \rightarrow \Gamma \vdash \forall r.A \qquad \text{if } r \text{ is not in } \Gamma \\
\\
\mathsf{elim}_{\forall} & : & \Gamma \vdash \forall r.A \rightarrow \Gamma \vdash A[{}^{e}\!/_{r}] \\
\\
\mathsf{intro}_{\exists} & : & \Gamma \vdash A[{}^{e_1}\!/_{x}] \Rightarrow B[{}^{e_2}\!/_{y}] \rightarrow \Gamma \vdash \exists x.A \Rightarrow \exists y.B
\end{array}
$$

Fig. 13. Proof Algebra

propagate, through the invariants, the information returned by the analysis, in order to eventually enforce the preservation of the original semantics. That is the case, for instance, of optimizations that simplify the evaluation of expressions, such as constant propagation, common sub-expression elimination, copy propagation and redundant conditional elimination. In such cases, the transformation of the original certificates entails representing the result of the analysis in the underlying verification logic, and generating a certificate for this specification. A certificate transformation process then integrates this certified analysis result with the original certificate in order to generate a certificate for the optimized program. However, it is not always the case that invariants must be strengthened. Other optimizations, e.g. dead-variable elimination, may require loop invariants to be weakened.

In this section, we study several standard compiler optimizations, applied to our particular running example. Even though some optimizations preserve the verification conditions for our particular example, we give a short explanation of the general technique to transform the certificate. For instance, dead register elimination does not alter verification conditions and thus no certificate translation is needed. In other cases, an ad-hoc transformation may seem more convenient in terms of the final annotation and certificate size, but we prefer to formulate the transformation of the certificate as generally as possible.

5.1 Non-optimizing Compilation

Description. The first compiler transformation translates the high-level representation of the source program into the intermediate RTL representation de-

fined in Section 4. The compilation of a procedure p with body c is defined as $\mathcal{C}_{(l_{\mathsf{in}},l_{\mathsf{out}})}(c)$, where the function $\mathcal{C}$ can be found in Figure 14.

The compilation of expressions, $\mathcal{C}^{\mathbf{e}}$, takes a variable v and an expression e and returns a subgraph of RTL instructions that computes the value of the expression e and stores it on the variable v. In the figure, the union of two graphs $\langle \mathcal{N}_1, \mathcal{E}_1, G_1 \rangle$ and $\langle \mathcal{N}_2, \mathcal{E}_2, G_2 \rangle$ is defined as $\langle \mathcal{N}_1 \cup \mathcal{N}_2, \mathcal{E}_1 \cup \mathcal{E}_2, G_1 \cup G_2 \rangle$. The function $\mathcal{C}^{\mathbf{b}}$ compiles the evaluation of a boolean expression, and takes two additional parameters: the labels l_t and l_f into which the execution must jump depending on whether the boolean condition is satisfied. The function $\mathcal{C}$ takes, in addition to a source statement c, a label that points to the code that must be executed after the execution of c.

$$
\begin{array}{l}
\mathcal{C}^{\mathbf{b}}_{(l,l_t,l_f)}(v_1 \bowtie v_2) = \langle \{l, l_t, l_f\}, \{\langle l, l_t\rangle, \langle l, l_f\rangle\}, G\rangle \\
\qquad\qquad\qquad\qquad \text{where:} \quad G[\langle l, l_t\rangle] = (v_1 \bowtie v_2) \\
\qquad\qquad\qquad\qquad\qquad\quad\; G[\langle l, l_f\rangle] = \neg(v_1 \bowtie v_2) \\
\mathcal{C}^{\mathbf{b}}_{(l,l_t,l_f)}(e_1 \bowtie e_2) = \mathcal{C}_{(l,l_1)}(v_1 := e_1) \cup \mathcal{C}_{(l_1,l_2)}(v_2 := e_2) \cup \mathcal{C}^{\mathbf{b}}_{(l_2,l_t,l_f)}(v_1 \bowtie v_2) \\
\mathcal{C}^{\mathbf{b}}_{(l,l_t,l_f)}(\neg b) = \mathcal{C}^{\mathbf{b}}_{(l,l_f,l_t)}(b) \\
\mathcal{C}^{\mathbf{b}}_{(l,l_t,l_f)}(b_1 \wedge b_2) = \mathcal{C}^{\mathbf{b}}_{(l,l'_t,l_f)}(b_1) \cup \mathcal{C}^{\mathbf{b}}_{(l,l_t,l_f)} \\
\\
\mathcal{C}_{(l,l')}(v := v_1 \oplus v_2) = \langle \{l, l'\}, \{\langle l, l'\rangle\}, [\langle l, l'\rangle \mapsto v := v_1 \oplus v_2]\rangle \\
\mathcal{C}_{(l,l')}(v := a[v']) = \langle \{l, l'\}, \{\langle l, l'\rangle\}, [\langle l, l'\rangle \mapsto v := a[v']]\rangle \\
\mathcal{C}_{(l,l')}(v := e_1 \oplus e_2) = \mathcal{C}_{(l,l_1)}(v_1 := e_1) \cup \mathcal{C}_{(l_1,l_2)}(v_2 := e_2) \cup \mathcal{C}_{(l_2,l')}(v := v_1 \oplus v_2) \\
\mathcal{C}_{(l,l')}(v := a[e]) = \mathcal{C}_{(l,l'')}(v' := e) \cup \mathcal{C}_{(l'',l')}(v := a[v']) \\
\\
\mathcal{C}_{(l,l')}(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2) = \mathcal{C}^{\mathbf{b}}_{(l,l_t,l_f)}(b) \cup \mathcal{C}_{(l_t,l')}(c_1) \cup \mathcal{C}_{(l_f,l')}(c_2) \\
\mathcal{C}_{(l,l')}(\textbf{while } b \textbf{ do } c) = \mathcal{C}^{\mathbf{b}}_{(l,l_t,l')}(b) \cup \mathcal{C}_{(l_t,l)}(c) \\
\mathcal{C}_{(l,l')}(v := \textbf{invoke } p(e_1,..,e_k)) = \mathcal{C}(v_1 := e_1) \cup .. \cup \mathcal{C}(v_k := e_k) \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{C}(v := \textbf{invoke } f(v_1,..,v_k)) \\
\mathcal{C}_{(l,l')}(c_1; c_2) = \mathcal{C}_{(l,l'')}(c_1) \cup \mathcal{C}_{(l'',l')}(c_2) \\
\mathcal{C}_{(l,l')}(\textbf{return } e) = \mathcal{C}_{l,l''}(v := e) \cup \langle \{l'', l'\}, \{\langle l'', l'\rangle\}, [\langle l'', l'\rangle \mapsto \textbf{return } v]\rangle
\end{array}
$$

Fig. 14. Compiler Definition

Lemma 3. *A well-formed source program is compiled into a well-formed deterministic* RTL *program.*

After this compilation step, we do not need to modify the original procedure specifications:

Definition 6 (Compilation of Specifications). *Let* $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$ *be a specification for a source level procedure* p*. We define the specification for the compilation of the procedure* p *as* $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$.

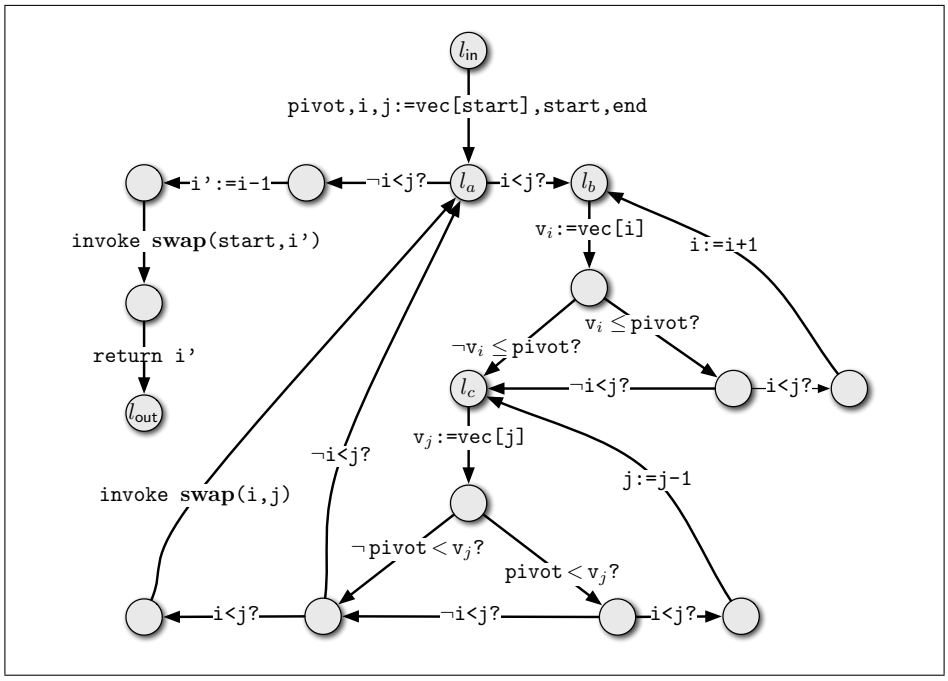


Fig. 15. Intermediate Representation of the procedure `partition`.

Lemma 4. *The result of compiling a well-annotated source program is a well-annotated* RTL *program.*

Transformed running example. The code in Figure 15 is the result of compiling the source program procedure `partition` into an RTL representation.

To simplify the graphical representation, we merge consecutive edges representing assignments into a single edge. The definition of the semantics and the computation of the wp function are easily extended to these edges.

Comparison of verification conditions. Consider the first proof obligation of the source version of the procedure `partition`, i.e. $\mathsf{Pre} \Rightarrow \phi[{}^{\mathtt{vec}}/_{\mathtt{vec}^\star}]$ where $\langle \phi, \xi \rangle = \mathsf{WP}(c, \mathsf{Post})$, for some ξ, and c is the body of the procedure **partition** and Post its postcondition. If we compute the proof obligation we get the following formula:

$$\begin{aligned} \mathsf{Pre} \Rightarrow\ & \mathtt{start} + 1 \leq \mathtt{start} + 1 \leq \mathtt{end} < |\mathtt{vec}| \wedge \\ & \mathsf{smaller}(\mathtt{vec}[\mathtt{start}], \mathtt{start} + 1, \mathtt{start} + 1) \wedge \\ & \mathsf{greater}(\mathtt{vec}[\mathtt{start}], \mathtt{end.end}) \wedge \\ & \mathtt{vec}[\mathtt{start}] = \mathtt{vec}[\mathtt{start}] \wedge \\ & \mathsf{inBound}(\mathtt{i}) \wedge \mathsf{inBound}(\mathtt{j}) \end{aligned}$$

If we unfold the definition of the predicates above we can see that the proof obligation is valid. Computing the corresponding verification condition at the compiled RTL version shows that it is syntactically preserved by non-optimizing compilation.

However, minor transformations of the verification conditions are introduced when considering a fragment of code that evaluates non-trivial conditional expressions. Then, certificates must be adapted accordingly. Consider, for instance, the verification condition related to the loop invariant $\mathsf{annot}(l_b)$. At source level, the VCgen returns the proof obligation $\mathsf{annot}(l_b) \Rightarrow (b \Rightarrow \mathsf{annot}(l_b)[^{\mathtt{i}+1}/_{\mathtt{i}}]) \wedge (\neg b \Rightarrow \mathsf{annot}(l_c))$, where b stands for $\mathtt{vec}[\mathtt{i}] \leq \mathtt{pivot} \wedge \mathtt{i} < \mathtt{j}$. Computing the verification condition at label l_b of the compiled code returns

$$\begin{aligned} \mathsf{annot}(l_b) \Rightarrow\ & (\mathtt{vec}[\mathtt{i}] \leq \mathtt{pivot} \Rightarrow \mathtt{i} < \mathtt{j} \Rightarrow \mathsf{annot}(l_b)[^{\mathtt{i}+1}/_{\mathtt{i}}]) \wedge \\ & (\mathtt{vec}[\mathtt{i}] \leq \mathtt{pivot} \Rightarrow \neg \mathtt{i} < \mathtt{j} \Rightarrow \mathsf{annot}(l_c)) \wedge \\ & (\neg \mathtt{vec}[\mathtt{i}] \leq \mathtt{pivot} \Rightarrow \mathsf{annot}(l_c)) \end{aligned}$$

A proof for the verification condition above can be generated from the original one. In the rest of this section we generalize this result to any certified program.

Transformation of the certificate. The following generic results state that it is possible to reconstruct the original certificates in the presence of non-optimizing compilation.

The first lemma states that the application of the predicate transformer wpi at RTL level *decompiles* the code that results from the compilation of assignments.

Lemma 5. *Let $\langle \mathcal{N}, \mathcal{E}, G \rangle$ be the subgraph that results from computing the assignment $x := e$, i.e. , $\mathcal{C}_{(l,l')}(x := e)$. Then, by structural induction on e, one can proof that $\mathsf{wp}(l)$ is syntactically equal to $\mathsf{wp}(l')[^{e}/_{v}]$.*

Then, the following lemma states a correspondence between a boolean condition and the code that results from its compilation.

Lemma 6. *Let $\langle \mathcal{N}, \mathcal{E}, G \rangle$ be the subgraph that results from compiling the boolean condition b, i.e., $\mathcal{C}^{\mathbf{b}}_{(l,l_t,l_f)}(b)$. Then, one can generate a certificate $c : (b \Rightarrow \mathsf{wp}(l_t)) \wedge (\neg b \Rightarrow \mathsf{wp}(l_f)) \vdash \mathsf{wp}(l)$.*

Proof. The proof follows by structural induction on b, from Lemma 5 and by using the operations on Figure 13. Consider, for instance, the base case, i.e. that b is equal to $v_1 \bowtie v_2$. Then one can show that $\mathsf{wp}(l)$ is syntactically equal to $(b \Rightarrow \mathsf{wp}(l_t)) \wedge (\neg b \Rightarrow \mathsf{wp}(l_f))$, and thus axiom *is a certificate for the goal we want to prove. Consider now the case $b = b_1 \wedge b_2$ and thus $\langle \mathcal{N}, \mathcal{E}, G \rangle$ is equal to $\mathcal{C}^{\mathbf{b}}_{l,l'_t,l_f}(b_1) \cup \mathcal{C}^{\mathbf{b}}_{l,l_t,l_f}(b_2)$. By inductive hypothesis, we know that we can generate a certificate for the goals*

$$c_1 : (b_1 \Rightarrow \mathsf{wp}(l'_t)) \wedge (\neg b_1 \Rightarrow \mathsf{wp}(l_f)) \vdash \mathsf{wp}(l)$$

and

$$c_2 : (b_2 \Rightarrow \mathsf{wp}(l_t)) \wedge (\neg b_2 \Rightarrow \mathsf{wp}(l_f)) \vdash \mathsf{wp}(l'_t)$$

Let φ stands for the formula

$$(b_1 \wedge b_2 \Rightarrow \mathsf{wp}(l_t)) \wedge (\neg(b_1 \wedge b_2) \Rightarrow \mathsf{wp}(l_f))$$

The following derivation steps define the certificate c for the goal we want to prove in this case (we assume $\neg\varphi$ is a syntax sugar for $\varphi \Rightarrow \mathsf{false}$):

$$\begin{array}{ll}
p_1 & = \mathsf{elim}_{\wedge,r}(\mathsf{axiom}(b_1 \wedge b_2)) : \varphi, b_1, \neg b_2, b_1 \wedge b_2 \vdash b_2 \\
p_2 & = \mathsf{elim}_{\Rightarrow}(\mathsf{axiom}(\neg b_2), p_1) : \varphi, b_1, \neg b_2, b_1 \wedge b_2 \vdash \mathsf{false} \\
p_3 & = \mathsf{intro}_{\Rightarrow}(p_2) : \varphi, b_1, \neg b_2 \vdash \neg(b_1 \wedge b_2) \\
p_4 & = \mathsf{axiom}(\varphi) : \varphi, b_1, \neg b_2 \vdash \varphi \\
p_5 & = \mathsf{elim}_{\wedge,l}(p_4) : \varphi, b_1, \neg b_2 \vdash \neg(b_1 \wedge b_2) \Rightarrow \mathsf{wp}(l_f) \\
p_6 & = \mathsf{elim}_{\Rightarrow}(p_3, p_5) : \varphi, b_1, \neg b_2 \vdash \mathsf{wp}(l_f) \\
p_7 & = \mathsf{intro}_{\Rightarrow}(p_6) : \varphi, b_1 \vdash \neg b_2 \Rightarrow \mathsf{wp}(l_f) \\
p_8 & = \mathsf{intro}_{\wedge}(\mathsf{axiom}(b_1), \mathsf{axiom}(b_2)) : \varphi, b_1, b_2 \vdash b_1 \wedge b_2 \\
p_9 & = \mathsf{elim}_{\wedge,r}(\mathsf{axiom}(\varphi)) : \varphi, b_1, b_2 \vdash b_1 \wedge b_2 \Rightarrow \mathsf{wp}(l_t) \\
p_{10} & = \mathsf{elim}_{\Rightarrow}(p_8, p_9) : \varphi, b_1, b_2 \vdash \mathsf{wp}(l_t) \\
p_{11} & = \mathsf{intro}_{\Rightarrow}(p_{10}) : \varphi, b_1 \vdash b_2 \Rightarrow \mathsf{wp}(l_t) \\
p_{12} & = \mathsf{intro}_{\wedge}(p_7, p_{11}) : \varphi, b_1 \vdash b_2 \Rightarrow \mathsf{wp}(l_t) \wedge \neg b_2 \Rightarrow \mathsf{wp}(l_f) \\
p_{13} & = \mathsf{elim}_{\Rightarrow}(\mathsf{weak}(\mathsf{intro}_{\Rightarrow}(c_2)), p_{12}) : \varphi, b_1 \vdash \mathsf{wp}(l'_t) \\
p_{14} & = \mathsf{intro}_{\Rightarrow}(p_{13}) : \varphi \vdash b_1 \Rightarrow \mathsf{wp}(l'_t) \\
p_{15} & = \mathsf{axiom}(\neg b_1) : \varphi, \neg b_1, b_1 \wedge b_2 \vdash \neg b_1 \\
p_{16} & = \mathsf{elim}_{\wedge,l}(\mathsf{axiom}(b_1 \wedge b_2)) : \varphi, \neg b_1, b_1 \wedge b_2 \vdash b_1 \\
p_{17} & = \mathsf{elim}_{\Rightarrow}(p_{15}, p_{16}) : \varphi, \neg b_1, b_1 \wedge b_2 \vdash \mathsf{false} \\
p_{18} & = \mathsf{intro}_{\Rightarrow}(p_{17}) : \varphi, \neg b_1 \vdash \neg(b_1 \wedge b_2) \\
p_{19} & = \mathsf{elim}_{\wedge,r}(\mathsf{axiom}(\varphi)) : \varphi, \neg b_1 \vdash \neg(b_1 \wedge b_2) \Rightarrow \mathsf{wp}(l_f) \\
p_{20} & = \mathsf{elim}_{\Rightarrow}(p_{18}, p_{19}) : \varphi, \neg b_1 \vdash \mathsf{wp}(l_f) \\
p_{21} & =: \varphi, \neg b_1 \vdash \mathsf{wp}(l_f) \\
p_{22} & = \mathsf{intro}_{\Rightarrow}(p_{21}) : \varphi \vdash \neg b_1 \Rightarrow \mathsf{wp}(l_f) \\
p_{23} & = \mathsf{intro}_{\wedge}(p_{14}, p_{23}) : \varphi \vdash (b_1 \Rightarrow \mathsf{wp}(l'_t)) \wedge (\neg b_1 \Rightarrow \mathsf{wp}(l_f)) \\
c & = \mathsf{elim}_{\Rightarrow}(p_{23}, \mathsf{intro}_{\Rightarrow}(c_1)) : \varphi \vdash \mathsf{wp}(l)
\end{array}$$

Based on these previous results, the following lemma relates the computation of verification conditions between a source program and its RTL representation.

Lemma 7. *Let c be a statement of a procedure p and $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$ its specification. Let $\langle \mathcal{N}, \mathcal{E}, G \rangle$ be defined as $\mathcal{C}_{(l,l')}(c)$, and $(\varphi, \theta) = \mathsf{WP}(c, \mathsf{wp}(l'))$. Then, one can generate, for every program label $l \in \mathcal{N}$, certificates for the goal $\vdash \varphi \Rightarrow \mathsf{wp}(l)$.*

*Proof. The proof proceeds by structural induction on the statement c. Consider for instance the case of a conditional statement, i.e., $c =$ **if** b **then** c_1 **else** c_2. Then, $\langle \mathcal{N}, \mathcal{E}, G \rangle$ is defined as $\mathcal{C}^{\mathbf{b}}_{l,l_t,l_f}(b) \cup \mathcal{C}_{l_t,l'}(c_1) \cup \mathcal{C}_{l_f,l'}(c_2)$. By I.H., we have the certificates*

$$q_1 : \quad \vdash \varphi_1 \Rightarrow \mathsf{wp}(l_t)$$

and

$$q_2 : \quad \vdash \varphi_2 \Rightarrow \mathsf{wp}(l_f)$$

where $\mathsf{WP}(c_1, \mathsf{wp}(l')) = (\varphi_1, \theta_1)$ *and* $\mathsf{WP}(c_2, \mathsf{wp}(l')) = (\varphi_2, \theta_2)$ *for some sets* θ_1 *and* θ_2*. By definition we have* φ *equal to* $b \Rightarrow \varphi_1 \wedge \neg b \Rightarrow \varphi_2$*. From Lemma 6, we have a certificate* $q : b \Rightarrow \mathsf{wp}(l_t) \wedge \neg b \Rightarrow \mathsf{wp}(l_f) \vdash \mathsf{wp}(l)$*. The following steps constructs the certificate:*

$$
\begin{aligned}
p_1 &= \mathsf{axiom}(b) : \varphi, b \vdash b \\
p_2 &= \mathsf{elim}_{\wedge,l}(\mathsf{axiom}(\varphi)) : \varphi, b \vdash b \Rightarrow \varphi_1 \\
p_3 &= \mathsf{elim}_{\Rightarrow}(p_1, p_2) : \varphi, b \vdash \varphi_1 \\
p_4 &= \mathsf{weak}(q_1) : \varphi, b \vdash \varphi_1 \Rightarrow \mathsf{wp}(l_t) \\
p_5 &= \mathsf{elim}_{\Rightarrow}(p_3, p_4) : \varphi, b \vdash \mathsf{wp}(l_t) \\
p_6 &= \mathsf{intro}_{\Rightarrow}(p_5) : \varphi \vdash b \Rightarrow \mathsf{wp}(l_t) \\
p_7 &= \mathsf{axiom}(\neg b) : \varphi, \neg b \vdash \neg b \\
p_8 &= \mathsf{elim}_{\wedge,r}(\mathsf{axiom}(\varphi)) : \varphi, \neg b \vdash \neg b \Rightarrow \varphi_2 \\
p_9 &= \mathsf{elim}_{\Rightarrow}(p_7, p_8) : \varphi, \neg b \vdash \varphi_2 \\
p_{10} &= \mathsf{weak}(\mathsf{elim}_{\wedge,r}(q_2)) : \varphi, \neg b \vdash \varphi_2 \Rightarrow \mathsf{wp}(l_f) \\
p_{11} &= \mathsf{elim}_{\Rightarrow}(p_9, p_{10}) : \varphi, \neg b \vdash \mathsf{wp}(l_f) \\
p_{12} &= \mathsf{intro}_{\Rightarrow}(p_{11}) : \varphi \vdash \neg b \Rightarrow \mathsf{wp}(l_f) \\
p_{13} &= \mathsf{intro}_{\wedge}(p_6, p_{12}) : \varphi \vdash b \Rightarrow \mathsf{wp}(l_t) \wedge \neg b \Rightarrow \mathsf{wp}(l_f) \\
p_{14} &= \mathsf{elim}_{\Rightarrow}(\mathsf{intro}_{\Rightarrow}(\mathsf{weak}_{\varphi}(q)), p_{13}) : \varphi \vdash \mathsf{wp}(l) \\
p_{15} &= \mathsf{intro}_{\Rightarrow}(p_{14}) : \vdash \varphi \Rightarrow \mathsf{wp}(l)
\end{aligned}
$$

Theorem 1 (Equivalence of Proof Obligations). *Let* p *be a high-level procedure with specification* $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$*. Let* $\bar{p}$ *be an* RTL *procedure defined as the compilation of* p*, i.e.,* $\langle \mathcal{N}, \mathcal{E}, G \rangle$ *is equal to* $\mathcal{C}_{(l_{\mathsf{in}}, l_{\mathsf{out}})}(c)$ *where* c *is the body of* p*. Then, from Lemma 7, one can generate a certificate for the proof obligations in* $\mathsf{po}(\hat{p})$ *from the original certificate for the proof obligations in* $\mathsf{PO}(p)$*.*

5.2 Compilation of the Array Representation

Description. One particular difference between high and low level representations is how memory addressing, i.e. array access, is implemented. This compiler step models abstractly the typical distinction between addressing byte and integer array representations, by multiplying the value used to access an array cell by 4 (assuming an integer value is represented exactly with 4 byte values). Every array a of the source program is then compiled to a corresponding lower-level array $\hat{a}$ such that $|\hat{a}| = 4 * |a|$ and for every integer number n s.t. $0 \leq n < |a|$, we have $\hat{a}[4 * n] = a[n]$. The transformation of an RTL function $\langle \mathcal{N}, \mathcal{E}, G \rangle$ into $\langle \mathcal{N}, \mathcal{E}, \bar{G} \rangle$ is shown in Figure 16. Every assignment that contains an array access is replaced by two consecutive assignments. For simplicity, we abuse notation and do not make explicit the introduction of a fresh intermediate node.

Example 4. The code in Figure 17 is the result of transforming the array representation from the RTL code of Figure 15.

Since every array variable a is compiled into a lower-level array variable $\hat{a}$, we need to modify the original specification accordingly. To that end, we cannot simply substitute the occurrences of a by $\hat{a}$. Instead, we need to define a more

$$
\begin{array}{ll}
\bar{G}[\langle l, l'\rangle] \doteq v' := 4 * v; x := \hat{a}[v'] & \text{if } G[\langle l, l'\rangle] \doteq x := a[v] \\
\bar{G}[\langle l, l'\rangle] \doteq v' := 4 * v; \hat{a}[v'] := x & \text{if } G[\langle l, l'\rangle] \doteq a[v] := x \\
\bar{G}[e] \doteq G[e] & \text{otherwise}
\end{array}
$$

Fig. 16. Compiler Definition

complex renaming function. Let θ_a stand for the proposition $\forall i.\ a[i] = \hat{a}[4 * i]$. For every assertion φ, we denote $\alpha_a(\varphi)$ the assertion $\exists a.\ (\varphi \wedge \theta_a)$, i.e. a renaming, in φ, of the array variable a into its corresponding lower-level array variable $\hat{a}$.

Definition 7 (Compilation of Specifications). *Let* $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$ *be the original specification of a procedure* p*. We define the specification for the compilation of the procedure* p *as* $(\alpha_a(\mathsf{Pre}), \alpha_a \circ \mathsf{annot}, \alpha_a(\mathsf{Post}))$.

Comparison of verification conditions. Computing the verification condition at label l_2 returns:

$$
\begin{array}{rl}
\mathsf{annot}(l_b) \Rightarrow & (\mathtt{vec}[i] \leq \mathtt{pivot} \Rightarrow \mathtt{i} < \mathtt{j} \Rightarrow \mathsf{annot}(l_b)[^{i+1}/_{i}]) \wedge \\
& (\mathtt{vec}[i] \leq \mathtt{pivot} \Rightarrow \neg \mathtt{i} < \mathtt{j} \Rightarrow \mathsf{annot}(l_c)) \wedge \\
& (\neg \mathtt{vec}[i] \leq \mathtt{pivot} \Rightarrow \mathsf{annot}(l_c))
\end{array}
$$

Computing the verification condition at the same label from the transformed program returns:

$$
\begin{array}{rl}
\alpha_{\mathtt{vec}}(\mathsf{annot}(l_b)) \Rightarrow & (\hat{\mathtt{vec}}[4 * \mathtt{i}] \leq \mathtt{pivot} \Rightarrow \mathtt{i} < \mathtt{j} \Rightarrow \alpha_{\mathtt{vec}}(\mathsf{annot}(l_b))[^{\mathtt{i}+1}/_{\mathtt{i}}]) \wedge \\
& (\hat{\mathtt{vec}}[4 * \mathtt{i}] \leq \mathtt{pivot} \Rightarrow \neg \mathtt{i} < \mathtt{j} \Rightarrow \alpha_{\mathtt{vec}}(\mathsf{annot}(l_c))) \wedge \\
& (\neg \hat{\mathtt{vec}}[4 * \mathtt{i}] \leq \mathtt{pivot} \Rightarrow \alpha_{\mathtt{vec}}(\mathsf{annot}(l_c)))
\end{array}
$$

Notice that they are equivalent up to renaming of the array variable `vec`. Then, it should be clear that one can prove the former from the latter. In the rest of this section we show how we can systematically construct certificates for the transformed proof obligations from the original program certificates.

Transformation of the certificate. The following generic results state that its is possible to reconstruct a certificate for the final code from the original certificate.

Lemma 8. *Let* f *be the original procedure, and* $\bar{f}$ *the result of transforming the representation of arrays. Then, one can generate, for every program label* $l \in \mathsf{dom}(\mathsf{annot})$*, certificates for the goal:* $\vdash \alpha_{\mathtt{vec}}(\mathsf{wp}_f(l)) \Rightarrow \mathsf{wp}_{\bar{f}}(l)$.

Proof. The proof proceeds by the induction principle associated to the definition of well-annotated programs. The base cases, i.e. the labels l *such that* $l \in \mathsf{dom}(\mathsf{annot})$ *or* $l = l_{\mathsf{out}}$*, are trivial since by definition* $\mathsf{wp}_{\bar{f}}(l) = \alpha_{\mathtt{vec}}(\mathsf{wp}_f(l))$.

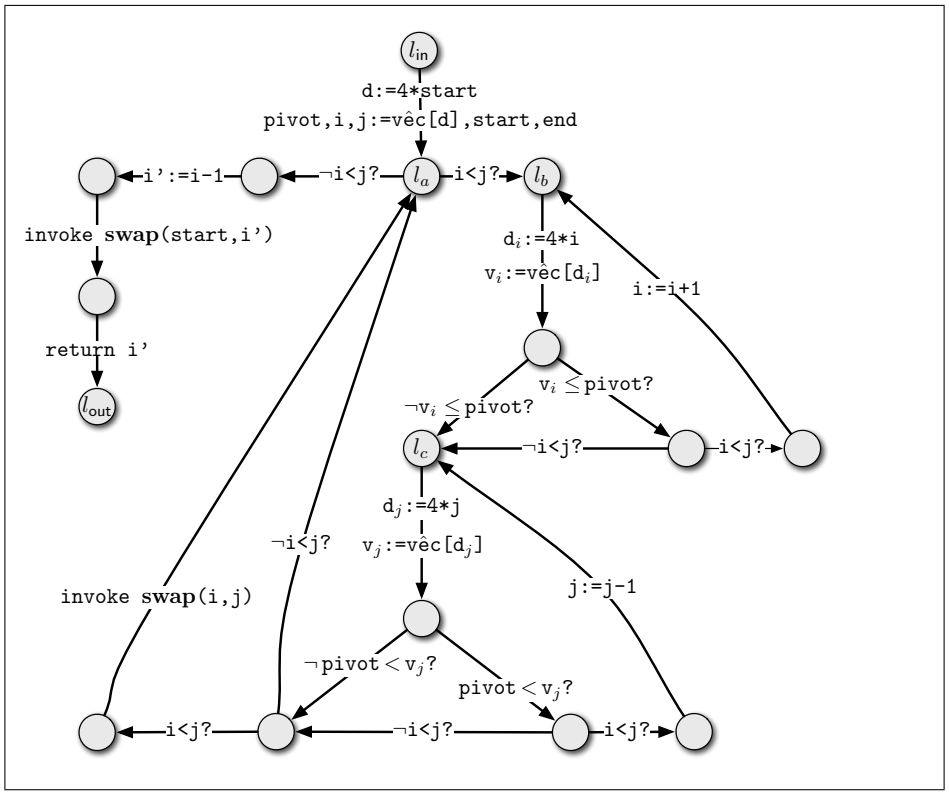


Fig. 17. RTL procedure `partition` after array compilation.

Consider the case $G_f[\langle l, l'\rangle] = x := a[v]$. *Then* $\mathsf{wp}_f(l) = \mathsf{wp}_f(l')[^{\mathtt{vec[v]}}/_x]$ *and* $\mathsf{wp}_{\bar{f}}(l) = \mathsf{wp}_{\bar{f}}(l')[^{\mathtt{vêc[4*v]}}/_x]$. *Let* c' *stand for the certificate generated as inductive hypothesis, i.e,* $c' : \alpha_{\mathtt{vec}}(\mathsf{wp}_f(l)) \vdash \mathsf{wp}_{\bar{f}}(l)$. *The following derivation steps construct the certificate we need for this proof case:*

$$
\begin{aligned}
&p_1{=}\mathsf{elim}_{\wedge,r}(\mathsf{axiom}) : \mathsf{wp}_f(l')[^{\mathtt{vec[v]}}/_x] \wedge \theta \vdash \theta\\
&p_2{=}\mathsf{elim}_{\wedge,l}(\mathsf{axiom}) : \mathsf{wp}_f(l')[^{\mathtt{vec[v]}}/_x] \wedge \theta \vdash \mathsf{wp}_f(l')[^{a[v]}/_x]\\
&p_3{=}\mathsf{elim}_{\forall}(p_1) : \mathsf{wp}_f(l')[^{\mathtt{vec[v]}}/_x] \wedge \theta \vdash \mathtt{vec[v]} = \mathtt{vêc[v]}\\
&p_4{=}\mathsf{elim}_{=}(p_3, p_2) : \mathsf{wp}_f(l')[^{\mathtt{vec[v]}}/_x] \wedge \theta \vdash \mathsf{wp}_f(l')[^{\mathtt{vêc[v]}}/_x]\\
&p_5{=}\mathsf{intro}_{\wedge}(p_4, p_1) : \mathsf{wp}_f(l')[^{\mathtt{vec[v]}}/_x] \wedge \theta \vdash \mathsf{wp}_f(l')[^{\mathtt{vêc[v]}}/_x] \wedge \theta\\
&p_6{=}\mathsf{intro}_{\exists}(\mathsf{intro}_{\Rightarrow}(p_5)) :\vdash \exists \mathsf{vec}.\ \mathsf{wp}_f(l')[^{\mathtt{vec[v]}}/_x] \wedge \theta \Rightarrow \exists \mathsf{vec}.\ \mathsf{wp}_f(l')[^{\mathtt{vêc[v]}}/_x] \wedge \theta\\
&p_7{=}\mathsf{intro}_{\Rightarrow}(\mathsf{elim}_{\Rightarrow}(\mathsf{axiom}, \mathsf{weak}(p_6))) :\\
&\qquad\qquad \vdash \exists \mathsf{vec}.\ \mathsf{wp}_f(l')[^{\mathtt{vec[v]}}/_x] \wedge \theta \Rightarrow \exists \mathsf{vec}.\ \mathsf{wp}_f(l')[^{\mathtt{vêc[v]}}/_x] \wedge \theta
\end{aligned}
$$

Theorem 2 (Certificate Translation). *Let* p *be an* RTL *procedure with specification* $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$. *Let* $\hat{p}$ *stand for the result of compiling the array ex-*

pressions in the procedure p. Then, one can generate certificates for the proof obligations of $\hat{p}$ from the certificates for the original procedure p.

5.3 Loop Induction Variable Strength Reduction

Description. Loop induction strength reduction is an optimization that reduces the complexity of the arithmetic operations executed inside a loop. Basically, an induction variable of a loop is a variable that is incremented (or decremented) inside the loop by a constant value. A *derived* induction variable of the loop is a variable that is defined as a linear function on an induction variable of the loop. For instance, in the following code fragment:

$$
\begin{array}{l}
\texttt{i} := 0; \\
\textbf{while } (\texttt{i} < N) \textbf{ do} \\
\quad \ldots \\
\quad \texttt{j} := a * \texttt{i} + c \\
\quad \texttt{i} := \texttt{i} + 1;
\end{array}
$$

the program variable i is a loop induction variable (with an increment of 1), and j is a derived induction variable defined as the linear function $a * i + c$. In the example above, one can see an optimization opportunity if the multiplication operation is replaced by a less costly addition operation. The following code shows an optimized version of the example above:

$$
\begin{array}{l}
\texttt{i} := 0; \\
\texttt{j} := c; \\
\textbf{while } (\texttt{i} < N) \textbf{ do} \\
\quad \ldots \\
\quad \texttt{j} := \texttt{j} + a \\
\quad \texttt{i} := \texttt{i} + 1;
\end{array}
$$

It should be clear that the transformation preserves the original semantics.

Transformed running example. Consider the optimization of the running example of Figure 15. For convenience in explaining the certificate translation process, we have split the transformation in two independent steps. In the first one, for each derived induction variable j we introduce a corresponding fresh variable j' and a set of assignments to j' in order to make j' hold the same value as j. We require these new assignments to be less costly than those updating j, and that they do not read the value of j.

In the procedure **partition** of the quicksort example, we are interested on reducing the strength of the derived induction variables $\texttt{d}_i$ and $\texttt{d}_j$, defined as linear functions $4{*}\texttt{i}$ and $4{*}\texttt{j}$, respectively. To that end, we introduce assignments immediately after each assignment of `i` and `j`. This first transformation step of the procedure **partition** can be found in Figure 18.

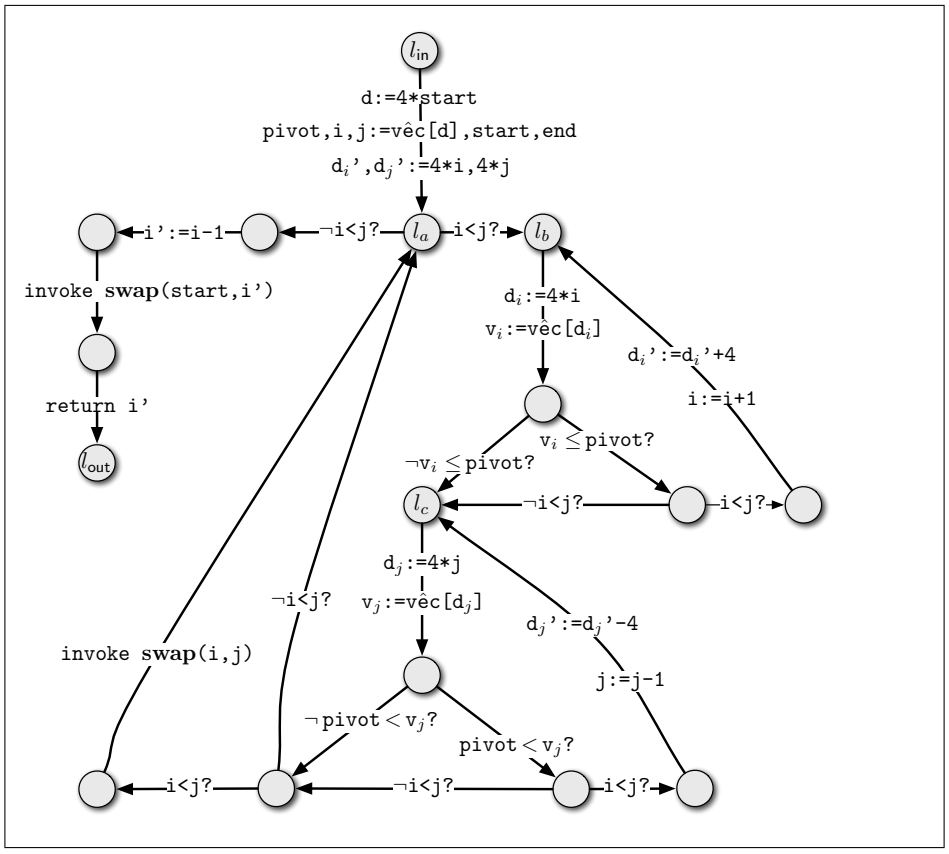


Fig. 18. Strength Reduction - First Step

In a second transformation step, we take advantage of the fresh variables $\mathtt{d}'_i$ and $\mathtt{d}'_j$ that has been introduced, replacing in the code the assignments $\mathtt{d}_i := 4{*}\mathtt{i}$ and $\mathtt{d}_j := 4{*}\mathtt{j}$ by $\mathtt{d}_i := \mathtt{d}'_i$ and $\mathtt{d}_j := \mathtt{d}'_j$. The transformation is shown in Figure 19.

In the following sections, we apply copy propagation and dead variable elimination in order to remove the occurrences of variables $\mathtt{d}_i$ and $\mathtt{d}_j$.

Comparison of verification conditions. One can see that the first transformation step does not alter the verification conditions. Indeed, an affectation of a fresh variable, i.e., a variable that appears neither in the program nor in the specification, does not affect the computation of verification conditions. Formally, from the definition of the function wpi, if x does not occur in φ, then $\mathsf{wpi}(x := e, \varphi) = \varphi$. In addition, one can see that if x does not occur in the program, then it is never introduced by the wpi function. That is, for very φ, if x does not occur in φ nor in ins, then x does not occur in $\mathsf{wpi}(\mathsf{ins}, \varphi)$. Let f and $\bar{f}$ denote the original and transformed procedure, respectively, the argu-

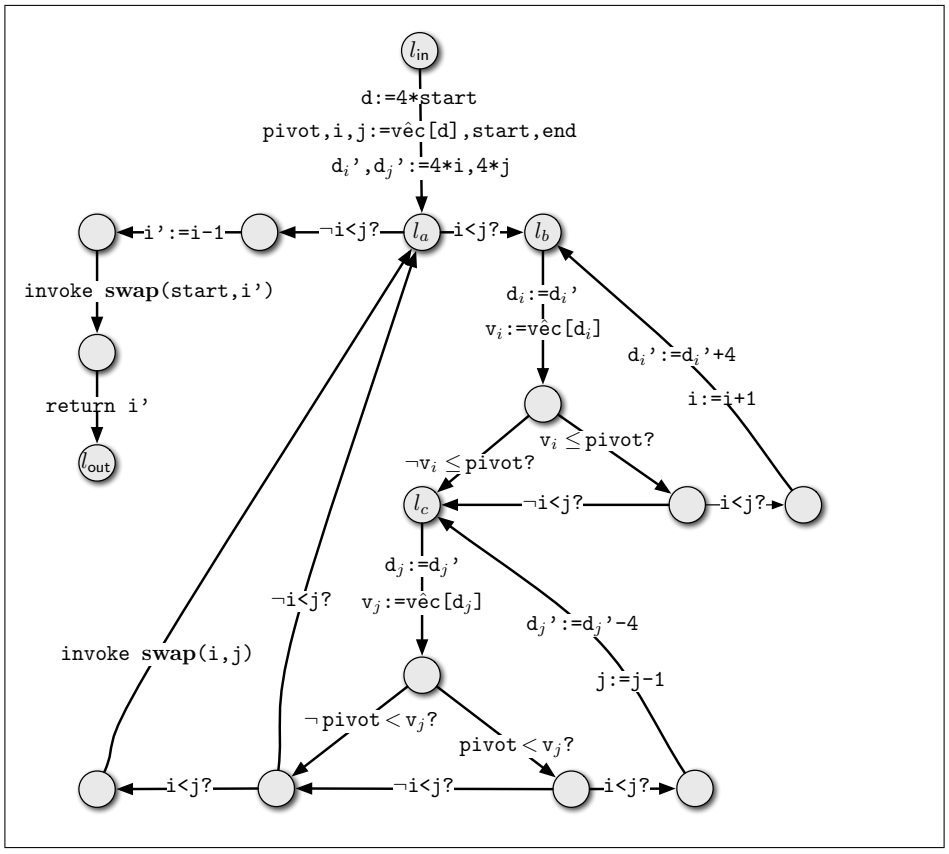


Fig. 19. Strength Reduction - Second Step

ment above implies that $\mathsf{wp}_f(l)$ and $\mathsf{wp}_{\bar{f}}(l)$ coincide syntactically for every label l. Therefore, proof obligations are preserved by this transformation step, and no transformation, neither of the specification nor the certificate, is needed.

However, one can see that certificates cannot be reused after the second transformation step, by simple comparison between the original and transformed program. Consider for instance the case of the verification of a simpler invariant at the loop headers pointed by labels l_b and l_c. Let ϕ stand for the invariant $\forall k \in \mathbb{N}.\ (\mathtt{start} \leq k < \mathtt{i} \Rightarrow \mathtt{vêc}[4 * \mathtt{i}] \leq \mathtt{pivot})$, and let $\mathsf{annot}(l_1) = \mathsf{annot}(l_4) = \phi$. Computing the proof obligation for label l_b for the program shown in Figure 18 returns the proof goal:

$$\phi \Rightarrow (\mathtt{vêc}[4 * \mathtt{i}] \leq \mathtt{pivot} \Rightarrow \mathtt{i} < \mathtt{j} \Rightarrow \phi[^{\mathtt{i}+1}/_{\mathtt{i}}])$$

In the other hand, computing the same proof obligation for the program shown in Figure 19 returns

$$\phi \Rightarrow (\mathtt{vêc}[\mathtt{d}_i'] \leq \mathtt{pivot} \Rightarrow \mathtt{i} < \mathtt{j} \Rightarrow \phi[^{\mathtt{i}+1}/_{\mathtt{i}}])$$

which is unprovable unless we assume as hypothesis the result of the analysis, i.e. the condition $d'_i = 4 * i$.

Transformation of the specification. In order to overcome the transformation of proof obligations, we propose first to strengthen the original specification to incorporate the result of the analysis that justifies the optimization. To simplify the exposition of this procedure, we split it in two steps. To that end, we first represent and certify the result of the static analysis in the underlying verification framework. Then, we merge this certified specification with the original procedure specification.

Consider $(\mathsf{true}, \mathsf{annot}_A, \mathsf{true})$ a procedure specification that represents the results of the analysis. One would like to generate a certificate for the specification $(\mathsf{Pre}, \mathsf{annot} \wedge \mathsf{annot}_A, \mathsf{Post})$ where $\mathsf{annot} \wedge \mathsf{annot}_A$ stands for the partial function such that $(\mathsf{annot} \wedge \mathsf{annot}_A)(l) \doteq \mathsf{annot}(l) \wedge \mathsf{annot}_A(l)$.

After providing a certificate for the specification $(\mathsf{true}, \mathsf{annot}_a, \mathsf{true})$ of the analysis result, we can integrate it with the certificate for the current specification $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$, as follows from the next result:

Lemma 9. *Let $s_1 = (\mathsf{Pre}_1, \mathsf{annot}_1, \mathsf{Post}_1)$ and $s_2 = (\mathsf{Pre}_2, \mathsf{annot}_2, \mathsf{Post}_2)$ be certified specifications of a procedure p. Then, both certificates can be merged to generate a certificate for the specification $s = (\mathsf{Pre}_1 \wedge \mathsf{Pre}_2, \mathsf{annot}_1 \wedge \mathsf{annot}_2, \mathsf{Post}_1 \wedge \mathsf{Post}_2)$.*

Proof. Let wp_{f_1}, wp_{f_2} and wp_f correspond to the weakest-precondition computation with specification s_1, s_2 and s, respectively. One can generate, by the induction principle induced by the definition of well-annotated programs, a certificate for the following goals:

$$\vdash \mathsf{wp}_{f_1}(l) \wedge \mathsf{wp}_{f_2}(l) \Rightarrow \mathsf{wp}_f(l)$$

and

$$\vdash \mathsf{wpi}_{f_1}(l) \wedge \mathsf{wpi}_{f_2}(l) \Rightarrow \mathsf{wpi}_f(l)$$

for every label l. This result follows from a proof of the distributivity of the predicate transformer wp w.r.t. conjunction. It should be clear, from the definition of $\mathsf{annot}_1 \wedge \mathsf{annot}_2$, that this is sufficient to certify the proof obligations corresponding to the result of merging the two specifications.

Certification of analysis results. In the example, for the second transformation step, we implicitly assume that the compiler has run a static analysis that determined that the condition $\mathtt{d}'_i = 4 * \mathtt{i}$ is valid at program label l_1. And similarly for the condition $\mathtt{d}'_j = 4 * \mathtt{j}$ at the program label l_4. Therefore, we assume the invariant specification $\mathsf{annot}_A(l)$ defined as $\mathtt{d}'_i = 4 * i \wedge \mathtt{d}'_j = 4 * j$, for $l \in \{l_a, l_b, l_c\}$. The first goal is to certify the specification $(\mathsf{true}, \mathsf{annot}_A, \mathsf{true})$ in the **partition** procedure. If we compute the verification conditions in order to

certify the result of the analysis we get verification goals such as:

$$\begin{array}{l}\mathsf{annot}_A(l_b) \Rightarrow \\ (\hat{\mathtt{vec}}[4 * \mathtt{i}] \leq \mathtt{pivot} \Rightarrow \mathtt{i} < \mathtt{j} \Rightarrow \mathtt{d}'_i + 4 = 4 * (\mathtt{i} + 1)) \wedge \\ (\hat{\mathtt{vec}}[4 * \mathtt{i}] \leq \mathtt{pivot} \Rightarrow \neg \mathtt{i} < \mathtt{j} \Rightarrow \mathsf{annot}_A(l_c)) \wedge \\ (\neg \hat{\mathtt{vec}}[4 * \mathtt{i}] \leq \mathtt{pivot} \Rightarrow \mathsf{annot}_A(l_c))\end{array}$$

One can see that to prove this goal is enough to perform arithmetic simplification and rewriting of equalities.

Transformation of the certificate. An essential requirement to translate the certificate is to provide a formal proof that states that predicate transformers of the replaced instructions are consistent with the original ones assuming valid the result of the analysis. More precisely, in the running example, we are interested in providing a formal proof, for every assertion ϕ, of the conditions $\mathsf{wpi}(\mathtt{d}_i := 4 * \mathtt{i}, \phi) \Rightarrow \mathsf{wpi}(\mathtt{d}_i := \mathtt{d}'_i, \phi)$ and $\mathsf{wpi}(\mathtt{d}_j := 4 * \mathtt{j}, \phi) \Rightarrow \mathsf{wpi}(\mathtt{d}_j := \mathtt{d}'_j, \phi)$ assuming as hypotheses the conditions $\mathtt{d}'_i = 4 * \mathtt{i}$ and $\mathtt{d}'_j = 4 * \mathtt{j}$, respectively. In our setting, this corresponds to an application of the operation $\mathsf{elim}_=$ of the proof algebra. The following result states that this, together with the certificate of the analysis, is sufficient to generate a new certificate corresponding to the transformed program.

Lemma 10. *Let f and $\bar{f}$ stand for the original and transformed program, respectively. Suppose that* $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$ *is a certified specification for f and that the result of the analysis* $(\mathsf{true}, \mathsf{annot}_A, \mathsf{true})$ *is certified. Assume, the $\mathcal{N}_f = \mathcal{N}_{\bar{f}}$, $\mathcal{E}_f = \mathcal{E}_{\bar{f}}$ and for every edge $\langle l, l' \rangle \in \mathcal{E}_f$ s.t. $G_f[\langle l, l' \rangle] \neq G_{\bar{f}}[\langle l, l' \rangle]$, and any assertion φ, that we have a certificate* justif *for the following goal:*

$$\vdash \mathsf{wpi}(G[\langle l, l' \rangle], \varphi) \wedge \mathsf{annot}_A(l) \Rightarrow \mathsf{wpi}(G_{\bar{f}}[\langle l, l' \rangle], \varphi)$$

Then, one can generate a certificate for the transformed program $\bar{f}$ with specification $(\mathsf{Pre}, \mathsf{annot} \wedge \mathsf{annot}_A, \mathsf{Post})$.

Proof. Assume for simplicity that annot_A *is a total function. From the certificate* justif*, and by the induction principle associated to well-annotated programs, one can generate certificates for the following goals:*

$$\vdash \mathsf{wp}_f(l) \wedge \mathsf{annot}_A(l) \Rightarrow \mathsf{wp}_{\bar{f}}(l)$$

and

$$\vdash \mathsf{wpi}(G_f[\langle l, l' \rangle], \mathsf{wp}_f(l')) \wedge \mathsf{annot}_A(l) \Rightarrow \mathsf{wpi}(G_{\bar{f}}[\langle l, l' \rangle], \mathsf{wp}_{\bar{f}}(l'))$$

for every program label l and edge $\langle l, l' \rangle$. Recall that proof obligations have the form $\mathsf{wp}_{\bar{f}}(l) \Rightarrow \bigwedge_{l' \in \mathsf{succ}(l)} \mathsf{wpi}(G_{\bar{f}}[\langle l, l' \rangle], \mathsf{wp}_{\bar{f}}(l'))$, and that $\mathsf{wp}_{\bar{f}}(l) \doteq \mathsf{wp}_f(l) \wedge \mathsf{annot}_A(l)$. It is sufficient then to provide certificates for

$$\vdash \mathsf{wp}_f(l) \Rightarrow \mathsf{wpi}(G_f[\langle l, l' \rangle], \mathsf{wp}_f(l'))$$

and

$$\vdash \mathsf{wp}_A(l) \Rightarrow \mathsf{wpi}(G_f[\langle l, l' \rangle], \mathsf{wp}_A(l'))$$

where $\mathsf{wp}_A(l)$ *is computed with the result of the analysis as specification. The certificates required above correspond exactly to the original certificates and the certificates of the result of the analysis.*

5.4 Copy Propagation

Description. Copy propagation is a simple compiler optimization that consists in replacing some occurrences of a program variable by a variable that holds the same value. In general, for a sequence of statements of the form

$$\mathtt{x} := \mathtt{y}\,;\, c_1\,;\, c_2$$

the transformation replaces any occurrence of the variable $\mathtt{x}$ by $\mathtt{y}$ in c_2, as long as neither $\mathtt{x}$ nor $\mathtt{y}$ gets modified by one of the instructions in c_1. This is a cleanup transformation, intended to reduce the set of used registers and simplifying the transformed code resulting from a previous optimization. In addition, it is an enabling transformation, that opens the door to further optimization opportunities.

Transformed running example. In the previous transformation, we have reduced the operation strength of assignments of the form $\mathtt{d}_i := 4 * \mathtt{i}$ by a substitution for a copy operation. In principle, there is no reason to preserve both variables $\mathtt{d}_i$ and $\mathtt{d}'_i$, nor both of variables $\mathtt{d}_j$ and $\mathtt{d}'_j$. We proceed then by substituting the occurrences of the variables $\mathtt{d}_i$ and $\mathtt{d}_j$ by $\mathtt{d}'_i$ and $\mathtt{d}'_j$, respectively, as shown in Figure 20.

Comparison of verification conditions In this particular example, after computing the verification conditions, one can easily see that they are preserved. Hence, no certificate translation is needed in this case.

In general, verification conditions do not coincide after the transformation. However, one can prove that they only differ on some variable renaming. Then, depending on the underlying notion of certificates, it is possible that no transformation is needed at all, or with a minor variable renaming in the representation of the certificate.

5.5 Dead Variable Elimination

Description. Dead variable elimination is a compiler transformation that removes assignments to variables that are never used. The occurrence of such assignments are mainly the result of earlier program optimizations. For instance, in the following transformations

$$\begin{array}{lcll} \mathtt{y} := 0 & & \mathtt{y} := 0 & & \mathsf{nop} \\ \mathtt{x} := \mathtt{y} * \mathtt{z} & \longrightarrow & \mathtt{x} := 0 & \longrightarrow & \mathsf{nop} \\ \mathtt{r} := f(\mathtt{x}) & & \mathtt{r} := f(0) & & \mathtt{r} := f(0) \end{array}$$

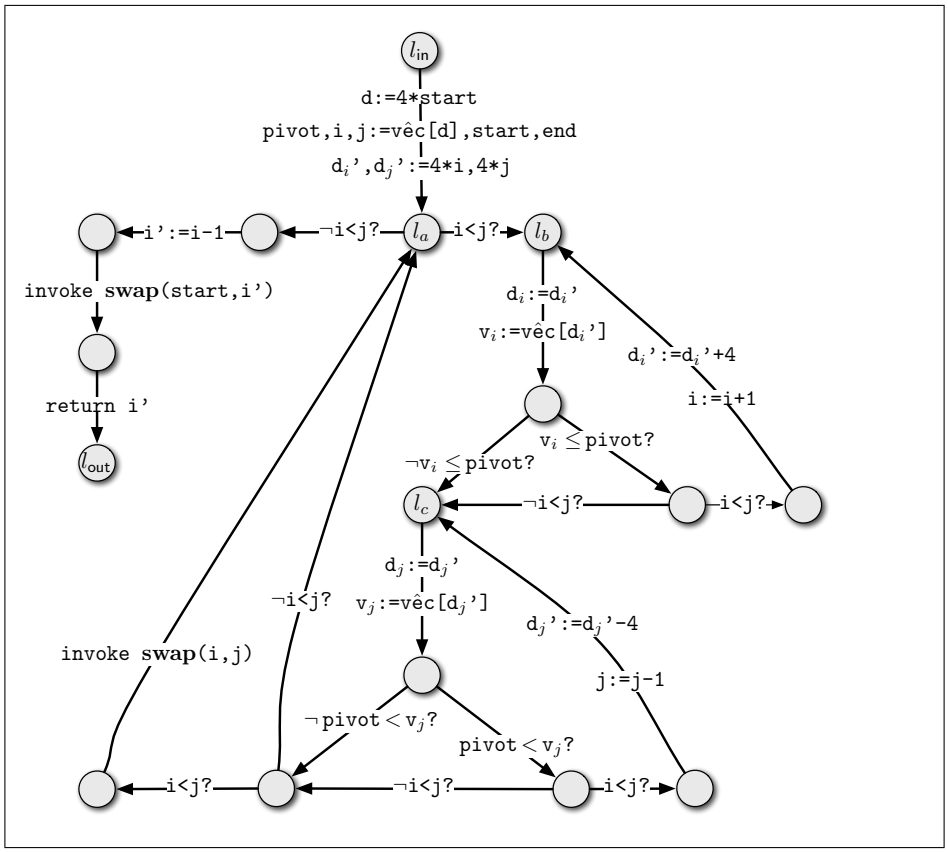


Fig. 20. Copy Propagation Transformation

the second sequence of instructions is the result of propagating the constant values held by the variables `y` and `x`. Since neither `x` nor `y` is used in the rest of the program, the second transformation performs dead variable elimination removing the assignments to `x` and `y`.

There are two main improvements as a consequence of dead variable elimination. First, the unnecessary computation of the right hand side expression is removed, reducing execution time and program size. Second, it reduces the number of pseudo-variables that are used, which facilitates register allocation in the last compilation steps.

Commonly, the notion of variable liveness formalizes the situation in which the value of a variable is not needed in the future. We say that a variable is read at a program edge l if it appears at an edge $\langle l, l' \rangle$ in the right hand side of an assignment, as parameter in a function call, in a return statement or in a conditional expression. We say that a variable `x` is live at a certain program

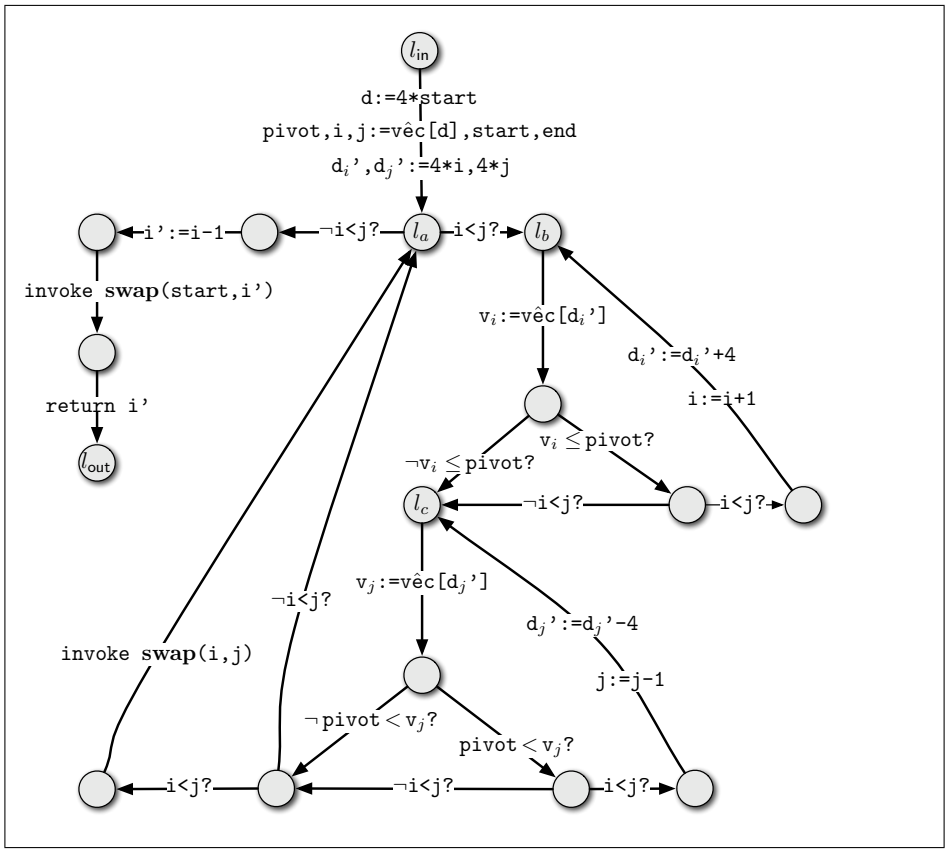


Fig. 21. Dead Variable Elimination

label l if there is a program path from label l to a program point that reads `x` and in which `x` is not updated.

Dead variable elimination consists in removing every assignment to a variable that is not live at the following program point.

Transformed running example. In the code at the right of Figure 20, the values assigned to the variables $\mathtt{d}_i$ and $\mathtt{d}_j$ are never used. The transformation, shown in Figure 21, takes the result of the previous optimization and removes the assignments to $\mathtt{d}_i$ and $\mathtt{d}_j$.

Comparison of verification conditions. As can be seen after computing the verification conditions over the original and transformed program, they are preserved. Hence, there is no need to transform the specification nor the certificates.

However, it is not always the case that verification conditions are preserved. In fact, in general, they can become unprovable due to the occurrence of dead

variables in the loop invariants. After some instructions are sliced-out of the program, even though the input/output semantics is preserved, the conditions over dead variables at the intermediate program points may not be satisfied. The following example illustrates this situation:

$$\begin{array}{lcl} \mathtt{x} := \mathtt{z} & & \mathbf{nop} \\ \{\mathtt{x} = \mathtt{z}\} & \longrightarrow & \{\mathtt{x} = \mathtt{z}\} \\ \mathtt{y} := f(\mathtt{x}) & & \mathtt{y} := f(\mathtt{z}) \end{array}$$

After propagating the variable $\mathtt{z}$ to the function call, the assignment $\mathtt{x} := \mathtt{z}$ is not needed anymore and then it can be removed. The problem is that after removing the assignment to the dead variable, the condition $\mathtt{x} = \mathtt{z}$ becomes invalid. As a second example, consider the tuple $(\mathsf{true}, \mathsf{annot}, \mathsf{true})$ as a specification for the procedure **partition**, where $\mathsf{annot}(l_a) = \mathsf{annot}(l_b) = \mathsf{annot}(l_c) = \varphi$ and $\varphi \doteq \mathtt{d}_i = \mathtt{d}'_i \wedge \mathtt{d}'_i = 4 * \mathtt{i}$. After introduction of logical implication, a fragment of the proof obligation at label l_b for the program before the optimization is:

$$\Gamma \vdash \mathsf{annot}(l_b)[{}^{\mathtt{d}'_i+4}/_{\mathtt{d}'_i}][{}^{\mathtt{i}+1}/_{\mathtt{i}}][{}^{\mathtt{d}'_i}/_{\mathtt{d}_i}]$$

where $\Gamma \doteq \{\mathsf{annot}(l_b), \hat{\mathtt{vec}}[\mathtt{d}'_i] \leq \mathtt{pivot}, \mathtt{i} < \mathtt{j}\}$. The proof obligation, at the same label l_b, computed after performing dead variable elimination becomes:

$$\Gamma \vdash \mathsf{annot}(l_b)[{}^{d'_i+4}/_{\mathtt{d}'_i}][{}^{\mathtt{i}+1}/_{\mathtt{i}}]$$

which is clearly unprovable because of the removal of the instruction $\mathtt{d}_i := \mathtt{d}'_i$.

A solution for this problem consists in weakening the original specification to remove the occurrences of dead variables at the intermediate assertions. More precisely, one can show that it is feasible to quantify existentially the dead variables that occurs at the intermediate annotations, removing dead assignments, and transforming the original certificates. We have developed this method in the context of an abstract interpretation framework [4].

An alternative approach consists in renaming each dead variable that appears in an assertion to its corresponding ghost variable. In this case, assignments to dead variables are not removed but replaced by assignments to ghost variables (namely ghost assignments). Proof obligations coincide up to renaming of dead variables to ghost variables. Since ghost assignments are part of the specification and thus never executed, they can be sliced out by the code client after the verification process and prior to its execution. A more detailed account of this technique can be found in a previous work by Barthe et al. [3].

5.6 Loop Unrolling

Description. Loop unrolling is a compiler transformation that duplicates code by unfolding the execution of a loop body. The transformation does not necessarily improve the code execution performance, it is rather an enabling transformation, i.e., it prepares the code for further compiler opportunities.

There are several variants of this transformation. In this section, we consider a transformation that prefixes a loop with a single sequential execution of its

body (under the guard of the loop, in order to preserve the program semantics). Consider for instance a program of the form **while** b **do** c, the result of unrolling the loop in this program is **if** b **then** c; **while** b **do** c. We define the loop unrolling transformation as a particular instance of a more general notion of node duplication, formalized by the following definition:

Definition 8 (Node replication). *A program* $\langle \mathcal{N} \cup \mathcal{N}^+, \mathcal{E}^+, G^+ \rangle$ *is the result of replicating nodes of program* $\langle \mathcal{N}, \mathcal{E}, G \rangle$ *if*

- $\mathcal{N}^+ \subseteq \{l^+ \mid l \in \mathcal{N}\}$;
- *for every* $l_1, l_2 \in \mathcal{N}$, *if* $\langle l_1^+, l_2 \rangle$, $\langle l_1, l_2^+ \rangle$, *or* $\langle l_1^+, l_2^+ \rangle$ *is in* $\mathcal{E}^+$ *then* $\langle l_1, l_2 \rangle$ *is in* $\mathcal{E}$, *i.e., subgraph duplication preserves the structure; and*
- *for every* $l_1, l_2 \in \mathcal{N}$, *if* $e \in \{\langle l_1^+, l_2 \rangle, \langle l_1, l_2^+ \rangle, \langle l_1^+, l_2^+ \rangle\}$ *then* $G^+[e] = G[\langle l_1, l_2 \rangle]$.

Transformed running example. Consider the procedure **partition** in the context of the whole running example. Assuming that the procedure **partition** is not called from any program point outside the body of the procedure **quicksort**, we know that the condition `i < j` always holds just before the execution of the body. Consequently, the loop is executed at least once, for every initial execution state, and one can take advantage of this fact to search for further optimizations. In this section, we unroll one step of the execution of the outer loop statement. In the following section, we optimize the duplicated instance of the loop body.

Figure 22 shows the result of unrolling the outer loop of the procedure **partition**. In the figure, the subgraph corresponding to the loop body is duplicated and placed immediately before the loop header. The evaluation of the loop guard is included in the duplicated code, in order to ensure preservation of the program semantics. Notice that the last duplicated node jumps to the original loop header (i.e., node l_a), instead of jumping backwards to the duplicated evaluation of the guard (i.e., node l'_a), and thus avoiding the re-entrance inside the duplicated code.

Transformation of the certificate. In general, as one can see, dealing with this transformation is simple since proof obligations are not modified, but duplicated.

Let $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$ stand for the procedure specification previous to the application of loop unrolling. Consider an invariant specification annot' that extends annot in the set of duplicated labels. That is, $\mathsf{annot}'(l) = \mathsf{annot}(l)$ for $l \in \mathsf{dom}(\mathsf{annot})$, and $\mathsf{annot}'(l'_a)$, $\mathsf{annot}'(l'_b)$ and $\mathsf{annot}'(l'_c)$ equal to $\mathsf{annot}(l_a)$, $\mathsf{annot}(l_b)$ and $\mathsf{annot}(l_c)$, respectively.

One can see that the original verification conditions, i.e., those at program points in $\mathsf{dom}(\mathsf{annot})$, are not modified. However, new verification conditions are introduced at the annotated program points that are duplicated: l'_a, l'_b ad l'_c. Since the code involved in the computation of the proof obligations at labels l'_b and l'_c preserves the same structure of the original code, then, as one can see, proof obligations are equal to the original proof obligations at l_b and l_c. The proof

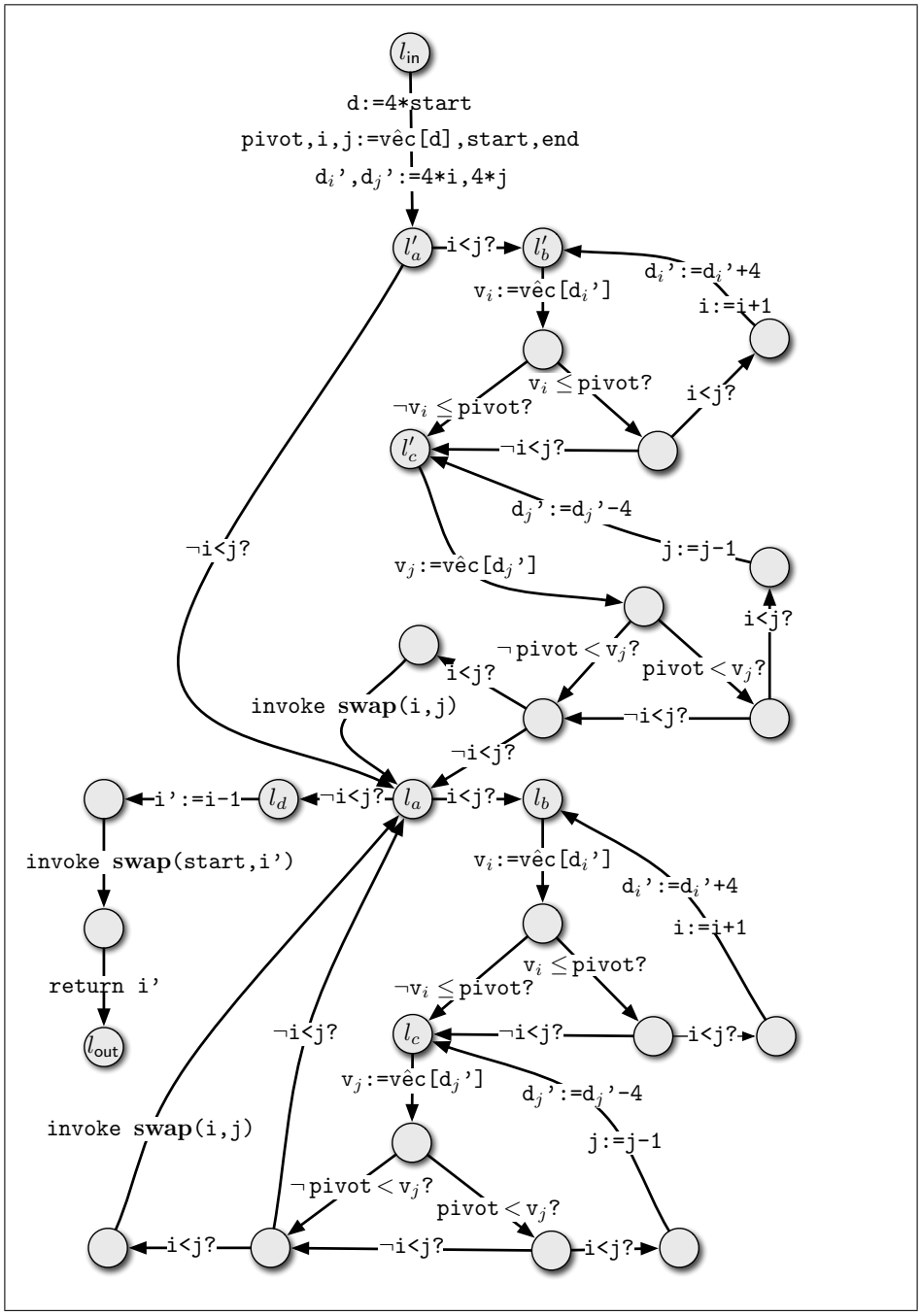


Fig. 22. Loop Unrolling

obligation at label l'_a do not coincide with the original proof obligation at label l_a. The proof obligation at label l_a, after the application of the optimizations is:

$$\vdash \mathsf{annot}(l_a) \Rightarrow (\mathtt{i} < \mathtt{j} \Rightarrow \mathsf{annot}(l_b)) \wedge (\neg \mathtt{i} < \mathtt{j} \Rightarrow \mathsf{wp}(l_d))$$

If we compute the proof obligation at label l'_a we get

$$\vdash \mathsf{annot}(l'_a) \Rightarrow (\mathtt{i} < \mathtt{j} \Rightarrow \mathsf{annot}(l'_b)) \wedge (\neg \mathtt{i} < \mathtt{j} \Rightarrow \mathsf{annot}(l_a))$$

which by definition is equal to

$$\vdash \mathsf{annot}(l_a) \Rightarrow (\mathtt{i} < \mathtt{j} \Rightarrow \mathsf{annot}(l_b)) \wedge (\neg \mathtt{i} < \mathtt{j} \Rightarrow \mathsf{annot}(l_a))$$

Although the new proof obligation at label l'_a is clearly different, it is still trivial to discharge.

The following result generalizes certificate translation after the application of a loop unrolling transformation.

Lemma 11. *Let $p^+ = \langle \mathcal{N} \cup \mathcal{N}^+, \mathcal{E}^+, G^+ \rangle$ be the result of duplicating some of the nodes of the procedure $p = \langle \mathcal{N}, \mathcal{E}, G \rangle$. Let l^+ denote a label in $\mathcal{N}^+$, i.e., a label such that $l \in \mathcal{N}$. Let $\langle \mathsf{Pre}, \mathsf{annot}^+, \mathsf{Post} \rangle$ be the specification of p^+, where $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post} \rangle$ is the specification of p, and annot^+ extends annot to $\mathcal{N}^+$, defining $\mathsf{annot}^+(l^+)$ as $\mathsf{annot}(l)$. Then, one can construct, for $\bar{l} \in \{l, l^+\}$ a certificate for the following goal:*

$$c: \quad \vdash \mathsf{wp}_p(l) \Rightarrow \mathsf{wp}_{p^+}(\bar{l})$$

Since $\mathsf{wp}_p(l)$ coincides with $\mathsf{wp}_{p^+}(\bar{l})$ for every $l \in \mathsf{dom}(\mathsf{annot})$, it follows that one can generate a certificate for the transformed program from the certificate of the goal above and the original certificates.

A more general result is part of a development of certificate translation in the context of an abstract interpretation setting [4].

5.7 Redundant Conditional Elimination

Description. Redundant conditional elimination is a program optimization that removes conditional branching that can be predicted statically. First, an automated analysis gathers information about the program variables along the control flow paths of the program. Then, in the basis of the result of the analysis, a transformation step removes the evaluation of conditional expressions that are inferred to be always valid (or always invalid), and conditional jumps are modified in accordance. In addition, the instructions that become unreachable, and then non-executable, may also be removed. In the following example

$\mathtt{y} := \mathtt{z} * \mathtt{z}$;
while $(\mathtt{x} < \mathtt{y})$ **do** c_1;
if $(\mathtt{x} < 0)$ **then** c_2;

if the statement c_1 does not modify the variable $\mathtt{y}$, the analysis may infer that the condition $\mathtt{x} \geq 0$ holds right after the execution of the body of the loop. In that case, we know it is safe to remove the statement **if** $(\mathtt{x} < 0)$ **then** c_2, since it will never be executed. In the rest of the section we assume the static analysis is capable of discovering relational properties on the values of program variables.

Transformed running example. Consider the code in Figure 22, i.e., after unrolling one execution of the loop body in the running example, executing in the context of the procedure **quicksort**. Notice from Figure 1 that the only invocation of the procedure **partition** is performed with parameters `start` and `end`, and under the guard `start` $<$ `end`. One would expect thus an inter-procedural analysis to statically infer that the condition `i` $<$ `j` always holds at the program point with label l'. Consequently, one of the branches at l'_a is always taken and then it is safe to remove one of the conditional edges. The transformation then removes the branch $\neg$(`i` $<$ `j`) at node l'_a to jump unconditionally to l'_b. The transformed RTL code for the procedure **partition** can be found in Figure 23.

Comparison of verification conditions. Inspecting the transformed code in Figure 23, one can see that the proof obligation at label l'_a is the only one that is affected by the transformation. The original proof obligation at label l'_a is

$$\vdash \mathsf{annot}(l'_a) \Rightarrow (\texttt{i} < \texttt{j} \Rightarrow \mathsf{annot}(l'_b)) \wedge (\neg \texttt{i} < \texttt{j} \Rightarrow \mathsf{annot}(l_a))$$

whereas, after the transformation, the proof obligation becomes

$$\vdash \mathsf{annot}(l'_a) \Rightarrow \mathsf{annot}(l'_b)$$

In this particular example, the new proof obligation can be still discharged, since by definition both $\mathsf{annot}(l'_a)$ and $\mathsf{annot}(l'_b)$ are equal to $\mathsf{annot}(l_a)$. However, that is not generally the case, since $\mathsf{annot}(l'_b)$ may be distinct to $\mathsf{annot}(l_a)$ and thus the condition `i` $<$ `j` may be needed as hypothesis to prove the implication $\mathsf{annot}(l_a) \Rightarrow \mathsf{annot}(l'_b)$. In the rest of this section, we generalize certificate transformation in the presence of redundant conditional elimination.

Transformation of the specification. To deal with this transformation, we proceed by incorporating the result of the analysis as a strengthening of the original invariants. As explained in Section 5.3, this process entails first providing a certificate of the result of the analysis represented in the logic of the verification setting. To that end, we must rely on the existence of certifying analyzers, an extension of standard analyses that provide, in addition of an analysis result, a certificate of its validity.

Certification of analysis results. In Section 5.3, we have considered an intra-procedural analysis and, thus, it was sufficient to consider a specification of the result of the analysis at intermediate program points of the procedure **partition**. However, in this case, since we are considering an inter-procedural analysis, we need to extend the scope of the certifying analyzer. More precisely, we also need to transform (strengthen) the precondition of the procedure **partition**. Since this affects the computation of verification conditions on the code that invokes this procedure, we need to consider the verification on the result of the analysis in this code as well. In our running example, we must provide a specification

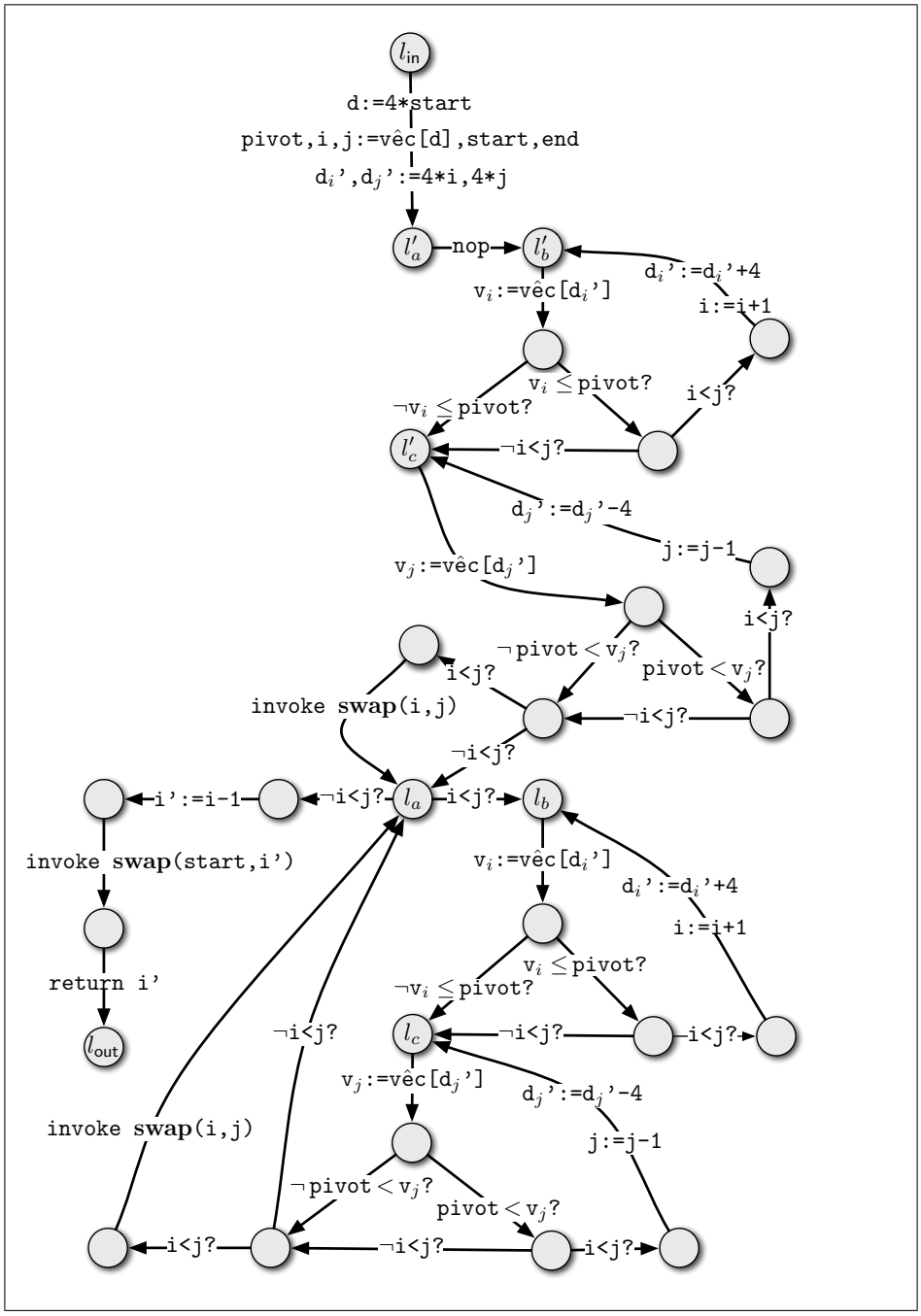


Fig. 23. Redundant Conditional Elimination

for the result of the analysis for all the procedures. Let $(\mathsf{true}, \emptyset, \mathsf{true})$ be the specification of the result of the analysis for the procedures **quicksort** and **swap**. Let $(\mathtt{start} < \mathtt{end}, \mathsf{annot}_a, \mathsf{true})$ the specification of the result of the analysis for the procedure **partition**, where $\mathsf{annot}_a(l'_a) \doteq \mathtt{i} < \mathtt{j}$ and $\mathsf{annot}_a(k) \doteq \mathsf{true}$ for any other label k.

As in Section 5.3, providing a certificate for the result of the analysis is straightforward due to the simplicity of verification conditions. To illustrate this, consider for instance the proof obligation that is computed for the procedure **quicksort**:

$$\mathsf{true} \Rightarrow \mathtt{start} < \mathtt{end} \Rightarrow (\mathtt{start} < \mathtt{end} \wedge \forall \mathtt{res}.(\mathsf{true} \Rightarrow \varphi))$$

where φ stands for $\mathsf{true} \wedge \forall \mathtt{res}(\mathsf{true} \Rightarrow \mathsf{true} \wedge \forall \mathtt{res}(\mathsf{true} \Rightarrow \mathsf{true}))$. It is clearly easy to discharge. The proof obligations computed at the starting point of the procedure **partition** and at the label in which the program is transformed are:

$$\vdash \mathtt{start} < \mathtt{end} \Rightarrow \mathtt{start} < \mathtt{end}$$

and

$$\vdash \mathtt{i} < \mathtt{j} \Rightarrow \mathtt{i} < \mathtt{j} \Rightarrow \mathsf{true}$$

respectively.

Transformation of the certificate. The transformation of the certificate can be performed by the general technique shown in Section 5.3. Once the result of the analysis is certified, we can incorporate it to the current specification and generate a certificate for this new specification. Let $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$ stand for the current specification of the procedure **partition**, we define the transformed specification as $(\mathsf{Pre} \wedge \mathtt{start} < \mathtt{end}, \mathsf{annot}', \mathsf{Post})$, where $\mathsf{annot}'(l'_a) \doteq \mathsf{annot}(l') \wedge \mathtt{i} < \mathtt{j}$ and $\mathsf{annot}'(k) = \mathsf{annot}(k)$ for any other label k. We know that we can generate a certificate for the extended specification from Lemma 9. Let p and $\hat{p}$ stand for the original and transformed program, and notice that $\mathsf{succ}_{\hat{p}}(l'_a) \subseteq \mathsf{succ}_p(l'_a)$. As in Section 5.3, in order to generate a certificate for the transformed program, we need to provide a formal proof of the following goal:

$$\vdash \bigwedge_{l_i \in \mathsf{succ}_p(l'_a)} \mathsf{wpi}_p(G_p[\langle l'_a, l_i \rangle], \varphi_i) \wedge \mathsf{annot}'(l'_a) \Rightarrow \bigwedge_{l_i \in \mathsf{succ}_{\hat{p}}(l'_a)} \mathsf{wpi}_{\hat{p}}(G_{\hat{p}}[\langle l'_a, l_i \rangle], \varphi_i)$$

for any φ, which in this case is defined as

$$\vdash (\mathtt{i} < \mathtt{j} \Rightarrow \varphi_t) \wedge (\neg \mathtt{i} < \mathtt{j} \Rightarrow \varphi_f) \wedge \mathtt{i} < \mathtt{j} \Rightarrow \varphi_t$$

From Lemma 10, a certificate for the goal above is sufficient to provide a certificate for the transformed program in the general case.

5.8 Stack-based Code Generation

In this section we consider the last compilation phase, in which the intermediate program representation is transformed into the final interpretable stack-based code. We introduce briefly the programming language, its semantics and the underlying verification framework. We provide a definition of the compiler, that transforms an RTL graph to a sequence of labeled stack-based instructions. Finally, we show that verification conditions are preserved, and thus no transformation of the certificate is needed.

The transformation not only produces a linearized version of an RTL graph, but replaces RTL with stack-based instructions. Stack-based computation relies on instructions that put, remove and modify values stored in the stack. As with RTL programs, a stack-based program is composed of a set of procedures. Each procedure p consists of a set of formal parameters and a list of labeled instructions from the set described in Figure 24. In the figure, *sig* is the signature of the invoked procedure, consisting of a procedure identifier and the number of arguments it takes from the stack. For notational convenience, we let a partial function G_p map program labels to instructions. For every instruction with only one predecessor, we omit its label. For an instruction at label l with a single successor we let $l + 1$ stand for the label of the next instruction. For a label l such that $G[l] = \mathbf{cjmp} \bowtie l_t, l_f$, $\mathsf{succ}(l)$ is defined as $\{l_t, l_f\}$, for $G[l] = \mathbf{jmp}\ l'$, $\mathsf{succ}(l) = \{l'\}$ and if $G[l] = \mathbf{return}$ then $\mathsf{succ} = \emptyset$. For any other case, $\mathsf{succ}(l) = \{l + 1\}$.

$$
\begin{array}{rl}
\mathit{ins} ::= & \mathbf{prim}\ \oplus \\
\mid & \mathbf{push}\ n \\
\mid & \mathbf{load}\ x \\
\mid & \mathbf{store}\ x \\
\mid & \mathbf{aload}\ a \\
\mid & \mathbf{astore}\ a \\
\mid & \mathbf{nop} \\
\mid & \mathbf{jmp}\ l \\
\mid & \mathbf{cjmp} \bowtie l, l \\
\mid & \mathbf{invoke}\ \mathit{sig} \\
\mid & \mathbf{return}
\end{array}
$$

Fig. 24. Stack-based Instruction Set

A stack-based program is well-formed if the control-flow representation of every procedure is a closed graph. Formally, for every procedure p, and $l \in \mathsf{dom}(G_p)$, we have that $\mathsf{succ}(l) \subseteq \mathsf{dom}(G_p)$. As with RTL programs we assume the existence of an initial label l_{in}. As in previous sections, an execution environment is composed of a global array state in $\Sigma_{\mathcal{A}}$ and a local scalar state in $\Sigma_{\mathcal{V}}$. The

semantics of well-formed programs is defined in Figure 25 by a relation $\rightsquigarrow \subseteq \mathcal{L} \times \mathit{Stack} \times \Sigma \rightarrow \mathit{Stack} \times \Sigma$. It differs from the semantics of RTL programs in the computation of expressions, argument passing and value returning.

Verification setting. The specification language is slightly modified in order to reason about stack expressions. We use the special variable $\mathbf{s}$ to denote the operand stack. The expression $\mathbf{s}[0]$ denotes the top element of the stack $\mathbf{s}$ and the expression $\uparrow \mathbf{s}$ denotes the stack after removing the top element from $\mathbf{s}$. We assume these expressions to be immediately reduced when introduced by variable substitution, according to the rules $(e :: \mathbf{s})[0] = e$ and $\uparrow (e :: \mathbf{s}) = \mathbf{s}$. A specification for a stack-based procedure is a tuple $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$, where annot is a partial mapping from labels to assertions that may contain stack expressions. Pre and Post do not contain stack expressions.

For the proof obligations to be computable, we require the sequence of instructions to be well-annotated, i.e., that every cycle of the control-flow graph contains at least one annotated label.

From a well-annotated stack-based program a VCgen extracts a set of proof obligations from each of its procedures, by using the functions wp and wpi defined in Figure 26:

$$\begin{aligned}\mathsf{po}(p) = \{&\mathsf{Pre}_p \Rightarrow \mathsf{wp}_p(l_{\mathsf{in}})[{}^{A}/_{A^\star}]\} \cup \\ &\{\mathsf{annot}_p(l) \Rightarrow \mathsf{wpi}_p(l) \mid l \in \mathsf{dom}(\mathsf{annot}_p)\}\end{aligned}$$

where A represents the set of array variables that may get modified by p. For simplicity, we overload the predicate transformers wp and wpi to be defined over both RTL and stack-based instructions. When a label l has only one successor, this is denoted $l + 1$.

The following result states that the verification framework above is sound with respect to the program semantics.

Lemma 12 (VCgen Soundness). *Consider a well-annotated stack-based program P. Assume that for every procedure p, $\mathsf{po}(p)$ is a valid set of proof obligations. Then, for every procedure p, if $\langle l_{\mathsf{in}}, [], \sigma\rangle \rightsquigarrow_p \langle n :: \mathbf{s}, \sigma'\rangle$ then $\models [\sigma' : \mathtt{res} \mapsto n] : \mathsf{Post}$.*

Compilation. In this section, we define a simple compiler that transforms an RTL program representation into the stack-based code described above. The transformation can be seen as the last step of a compiler from a simple imperative language described in Section 3, to the final executable code. The compilation is defined by a function $\mathcal{C}$ that maps an RTL graph into a semantically equivalent sequence of instructions that manipulates the values stored in the execution stack. The definition of this compilation function can be found in Figure 27.

Since the structure of the code is preserved it follows that a well-formed RTL program is compiled into a well-formed stack-based program. Furthermore, if the specification is preserved, a well-annotated RTL program is compiled into a well-annotated stack-based program.

$$\frac{G_p[l] = \textbf{return}}{\langle l, n :: \mathbf{s}, \sigma\rangle \rightsquigarrow_p \langle n, \sigma_{\mathcal{A}}\rangle} \qquad \frac{G_p[l] = \textbf{nop} \qquad \langle l+1, \mathbf{s}, \sigma\rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, \mathbf{s}, \sigma\rangle \rightsquigarrow_p s}$$

$$\frac{G_p[l] = v := \textbf{invoke}\ q\ k \qquad \langle l_{\text{in}}, [], \langle [\vec{x}_q \mapsto \vec{n}], \sigma_{\mathcal{A}}\rangle\rangle \rightsquigarrow_q \langle m :: \mathbf{s}', \langle \sigma'_{\mathcal{V}}, \sigma'_{\mathcal{A}}\rangle\rangle \qquad \langle l+1, m :: \mathbf{s}, \langle \sigma_{\mathcal{V}}, \sigma'_{\mathcal{A}}\rangle\rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, \vec{n} :: \mathbf{s}, \sigma\rangle \rightsquigarrow_p s}$$

$$\frac{G_p[l] = \textbf{jmp}\ l' \qquad \langle l', \mathbf{s}, \sigma\rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, \mathbf{s}, \sigma\rangle \rightsquigarrow_p s}$$

$$\frac{G_p[l] = \textbf{cjmp} \bowtie l_t, l_f \qquad n_1 \bowtie n_2 \qquad \langle l_t, \mathbf{s}, \sigma\rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, n_1 :: n_2 :: \mathbf{s}, \sigma\rangle \rightsquigarrow_p s}$$

$$\frac{G_p[l] = \textbf{cjmp} \bowtie l_t, l_f \qquad \neg(n_1 \bowtie n_2) \qquad \langle l_f, \mathbf{s}, \sigma\rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, n_1 :: n_2 :: \mathbf{s}, \sigma\rangle \rightsquigarrow_p s}$$

$$\frac{G_p[l] = \textbf{prim} \oplus \qquad \langle l+1, n_1 \oplus n_2 :: \mathbf{s}, \sigma\rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, n_1 :: n_2 :: \mathbf{s}, \sigma\rangle \rightsquigarrow_p s}$$

$$\frac{G_p[l] = \textbf{push}\ n \qquad \langle l+1, n :: \mathbf{s}, \sigma\rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, \mathbf{s}, \sigma\rangle \rightsquigarrow_p s}$$

$$\frac{G_p[l] = \textbf{load}\ x \qquad \langle l+1, \sigma_{\mathcal{V}}\, x :: \mathbf{s}, \sigma\rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, \mathbf{s}, \sigma\rangle \rightsquigarrow_p s}$$

$$\frac{G_p[l] = \textbf{store}\ x \qquad \langle l+1, \mathbf{s}, [\sigma : x \mapsto n]\rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, n :: \mathbf{s}, \sigma\rangle \rightsquigarrow_p s}$$

$$\frac{G_p[l] = \textbf{aload}\ a \qquad 0 \leq i < |a| \qquad \langle l+1, \sigma_{\mathcal{A}}\, a\, i :: \mathbf{s}, \sigma\rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, i :: \mathbf{s}, \sigma\rangle \rightsquigarrow_p s}$$

$$\frac{G_p[l] = \textbf{astore}\ a \qquad 0 \leq i < |a| \qquad \langle l+1, \mathbf{s}, [\sigma : a \mapsto [a : i \mapsto n]]\rangle \rightsquigarrow_p s \qquad s \in \Sigma_F}{\langle l, i :: n :: \mathbf{s}, \sigma\rangle \rightsquigarrow_p s}$$

Fig. 25. Semantics of Stack-based Programs

$$
\begin{array}{ll}
G[l] = \textbf{nop} & \mathsf{wpi}(l) = \mathsf{wp}(l+1) \\
G[l] = \textbf{prim}\ \oplus & \mathsf{wpi}(l) = \mathsf{wp}(l+1)[^{\mathbf{s}[0]\oplus\mathbf{s}[1] :: \uparrow^2\mathbf{s}}/_{\mathbf{s}}] \\
G[l] = \textbf{push}\ n & \mathsf{wpi}(l) = \mathsf{wp}(l+1)[^{n :: \mathbf{s}}/_{\mathbf{s}}] \\
G[l] = \textbf{load}\ x & \mathsf{wpi}(l) = \mathsf{wp}(l+1)[^{x :: \mathbf{s}}/_{\mathbf{s}}] \\
G[l] = \textbf{store}\ x & \mathsf{wpi}(l) = \mathsf{wp}(l+1)[^{\mathbf{s}[0],\uparrow\mathbf{s}}/_{x,\mathbf{s}}] \\
G[l] = \textbf{aload}\ x & \mathsf{wpi}(l) = \mathsf{wp}(l+1)[^{a[\mathbf{s}[0]] :: \uparrow\mathbf{s}}/_{\mathbf{s}}] \\
G[l] = \textbf{astore}\ x & \mathsf{wpi}(l) = \mathsf{wp}(l+1)[^{[a:\mathbf{s}[0]\mapsto\uparrow\mathbf{s}[0]],\uparrow^2\mathbf{s}}/_{a,\mathbf{s}}] \\
G[l] = \textbf{jmp}\ l' & \mathsf{wpi}(l) = \mathsf{wp}(l+1) \\
G[l] = \textbf{cjmp} \bowtie l_t, l_f & \mathsf{wpi}(l) = (\mathbf{s}[0] \bowtie \uparrow os[0]) \Rightarrow \mathsf{wp}(l_t)) \wedge \\
 & \qquad (\neg(\mathbf{s}[0] \bowtie \uparrow os[0]) \Rightarrow \mathsf{wp}(l_f)) \\
G[l] = \textbf{invoke}\ f\ n & \mathsf{wpi}(l) = \mathsf{Pre}[^{\mathbf{s}[0],\uparrow\mathbf{s}[0],\ldots,\uparrow\mathbf{s}[0]}/_{\hat{x}_f}] \wedge \\
 & \qquad \forall_{\mathtt{res},V'} \mathsf{Post}[^{V',V}/_{V,V^\star}][^{x_f{}^\star}/_{x}] \Rightarrow \mathsf{wp}(l+1)[^{V'}/_{V}] \\
G[l] = \textbf{return}\ v & \mathsf{wpi}(l) = \mathsf{Post}_p[^{\mathbf{s}[0]}/_{\mathtt{res}}]
\end{array}
$$

$$
\mathsf{wp}(l) = \begin{cases} \mathsf{annot}(l) \text{ if } l \in \mathsf{dom}(\mathsf{annot}) \\ \mathsf{wpi}(l) \qquad\qquad \text{otherwise} \end{cases}
$$

Fig. 26. VCgen rules for Stack-based Code

Example 5. Figure 28 shows the result of compiling the RTL representation of Figure 23 into the stack-based representation.

Preservation of Proof Obligations. In this section we formalize the main result of this compilation step: assuming, for the result of the transformation, the same specification as the original code, verification conditions are preserved.

Then, if $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$ is the specification of an RTL procedure p, we define the specification for the compilation of the procedure p as $(\mathsf{Pre}, \mathsf{annot}, \mathsf{Post})$. Notice that it follows that intermediate invariants do not refer to stack expressions.

We first prove, in the following lemma, that predicate transformers are preserved by the compiler transformation defined in this section. More precisely, the computation of the wp function coincides at every program point up to the evaluation of stack expressions.

Lemma 13. *Let $\langle \mathcal{N}, \mathcal{E}, G \rangle$ be the graph of an* RTL *procedure p and seq the result of its compilation into the stack-based procedure $\bar{p}$. Then, assuming the same specification for p and $\bar{p}$, we have that for every program label l in $\mathcal{N}$, $\mathsf{wp}_p(l) = \mathsf{wp}_{\bar{p}}(l)$, and $\mathsf{wpi}_p(l) = \mathsf{wpi}_{\bar{p}}(l)$.*

$$
\begin{array}{l}
\mathcal{C}_l(\langle \mathcal{N}, \mathcal{E}, G\rangle, S) = (\ \text{if } l \in S \longrightarrow \mathbf{jmp}\ l \\
\quad \square\ \text{if } l \notin S \text{ and } G[\langle l, l_t\rangle] = v_1 \bowtie v_2? \longrightarrow \\
\qquad \text{let } (seq_t, S_t) = \mathcal{C}_{l_t}(\langle \mathcal{N}, \mathcal{E}, G\rangle, \{l\} \cup S) \\
\qquad \text{let } (seq_f, S_f) = \mathcal{C}_{l_f}(\langle \mathcal{N}, \mathcal{E}, G\rangle, S_t) \\
\qquad \text{in } (l : \mathbf{load}\ v_2 :: \mathbf{load}\ v_1 :: \mathbf{cjmp} \bowtie\ l_t, l_f :: seq_t :: seq_f, S_f) \\
\quad \square\ \text{if } l \notin S \text{ and } G[\langle l, l'\rangle] = v := v_1 \oplus v_2 \longrightarrow \\
\qquad \text{let } (seq, S') = \mathcal{C}_{l'}(\langle \mathcal{N}, \mathcal{E}, G\rangle, \{l\} \cup S) \\
\qquad \text{in } (l : \mathbf{load}\ v_2 :: \mathbf{load}\ v_1 :: \mathbf{prim} \oplus :: \mathbf{store}\ v :: seq, S') \\
\quad \square\ \text{if } l \notin S \text{ and } G[\langle l, l'\rangle] = v := a[v'] \longrightarrow \\
\qquad \text{let } (seq, S') = \mathcal{C}_{l'}(\langle \mathcal{N}, \mathcal{E}, G\rangle, \{l\} \cup S) \\
\qquad \text{in } (l : \mathbf{load}\ v' :: \mathbf{aload}\ a :: \mathbf{store}\ v :: seq, S') \\
\quad \square\ \text{if } l \notin S \text{ and } G[\langle l, l'\rangle] = a[v] := v' \longrightarrow \\
\qquad \text{let } (seq, S') = \mathcal{C}_{l'}(\langle \mathcal{N}, \mathcal{E}, G\rangle, \{l\} \cup S) \\
\qquad \text{in } (l : \mathbf{load}\ v' :: \mathbf{load}\ v :: \mathbf{astore}\ a :: seq, S') \\
\quad \square\ \text{if } l \notin S \text{ and } G[\langle l, l'\rangle] = \mathbf{invoke}\ f(\vec{x}) \longrightarrow \\
\qquad \text{let } (seq, S') = \mathcal{C}_{l'}(\langle \mathcal{N}, \mathcal{E}, G\rangle, \{l\} \cup S) \\
\qquad \text{in } (l : \mathbf{load}\ x_1 :: .. :: \mathbf{load}\ x_k :: \mathbf{invoke}\ f\ k :: seq, S') \\
\quad \square\ \text{if } l \notin S \text{ and } G[\langle l, l'\rangle] = \mathbf{return}\ v \longrightarrow \\
\qquad \text{let } (seq, S') = \mathcal{C}_{l'}(\langle \mathcal{N}, \mathcal{E}, G\rangle, \{l\} \cup S) \\
\qquad \text{in } (l : \mathbf{load}\ v :: \mathbf{return} :: seq, S') \\
\quad \square\ \text{if } l \notin S \text{ and } G[\langle l, l'\rangle] = \mathbf{nop} \longrightarrow \\
\qquad \text{let } (seq, S') = \mathcal{C}_{l'}(\langle \mathcal{N}, \mathcal{E}, G\rangle, \{l\} \cup S) \\
\qquad \text{in } (l : \mathbf{nop} :: seq, S') \\
\quad)
\end{array}
$$

Fig. 27. Compiler to Stack-based Code (Excerpt)

Proof. The proof proceeds by the induction principle associated to well-annotated procedures. The base case, e.g., $l \in \mathsf{dom}(\mathsf{annot})$*, is trivial since it follows by definition of* wp_p *and* $\mathsf{wp}_{\bar{p}}$*. Consider the case where* $G[\langle l, l'\rangle] \doteq x := y \oplus n$*. From the definition of the function* $\mathcal{C}$*, the instruction is compiled into a sequence of instructions* $l : \mathbf{load}\ y :: \mathbf{push}\ n :: \mathbf{prim} \oplus :: \mathbf{store}\ x$*. The computation of* $\mathsf{wp}_p(l)$ *returns* $\mathsf{wp}_p(l+1)[^{y \oplus n}/_x]$*, whereas* $\mathsf{wp}_{\bar{p}}(l)$ *is defined as*

$$\mathsf{wp}_{\bar{p}}(l+1)[^{x,\mathbf{s}}/_{\mathbf{s}[0],\uparrow\mathbf{s}}][^{\mathbf{s}}/_{\mathbf{s}[0]\oplus\mathbf{s}[1] :: \uparrow^2\mathbf{s}}][^{\mathbf{s}}/_{n :: \mathbf{s}}][^{\mathbf{s}}/_{y :: \mathbf{s}}]$$

Since by I.H. $\mathsf{wp}_p(l+1) = \mathsf{wp}_{\bar{p}}(l+1)$*, by reducing the stack expressions introduced by the substitutions in the latter formula, we prove the coincidence of both formulae.*

Hence, it follows from the definition of the set of proof obligations for RTL and stack-based code, and the fact that the compiler preserves the code structure, that proof obligations are syntactically preserved.

Lemma 14 (Preservation of Proof Obligations). *Let* p *be an* RTL *program and* $\bar{p}$ *the result of its compilation into the stack-based language. Assume* p *is*

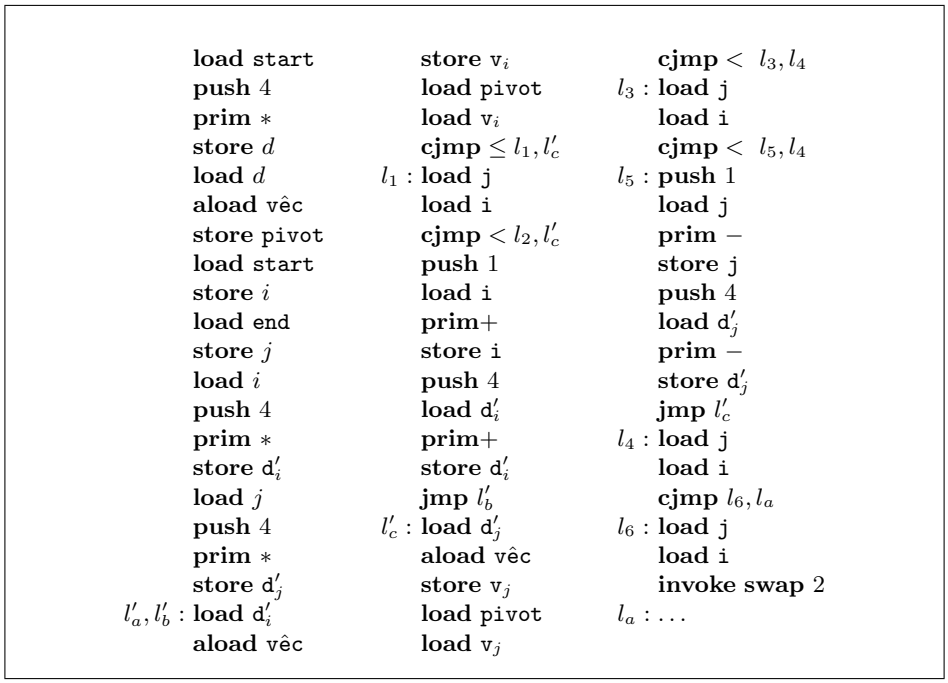


Fig. 28. Stack-based representation of the final code (Excerpt)

certified w.r.t. the specification (Pre, annot, Post). *Then,* $\bar{p}$ *is certified w.r.t. the specification* (Pre, annot, Post).

6 Conclusion

Certificate translation is a general method to transform certificates from source programs into certificates of compiled programs. In this tutorial, we have exemplified the underlying mechanisms of certificate translation on a running example. While representative, the example of the quicksort function fails to highlight some important aspects in certificate translation; these are briefly described in the next paragraph. For completeness, we conclude with a brief presentation of existing alternatives to certificate translation. Further discussion and pointers to the literature can be found in [3].

6.1 Other Topics in Certificate Translation

Important issues not covered by this tutorial include:

Certifying analyzers: optimizations that perform arithmetic reasoning require strengthening the loop invariants so that programs remain provable. These

strengthened invariants should assert the correctness of the results of the analysis, and should be proved automatically—and weaved together with the original proof of the program. This requires extending standard analyzers into certifying analyzers, that 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. The existence of certifying analyzers for transformations such as constant propagation or common subexpression elimination is shown in [2] in the context of a RTL language, and in a more general setting in [4].

Certificate translation in abstract interpretation: It is possible to take a more general approach to certificate translation by embedding the problem in the framework of abstract interpretation [8, 9]. One can then give sufficient conditions for transforming a certificate of a program G into a certificate of a program G', where G' is derived from G by a semantically justified program transformation, typically a program optimization. In [4], we provide substantial leverage w.r.t. [2], allowing to consider strongest post-condition calculi as well as weakest precondition calculi as done in this paper, and to some extent concurrent programs.

Hybrid certificates: in order to reduce the verification effort, verification environments increasingly rely on combining static analyses and verification condition generation. The verification condition generator exploits the information of the analysis in two useful ways: on the one hand, verification conditions that originate from spurious edges in the control-flow graph are discarded. This leads to fewer and smaller proof obligations. Furthermore, the verification condition generator adds the results of the analysis as additional assumptions to help prove the verification conditions. In [5], we initiate the study of certificate translation for hybrid verification, and we show preservation of proof obligations between hybrid verification frameworks for source code and a stack-based language similar to that of Section 4.

6.2 Alternatives to certificate translation

There are several mechanisms to certify a compiled code from a certificate of the source program:

Certifying compilers: They extend traditional compilers with a mechanism to automatically generate certificates for sufficiently simple safety properties, exploiting the information available about a program during its compilation. Certifying compilation [18] 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. The counterpart of this approach is that the properties under consideration are restricted to simple properties, namely typing predicates. An early example of certifying compiler is the Touchstone compiler [18], which was intended to explore the feasibility of PCC. This compiler generates, for programs written in a type-safe fragment of C, a formal proof for type-based safety and

memory safety of the resulting program in DEC Alpha assembly language. The Touchstone compiler automatically inserts the loops invariants in the resulting program and generates the correctness proofs.

Certified Compilers: The goal of certified compilers is to provide a formal guarantee of its correctness. It is a general result that proves that for every input program the results of the compilation have an equivalent semantics, for a particular definition of equivalence. A notable example of a certified compiler is provided by the CompCert [14] project. CompCert is a compiler, formalized in the Coq proof assistant, from a subset of C into PowerPC assembly code. A formal proof stating the equivalence between the source and the compiled code is formalized in the Coq proof assistant. There are two drawbacks to this approach, from the perspective of certificate translation. First, a formal definition of the compiler can be extremely large and, thus, the certificate of its correctness can be prohibitively expensive to check. Second, one must assume that the source code is available to the code user, in order to be inspected and compared with the compiled code. However, most commonly, one cannot expect code producers to release the corresponding source code.

Translation Validation: Translation validation [20, 17] is an alternative technique to formally verifying the correctness of compiler transformations. Instead of providing a full definition of the compiler and proving that it is correct in the sense that any compiled code is observably equivalent to the original one, it certifies correct every run of the compiler. That is, for every particular input program, and each transformation step, the infrastructure compares the semantics of the transformed code to the original semantics. For every transformation step, proof obligations, expressed in first-order logic, stating the semantics equivalence are fed into a prover to be discharged. The main advantage of this approach with respect to the previous one, is that the full definition of the compiler is not needed and, since proofs are specialized to a given particular program, certificates become significantly smaller. However, as with certified compilers, there is also the inconvenience of requiring the availability of the source program.

References

1. G. Barthe, P. Crégut, B. Grégoire, T. Jensen, and D. Pichardie. The MOBIUS proof carrying code infrastructure. In *Formal Methods for Components and Objects: 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, pages 1–24, Berlin, Heidelberg, 2008. Springer-Verlag.
2. G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science, pages 301–317. Springer-Verlag, 2006.
3. G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. *ACM Transactions on Programming Languages and Systems*, 2009. Extended version of [2].

4. G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In *European Symposium on Programming*, number 4960 in Lecture Notes in Computer Science, pages 368–382. Springer-Verlag, 2008.
5. G. Barthe, C. Kunz, D. Pichardie, and J. Samborski-Forlese. Preservation of proof obligations for hybrid verification methods. In *Software Engineering and Formal Methods*. IEEE Press, 2008.
6. L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In D. Mandrioli K. Araki, S. Gnesi, editor, *Formal Methods Europe*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.
7. D. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using esc/java2 and a report on a case study involving the use of esc/java2 to verify portions of an internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages*, pages 269–282, 1979.
10. Á. Darvas and P. Müller. Formal encoding of JML level 0 specifications in JIVE. Technical report, ETH Zurich, 2007. Annual Report of the Chair of Software Engineering.
11. R. W. Floyd. Assigning meanings to programs. *Proc. Symp. Appl. Math*, 19:19–31, 1967.
12. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
13. W. A. Howard. The Formulae-As-Types Notion Of Construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, Inc., New York, N.Y., 1980.
14. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett and S. L. Peyton Jones, editors, *Principles of Programming Languages*, pages 42–54. ACM Press, 2006.
15. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.
16. G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.
17. G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.
18. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Programming Languages Design and Implementation*, volume 33, pages 333–344, New York, NY, USA, 1998. ACM Press.
19. G.C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.
20. A. Pnueli, E. Singerman, and M. Siegel. Translation validation. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1998.