An Abstract Model of Certificate Translation

GILLES BARTHE, IMDEA Software Institute
CÉSAR KUNZ, IMDEA Software Institute and Universidad Politécnica de Madrid

13

A certificate is a mathematical object that can be used to establish that a piece of mobile code satisfies some security policy. In general, certificates cannot be generated automatically. There is thus an interest in developing methods to reuse certificates generated for source code to provide strong guarantees of the compiled code correctness. Certificate translation is a method to transform certificates of program correctness along semantically justified program transformations. These methods have been developed in previous work, but they were strongly dependent on particular programming and verification settings. This article provides a more general development in the setting of abstract interpretation, showing the scalability of certificate translation.

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

General Terms: Languages, Verification, Security

Additional Key Words and Phrases: Program verification, static analysis, program optimizations, proof-carrying code

ACM Reference Format:
Barthe, G. and Kunz, C. 2011. An abstract model of certificate translation. ACM Trans. Program. Lang. Syst. 33, 4, Article 13 (July 2011), 46 pages.
DOI = 10.1145/1985342.1985344 http://doi.acm.org/10.1145/1985342.1985344

1. INTRODUCTION

A certificate is a mathematical object that establishes the validity of a logical formula and that is self-contained and self-explanatory, and can be checked independently and automatically. Certificates arise naturally in many areas of mathematics, and in many different forms. In particular, certificates are common in the context of program verification, where they are used for automatic checking of program correctness; in this context, certificates provide evidence that a program meets its specification, where specifications may take the form of type annotations or assertions, and certificates may take the form of type derivations, derivations in Hoare-like logics, or proof terms for verification conditions. While certificate checking for program correctness is reasonably understood, certificate generation remains a challenging problem. Although it is possible to generate certificates automatically for specific properties

This work was partially funded by European Projects FP7-231620 HATS and FP7-256980 NESSoS, Spanish project TIN2009-14599 DESAFIOS 10, Madrid Regional project S2009TIC-1465 PROMETIDOS. C. Kunz is funded by a Juan de la Cierva Fellowship, MICINN, Spain.
Author's address: email: cesar.kunz@imdea.org.

DOI 10.1145/1985342.1985344 http://doi.acm.org/10.1145/1985342.1985344

that are enforceable by automated program analyses, and in particular type systems, certificate generation remains interactive in the general case. It is therefore of interest to develop methods that assist and simplify the construction of certificates.

The purpose of this article is to investigate methods for transforming certificates of program correctness. We focus on two instances of this general goal: certifying analysis and certificate translation. The aim of certifying analysis is to transform a proof of program correctness from one verification formalism to another; typically, a certifying analyzer will transform a proof using static analysis of a program into a proof of the same program based on deductive program verification. Certifying analyzers are proof-producing, in the sense that they return, in addition to their result, a formal proof of correctness that can be checked automatically. Certifying analyzers differ from certified analyzers [Cachera et al. 2004] in that the certificate establishes the validity of a single instance of the analysis, as certifying compilers [Necula and Lee 1998] do.

The aim of certificate translation is to transform a correctness proof for a source program into a correctness proof for a program built by transformations on the source program. In general, certificate translation considers settings where the source and transformed programs are verified using similar methods—for instance, abstract interpretation, type systems, or deductive program verification. Certificate translation differs from certifying compilation [Necula and Lee 1998] in that the latter generates a proof of correctness—rather than transforming an existing one—and from certified compilation [Leroy 2006], in that it proves the preservation of a particular property for a particular program—rather than establishing a general semantic equivalence.

The primary application of certifying analysis and of certificate translation is to provide automated means to construct certificates of executable code from interactive verification of source programs. Typically, the source program will be written in a high-level language and the transformed program will be obtained by optimizing compilation. Certificate translation allows for extracting guarantees on executable code from the results of source code verification, and to extend the scope of Proof Carrying Code [Necula and Lee 1996; Necula 1998] to a wide range of program properties. More specifically, certificate translation allows code producers to verify their source code using program verification environments, and yet to extract certificates of the executable code that can then be sent to the code consumers.

Certifying analysis and certificate translation are tightly connected, since the transformation from the source to the target program is usually justified through a program analysis, whose automatic certification is required to build a certificate for the target program. Section 2 illustrates this issue.

Previous works on certificate translation and other methods have been developed for specific compilation infrastructures and verification settings; see Section 7 for an overview. However, a limitation of earlier work is the lack of a unifying framework in which to formulate and analyze the basic concepts underlying certificate translation. The lack of a framework makes it difficult to assess the generality and scalability of certificate translation, and even to find the commonalities between the different results in the literature. The present article overcomes this limitation: we capture the essence of certificate translation in the setting of abstract interpretation [Cousot and Cousot 1977, 1979], a general framework for analyzing programs, and to reason about program analyses at large. Abstract interpretation is a natural framework to model certificate translation, since it provides a common model for two of its key components, namely: the verification environment in which the original certificate c is produced and the static analysis that justifies the program optimization. Furthermore, abstract interpretation provides a natural setting to analyze the relation between different verification methods.

Adopting the framework of abstract interpretation for studying certificate translation provides significant leverage on earlier work. First, it allows us to be generic, and to abstract away from the specifics of programming languages, program transformations, and verification methods. Concretely, we are able to formulate in an abstract setting a set of constraints on the verification methods and program transformations that guarantee the existence of certificate translators. The generality of the approach is demonstrated through examples, and specifically through a principled rediscovery of the results of Barthe et al. [2009]—to the exception of dead code, which in the present article is handled differently from Barthe et al. [2009]. Second, the framework allows us to derive new certificate translation results, by casting the verification frameworks considered as abstract interpretations, and by showing that the optimizations considered can be built by instantiating and combining our general transformations. For example, Kunz [2010] derives from our results proofs of correctness of certificate translation for a concurrent imperative language. In a similar way, the framework allows through mild modifications leveraging results to more general settings. For example, Section 5.4 extends the results of certificate translation to program analyses whose results can be expressed as 2-properties, that is, properties about two runs of programs. Third, the framework allows us to address issues that do not immediately relate to certificate translation nor certifying analysis, including the possibility of developing hybrid certificates that can be checked by the collaboration of two verification formalisms, typically static analysis and deductive verification.

This article is an extended version of the conference article that formalizes certificate translation in the setting of abstract interpretation [Barthe and Kunz 2008]. The main contributions are as follows:

— The introduction of *certified solutions*, extending the notion of solution with certificates (Section 3). Informally, every abstract semantics yields for each program a system of dataflow equations, expressed as a set of constraints. A solution for a program is a labeling of its program points with abstract properties that satisfy the set of constraints induced by the abstract semantics. By extension, a certified solution consists of a solution S and a certificate that S satisfies the dataflow constraints;
— Sufficient conditions for the existence of *certifying analyzers* (in Section 4). Certifying analyzers extend program analyzers by providing certificates of correctness for the results they generate, and are used as inputs by certificate translation algorithms. We consider both cases where the verification settings are based on symbolic evaluation and verification condition generation;
— Sufficient conditions for the existence of *certificate translation* algorithms that map certificates of source programs to certificates of transformed programs (in Sections 5 and 5.4). We consider three transformations, code duplication, local code transformation, and code coalescing, which can be combined to model a large set of program optimizations. For each transformation, we provide a set of proof obligations from which one can build certificate translation algorithms: that is, we describe algorithms that take as input the certificates for the proof obligations of the original program and return certificates for the transformed program;
— *Instantiations* of the certifying analyzers and certificate translation algorithms. For conciseness, we consider a simple sequential imperative language and standard optimizations: loop unrolling, common subexpression elimination, induction variable strength reduction, and dead variable elimination. This instantiation allow us to clarify our contributions and to recover, up to minor differences, the results of Barthe et al. [2009];
— A formal definition of *hybrid verification* (in Section 6). The interaction between two verifiers helps reducing the contract specification effort, as well as reducing the

l_1 : $w := 1$
l_2 : **while** $(y \neq 1)$ **do**
l_3 : **if** $(y \bmod 2 = 1)$ **then**
l_4 : $w := w * x$
fi
l_5 : $x := x * x$
$y := y/2$
done
l_6 : $x := w * x$

Fig. 1. Fast exponentiation algorithm.

complexity of the verification conditions. We use our abstract framework to model the collaboration between two verifiers. Based on this formalization, we provide a set of sufficient conditions to transfer a hybrid verification result into a nonhybrid certified solution.

2. MOTIVATING EXAMPLE

Certificate translation is best illustrated through an example; although the notions used throughout the example are only defined in subsequent sections, the example is sufficiently simple to convey an intuition about the issues with certificate translation, and its relationship with certifying analysis.

Consider the fast exponentiation algorithm fexp in Figure 1; in the figure $./2$ denotes integer division. The algorithm takes as input two integers x and y, computes x^y, and stores the final result in x. Using X and Y to denote the initial values for x and y respectively, one can specify the behavior of fexp with pre- and postconditions:

$$\{x = X \wedge y = Y\}\mathsf{fexp}\{x = X^Y\}.$$

Here the precondition captures the usage of X and Y as initial values of x and y, whereas the postcondition captures the correctness of the algorithm.

Using a verification condition generator, one can certify the correctness of the program with respect to its specification in two steps. First, one must provide a loop invariant; in this case, we define the loop invariant as $w * x^y = X^Y$. Then, one must provide certificates for the proof obligations; more concretely, one must provide a triple $(c_{pre}, c_{inv}, c_{post})$ where:

(1) c_{pre} is a certificate of the formula

$$(x = X \wedge y = Y) \Longrightarrow 1 * x^y = X^Y$$

stating that the precondition entails the invariant;

(2) c_{inv} is a certificate of the formula

$$(w * x^y = X^Y \wedge y \neq 1) \Longrightarrow$$
$$((y \bmod 2 = 1 \Longrightarrow w * x * (x^2)^{\frac{y}{2}} = X^Y) \wedge (y \bmod 2 \neq 1 \Longrightarrow w * (x^2)^{\frac{y}{2}} = X^Y))$$

stating that the invariant is preserved by the body of the loop, assuming the guard of the loop holds;

(3) c_{post} is a certificate of the formula

$$(w * x^y = X^Y \wedge y = 1) \Longrightarrow w * x = X^Y$$

stating that the invariant, when strengthened with the negation of the loop guard, entails the postcondition.

$$
\begin{array}{ll}
l_1: & w := 1 \\
l_2: & \textbf{while } (y \neq 1) \textbf{ do} \\
l_5: & \quad x := x * x \\
 & \quad y := y/2 \\
 & \textbf{done} \\
l_6: & x := w * x
\end{array}
$$

Fig. 2. Specialized exponentiation algorithm.

The certificates c_{pre} and c_{post} are built by applying substitutivity of equality and elementary reasoning, whereas the certificate for c_{inv} is built using basic properties of exponentiation and certificates for the two lemmas:

$$
\begin{aligned}
y \bmod 2 = 1 &\implies y = 2 * y/2 + 1 \\
y \bmod 2 \neq 1 &\implies y = 2 * y/2.
\end{aligned}
$$

Now suppose that $Y = 2^k$ for some k; then the algorithm fexp can be optimized into the algorithm pexp shown in Figure 2. Indeed, the assumption on Y entails that $y \bmod 2 \neq 1$ holds for each loop iteration, and therefore the **if** statement can be removed, as its test always fails.

The objective of certificate translation is to build from c a certificate c' of the correctness of pexp its specification, that is, of

$$\{x = X \wedge y = Y \wedge \exists k.\, Y = 2^k.\}.\mathsf{pexp}\{x = X^Y\}.$$

As for the fast exponentiation algorithm, one must provide a loop invariant and a triple $(c_{pre}, c_{inv}, c_{post})$ of certificates for verification conditions. If we choose the invariant used for fexp, then the second proof obligation becomes:

$$(w * x^y = X^Y \wedge y \neq 1) \implies w * (x^2)^{y/2} = X^Y,$$

which cannot be proved without knowing that $y \bmod 2 \neq 1$. Now, consider the strengthened invariant

$$w * x^y = X^Y \wedge \exists k'.\, y = 2^{k'}.$$

Using the strengthened invariant, the program pexp can be certified by providing a triple $(\bar{c}_{pre}, \bar{c}_{inv}, \bar{c}_{post})$ where:

(1) $\bar{c}_{pre}$ is a certificate of the formula

$$(x = X \wedge y = Y \wedge \exists k.\, Y = 2^k) \implies 1 * x^y = X^Y \wedge \exists k'.\, y = 2^{k'}$$

stating that the precondition entails the invariant;

(2) $\bar{c}_{inv}$ is a certificate of the formula

$$(w * x^y = X^Y \wedge \exists k'.\, y = 2^{k'} \wedge y \neq 1) \implies (w * (x^2)^{y/2} = X^Y \wedge \exists k'.\, y/2 = 2^{k'})$$

stating that the invariant is preserved by the body of the loop, assuming the guard of the loop holds;

(3) $\bar{c}_{post}$ is a certificate of the formula

$$(w * x^y = X^Y \wedge \exists k'.\, y = 2^{k'} \wedge y = 1) \implies w * x = X^Y,$$

stating that the invariant, when strengthened with the negation of the loop guard, entails the postcondition.

The main contribution of this paper is to provide a principled approach for building the certificates $\bar{c}_{pre}$, $\bar{c}_{inv}$ and $\bar{c}_{post}$ from c_{pre}, c_{inv} and c_{post}, respectively. It is direct to

build the certificates $\bar{c}_{pre}$ and $\bar{c}_{post}$; on the contrary, building the certificate $\bar{c}_{inv}$ is not immediate. According to our approach, the certificate $\bar{c}_{inv}$ is built from the certificate c_{inv}, and from an automatically generated certificate c_a. Informally, c_{inv} is used to establish the first conjunct of the conclusion from the first conjunct of the hypothesis, whereas c_a establishes that $\exists k'.\ y = 2^{k'}$ is an invariant for the loop body, that is, it is a certificate of the formula

$$(\exists k'.\ y = 2^{k'} \wedge y \neq 1) \Longrightarrow \exists k'.\ y/2 = 2^{k'}.$$

While the certificate c_{inv} is retrieved from the source program, the certificate c_a is built by a certifying analysis that computes, for each program point, whether a variable holds an arbitrary value, an even value, an odd value, or a power of 2. In its noncertifying form, the analysis returns for each program point an annotation stating that y is a power of two. Its certifying counterpart maps every annotation to a logical formula, and produces a certificate for the annotated program: in this case, the annotation "is a power of two" is translated to the formula $\exists k'.\ y = 2^{k'}$ and the certificate proves the annotated program.

3. PROGRAM, SEMANTICS, ABSTRACT INTERPRETATION

The purpose of this section is to introduce the framework in which we study the existence of certificate translators. Our framework is heavily inspired from abstract interpretation, and only the notion of certified solution (see Section 3.4) appears to be novel.

The section is organized as follows: first, we formalize programs as flow graphs in Section 3.1. We then define abstract semantics of programs using transfer functions over abstract states in Section 3.2; in Section 3.3 we define solutions of the data flow constraints induced by the transfer functions. Solutions are the central notion of this paper, and unify the main components of certificate translation: the static analysis that justifies program transformations and the verification infrastructure used to prove program correctness. Next, in Section 3.4 we introduce certified solutions, which extend solutions with certificates of the data flow constraints. Certified solutions are used in lightweight bytecode verification and in static analyses for which fixpoint checking might be too costly. In Section 3.5, we define partial labelings, which instead of providing a solution, provides sufficient information for the solution to be efficiently computed. Partial labelings arise in verification infrastructures based on verification conditions, and in lightweight bytecode verification. For completeness, we also review in Section 3.6 the definition of concrete semantics and the definition of soundness of an abstract semantics with respect to concrete semantics.

3.1 Programs

We view programs as flow graphs whose edges are decorated with statements.

Definition 3.1 *Programs*. A program consists of an annotated directed graph $\langle \mathcal{N}, \mathcal{E}, G \rangle$, where $\mathcal{N}$ is a finite set of nodes, with a distinguished initial node l_{init}, and $\mathcal{E} : \mathcal{N} \times \mathcal{N}$ is a relation on nodes that describes the execution flow, and $G : \mathcal{E} \to \mathsf{Stmt}$ is a function that assigns a statement to each pair of nodes in $\mathcal{E}$. We let $\mathcal{O} \subseteq \mathcal{N}$ denote the set of nodes without successors, that is, $l \in \mathcal{O}$ iff for all $l' \in \mathcal{N}$, $(l, l') \notin \mathcal{E}$.

The results of the paper are not tied to any programming language. However, it is convenient for illustrative purposes to consider the minimalistic statement language $\mu\mathsf{Stmt}$ of Figure 3. In this setting, each edge is labeled with a sequential composition of skip statements, assertions of the form $\mathsf{assert}\, b$, where b is a boolean expression, and

$$
\begin{array}{rl}
\textbf{integer expressions} & e ::= n \mid x \mid -e \mid e + e \mid e * e \mid e/e \mid e \bmod e \\
\textbf{boolean expressions} & b ::= \mathsf{true} \mid \mathsf{false} \mid e \leq e \mid e = e \mid e \neq e \mid \neg b \mid b \wedge b \\
\textbf{statements} & s ::= \mathsf{skip} \mid s; s \mid x := e \mid \mathsf{assert}\, b
\end{array}
$$

Fig. 3. Program statements.

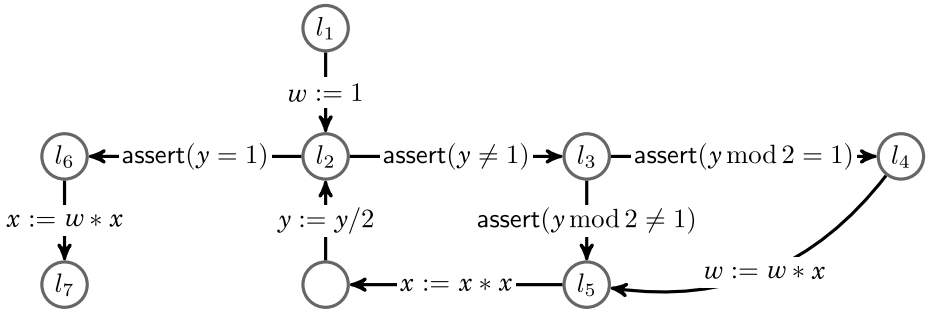

Fig. 4. Program representation.

assignments of the form $x := e$, where x is a scalar variable drawn from a set of variables Var, and e is an expression. This set of statements is sufficiently expressive to compile programs written in the simple imperative language used for the fast exponentiation algorithm of Section 2. In particular, fexp is represented by the program given in Figure 4. In the figure, the node l_2 represents the header of the loop, and the edges (l_2, l_3) and (l_2, l_6) represent the executions that enter or exit the loop body, respectively. The node l_3 represents the execution point immediately before the evaluation of the condition $y' \bmod 2 = 1$. The fact that the execution may follow two different branches, that is, towards nodes l_4 or l_5, is modeled by the edges (l_3, l_4) and (l_3, l_5), representing the cases in which the condition $y' \bmod 2 = 1$ is satisfied or not, respectively.

3.2 Abstract Semantics

Abstract semantics assign to each program point a property that describes the set of states that may reach that program point. The abstract properties form an abstract domain A with a rich algebraic structure.

Definition 3.2 *Abstract domain.* A tuple $\mathbf{A} = \langle A, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ is an abstract domain if:

— A is a set whose elements are called abstract properties;
— $\sqsubseteq$ is a pre-order, i.e. a reflexive and transitive relation;
— $\sqcup$ is a join operator, satisfying for all $a, b, c \in A$ the conditions $a \sqsubseteq a \sqcup b$, and $b \sqsubseteq a \sqcup b$, and

$$(a \sqsubseteq c \wedge b \sqsubseteq c) \Longrightarrow a \sqcup b \sqsubseteq c;$$

— $\sqcap$ is a meet operator, satisfying for all $a, b, c \in A$ the conditions $a \sqcap b \sqsubseteq a$, and $a \sqcap b \sqsubseteq b$, and

$$(c \sqsubseteq a \wedge c \sqsubseteq b) \Longrightarrow c \sqsubseteq a \sqcap b;$$

— $\top$ is the greatest element in A, i.e. for all $a \in A$, $a \sqsubseteq \top$;
— $\bot$ is the least element in A, i.e. for all $a \in A$, $\bot \sqsubseteq a$.

It is sufficient, for a particular style of abstract interpretation, to consider meet or join semi-lattices, depending on the flow f of the interpretation. However, we find it more

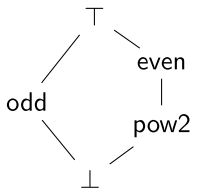

Fig. 5. Abstract domain of parity with powers of two.

convenient to require our domains to be lattices, since we may deal simultaneously with analyses defined by forward and backward transfer functions.

Note that it is customary for abstract interpretation to require that the underlying order of an abstract domain satisfies anti-symmetry, and thus is a pre-order. However, it is not essential to require anti-symmetry to model approximation; see, for instance, Section 5 of Cousot and Cousot [1992]. An informal argument is that the theory of partial orders can be viewed as an extension of the theory of preorders with a new relation, that is, equality, and a new axiom for anti-symmetry,that is,

$$\forall a, b\,.(a \sqsubseteq b \wedge b \sqsubseteq a) \Leftrightarrow a = b\,.$$

Hence every statement about partial orders can be reformulated as a statement on preorders by replacing equality by its definition. Moreover, the natural domain for the verification infrastructure is the domain of propositions, where the order $\sqsubseteq$ is logical implication, which is a preorder rather than a partial order—more concretely, viewing logically equivalent propositions as equal would later imply that logically equivalent formula have the same sets of certificates, which is not desirable.

Turning back to our motivating example, we observe that it implicitly involves two abstract domains: the domain of propositions and an abstract domain that extends parity with powers of 2. The lattice of propositions has as its set of elements the set of logical formulas, ordered by logical implication; then, $\sqcap$ and $\sqcup$ are defined as the logical conjunction and logical disjunction, respectively, and finally $\bot$ = false and $\top$ = true. The lattice for parity has elements even, odd, pow2, as well as elements $\top$ and $\bot$. Figure 5 provides a graphical view of the lattice.

We now turn to the definition of abstract semantics.

Definition 3.3 *Abstract semantics*. An abstract semantics over an abstract domain **A** is given by a pair $I = \langle [\![.]\!]_A, f\rangle$, where

— for every statement s, $[\![s]\!]_A : A \to A$ is a monotone function. In the sequel, we often omit the subscript A;
— f is the flow of the interpretation, either forwards (fwd) or backwards (bwd).

Typical examples of abstract semantics are the weakest precondition and strongest postcondition calculus. The definition of wp and sp for μStmt is standard:

$$\begin{array}{llll}
\mathsf{wp}(x := e)\ \phi & \doteq \phi[^{e}/_{x}] & \mathsf{sp}(x := e)\ \phi & \doteq \exists x'.\ (\phi[^{x'}/_{x}] \wedge x = e[^{x'}/_{x}]) \\
\mathsf{wp}(\mathsf{assert}\ b)\ \phi & \doteq b \Rightarrow \phi & \mathsf{sp}(\mathsf{assert}\ b)\ \phi & \doteq b \wedge \phi \\
\mathsf{wp}(\mathsf{skip})\ \phi & \doteq \phi & \mathsf{sp}(\mathsf{skip})\ \phi & \doteq \phi \\
\mathsf{wp}(s_1; s_2)\ \phi & \doteq \mathsf{wp}(s_1)\ (\mathsf{wp}(s_2)\ \phi) & \mathsf{sp}(s_1; s_2)\ \phi & \doteq \mathsf{sp}(s_2)\ (\mathsf{sp}(s_1)\ \phi),
\end{array}$$

where $\phi[^{e}/_{x}]$ stands for the substitution of the expression e for x in ϕ.

Other typical examples of (forward) abstract semantics are value analyses, which propagate knowledge about the value of a variable throughout the program points.

$+, *$	$\top$	odd	even	pow2
$\top$	$\top, \top$	$\top, \top$	$\top, \top$	$\top, \top$
odd	$\top, \top$	even, odd	odd, even	odd, even
even	$\top, \top$	odd, even	even, even	even, even
pow2	$\top, \top$	odd, even	even, even	even, pow2

	$./2$
$\top$	$\top$
odd	$\top$
even	$\top$
pow2	pow2

Fig. 6. Abstract operators over extended parity domain.

Suppose that $\mathbf{A}$ is a lattice of abstract values; one can then define the lattice of abstract environments as follows:

— its underlying set is $(\mathsf{Var} \to A \setminus \{\bot_A\}) \cup \{\bot\}$, and its pre-order is defined by the clauses $f \leq g$ iff $f = \bot$ or $f\ x \sqsubseteq g\ x$ for every $x \in \mathsf{Var}$;
— the least element is $\bot$ and the greatest element is $\lambda x.\top$;
— the join operator is defined by the clauses $f \sqcup g = g$ if $f = \bot$, $f \sqcup g = f$ if $g = \bot$, and as $\lambda x.\ (f\ x) \sqcup (g\ x)$ otherwise;
— the meet operator is defined by the clauses $f \sqcap g = \bot$ if $f = \bot$ or $g = \bot$, and as $\lambda x.\ (f\ x) \sqcap (g\ x)$ otherwise.

Then, given an interpretation function for expressions, that is, given for every expression e a function

$$[\![e]\!] : ((\mathsf{Var} \to A \setminus \{\bot_A\}) \cup \{\bot\}) \to A$$

one can define the abstract semantics of a statement as follows. First, the abstract semantics of $x := e$ is defined by the clauses $[\![x := e]\!]\ \rho = \bot$ if $\rho = \bot$ and otherwise

$$([\![x := e]\!]\ \rho)\ y = \begin{cases} \rho\ y & \text{if } x \neq y \\ [\![e]\!]\ \rho & \text{if } x = y. \end{cases}$$

The abstract semantics of $\mathsf{assert}\ b$ and skip is the identity. The abstract semantics of a sequence is the composition of the abstract semantics of its components, that is,

$$[\![s_1; s_2]\!]\ \rho = [\![s_2]\!]([\![s_1]\!]\ \rho).$$

The abstract interpretation of an expression over an abstract environment of type $\mathsf{Var} \to A \setminus \{\bot_A\}$ is defined by the abstract arithmetic operators shown in Figure 6. In Section 4 we provide another example of value analysis.

3.3 Dataflow Equations and Solutions

The abstract semantics defines for each program a set of data flow equations. A common means to verify program properties is to consider solutions of these systems of equations.

Definition 3.4 *Solution*. Let $\mathbf{A}$ be an abstract domain and let $I = \langle [\![.]\!], f \rangle$ be an abstract semantics over $\mathbf{A}$.

— A labeling for a program $\langle \mathcal{N}, \mathcal{E}, G \rangle$ is a mapping $S : \mathcal{N} \to A$.
— A labeling $S : \mathcal{N} \to A$ for a program $\langle \mathcal{N}, \mathcal{E}, G \rangle$ is a solution if the following holds:
  — $f = \mathsf{bwd}$ and for every l in $\mathcal{N}$, $S(l) \sqsubseteq \bigsqcap_{(l,l') \in \mathcal{E}} [\![G(l, l')]\!](S(l'))$; or
  — $f = \mathsf{fwd}$ and for every node l' in $\mathcal{N}$, $\bigsqcup_{(l,l') \in \mathcal{E}} [\![G(l, l')]\!](S(l)) \sqsubseteq S(l')$.

Notice that, since $\mathcal{E}$ is a finite relation, the expressions $\bigsqcap_{e \in \mathcal{E}}$ and $\bigsqcup_{e \in \mathcal{E}}$ represent a finite number of applications of the operations $\sqcap$ and $\sqcup$, respectively.

$$
\begin{array}{ll}
\mathsf{axiom} & : \mathcal{C}(a \sqsubseteq a) \\
\mathsf{weak}_{\sqcap} & : \mathcal{C}(a \sqsubseteq b) \to \mathcal{C}(a \sqcap c \sqsubseteq b) \\
\mathsf{weak}_{\sqcup} & : \mathcal{C}(a \sqsubseteq b) \to \mathcal{C}(a \sqsubseteq b \sqcup c) \\
\mathsf{elim}_{\sqcap} & : \mathcal{C}(c \sqcap a \sqsubseteq b) \to \mathcal{C}(c \sqsubseteq a) \to \mathcal{C}(c \sqsubseteq b) \\
\mathsf{intro}_{\sqcup} & : \mathcal{C}(a \sqsubseteq c) \to \mathcal{C}(b \sqsubseteq c) \to \mathcal{C}(a \sqcup b \sqsubseteq c) \\
\mathsf{intro}_{\sqcap} & : \mathcal{C}(a \sqsubseteq b) \to \mathcal{C}(a \sqsubseteq c) \to \mathcal{C}(a \sqsubseteq b \sqcap c)
\end{array}
$$

Fig. 7. Proof algebra.

Example 3.5. Consider the program shown in Figure 4 and the value analysis defined in the previous section. Let $\rho : \mathsf{Var} \to A \setminus \{\bot_A\}$ be such that $\rho y = \mathsf{pow2}$ and $\rho v = \top$ for any other variable v. Let S be labeling such that $S(l) = \rho$ for every $l \in \mathcal{N}$. One can see that S is a solution, as it satisfies the constraints given in Definition 3.4. For instance, in order to show that $\rho y = \mathsf{pow2}$ is preserved by the assignment $y := y/2$, one must check that $[\![y := y/2]\!]\rho \sqsubseteq \rho$, which holds from the definition of the abstract operator $./2$ shown in Figure 6.

Although we do not need it for this paper, we point out that it is possible to characterize solutions as (pre- or post-) fixpoints over the functional lattice $\mathcal{N} \to A$ of the monotonic operator that associates to every labeling $S \in \mathcal{N} \to A$ the labeling S' such that for every $l \in \mathcal{N}$, $S'(l)$ is defined as

$$\bigsqcup_{(l',l)\in\mathcal{E}} [\![G(l',l)]\!](S(l'))$$

in the case when $f = \mathsf{fwd}$—and dually in the case $f = \mathsf{bwd}$. See, for instance, Hankin et al. [2005] for more details on dataflow analyses.

3.4 Certified Solutions

A certified solution is a labeling with a certificate proving that it satisfies the data flow constraints imposed by the analysis. In order to capture the notion of certified solution at an appropriate level of abstraction, we do not commit to a particular representation of certificates; instead, we define an abstract notion of proof algebra.

Definition 3.6 *Proof algebra*. A proof algebra for an abstract domain $\mathbf{A}$ is a function $\mathcal{C}$ that assigns to every $a, a' \in A$ a set of certificates $\mathcal{C}(a \sqsubseteq a')$ such that:

— $\mathcal{C}$ is closed under the operations of Figure 7, where $a, b, c \in A$;
— $\mathcal{C}$ is sound, that is, for every $a, a' \in A$, if $a \not\sqsubseteq a'$, then $\mathcal{C}(a \sqsubseteq a') = \emptyset$.

In the sequel, we write $c : a \sqsubseteq a'$ or $c : a' \sqsupseteq a$ instead of $c \in \mathcal{C}(a \sqsubseteq a')$. Moreover, we sometimes use the function

$$\mathsf{trans} : \mathcal{C}(a \sqsubseteq b) \to \mathcal{C}(b \sqsubseteq c) \to \mathcal{C}(a \sqsubseteq c)$$

defined by composition of the algebra operations $\mathsf{weak}_{\sqcap}$ and $\mathsf{elim}_{\sqcap}$.

The particular form of certificates is irrelevant for this article. In the context of the lattice of logical assertions, it may be helpful for the reader to think about certificates in the perspective of the Curry-Howard isomorphism [Sørensen and Urzyczyn 2006]. Under this interpretation, propositions are types: conjunctions are interpreted as products, disjunctions are interpreted as disjoint sums, implications are interpreted as function spaces, etc. Then $\mathcal{C}$ is given by the typing judgment of a typed λ-calculus. For example, the operator $\mathsf{intro}_{\sqcap}$ is the term $\lambda f.\ \lambda g.\ \lambda a.\ \langle fa, ga \rangle$ of type $(a \Rightarrow b) \Rightarrow (a \Rightarrow c) \Rightarrow (a \Rightarrow (b \wedge c))$. Likewise one can provide a type-theoretical interpretation to the functions of Figure 7.

Definition 3.7 *Certified solution*. Let $\mathbf{A}$ be an abstract domain and let $\mathcal{C}$ be a proof algebra for $\mathbf{A}$. Let $I = \langle [\![.]\!], f\rangle$ be an abstract semantics over $\mathbf{A}$. A certified solution for a program $\langle \mathcal{N}, \mathcal{E}, G\rangle$ consists of a labeling $S : \mathcal{N} \to A$, and a family $c = \{c_l\}_{l\in\mathcal{N}}$ of certificates such that for every $l \in \mathcal{N}$,

— if $f = \mathsf{bwd}$ then $c_l : S(l) \sqsubseteq \bigsqcap_{(l,l')\in\mathcal{E}} [\![G(l,l')]\!](S(l'))$;
— if $f = \mathsf{fwd}$ then $c_l : \bigsqcup_{(l',l)\in\mathcal{E}} [\![G(l',l)]\!](S(l')) \sqsubseteq S(l)$.

It follows from the soundness of the proof algebra that every certified solution is a solution. Conversely, one can view every solution as a certified solution, by considering proof-irrelevant proof algebras, that is, in which for every $a, a' \in A$ the set of certificates $\mathcal{C}(a \sqsubseteq a')$ is either empty, or is equal to the singleton set $\{\bullet\}$, where $\bullet$ is a distinguished element.

3.5 Partial Labelings

Many techniques, including lightweight bytecode verification [Rose 2003] and abstraction carrying code [Albert et al. 2005], do not bundle code with a full (certified) solution, but with a partial labeling (and some certificates) from which a full (certified) solution can be reconstructed. The purpose of this section is to show the construction of a (certified) solution from a (certified) partial labeling. We start by defining partial labelings.

Definition 3.8 *Partial labeling*. A partial labeling for a program $\langle \mathcal{N}, \mathcal{E}, G\rangle$ is a partial function $S : \mathcal{N} \rightharpoonup A$ such that entry and output nodes are annotated, that is, $\mathcal{O} \cup \{l_{\mathsf{init}}\} \subseteq \mathsf{dom}(S)$, and such that the program is sufficiently annotated, that is, the graph $\langle \mathcal{N}_0, \mathcal{E} \cap \mathcal{N}_0 \times \mathcal{N}_0\rangle$ where $\mathcal{N}_0 = \mathcal{N} \setminus \mathsf{dom}(S)$ is acyclic.

The first condition on partial labelings ensures that the program pre- and postcondition are specified—by its annotations on entry and output nodes—whereas the second condition ensures that we dispose of sufficient annotations on loops to reconstruct a total labeling from the partial one. Note that a labeling can be seen as a partial labeling S such that $\mathsf{dom}(S) = \mathcal{N}$. Moreover, note that given a partial labeling S for a program $\langle \mathcal{N}, \mathcal{E}, G\rangle$, one can define for every node its weight as the length of the shortest path to an annotated node, i.e. a node that belongs to the domain of S. One can then reason by induction on the weight of nodes.

We now show how to build a labeling from a partial one.

Definition 3.9 *Annotation propagation, verification condition*. Let annot be a partial labeling for a program $P = \langle \mathcal{N}, \mathcal{E}, G\rangle$. The total labeling $\overline{\mathsf{annot}}$ for the program P is defined by the clause:

$$— \text{ if } f = \mathsf{bwd},\ \overline{\mathsf{annot}}(l) = \begin{cases} \mathsf{annot}(l) & \text{if } l \in \mathsf{dom}(\mathsf{annot}) \\ \bigsqcap_{(l,l')\in\mathcal{E}} [\![G(l,l')]\!](\overline{\mathsf{annot}}(l')) & \text{otherwise} \end{cases}$$

$$— \text{ if } f = \mathsf{fwd},\ \overline{\mathsf{annot}}(l) = \begin{cases} \mathsf{annot}(l) & \text{if } l \in \mathsf{dom}(\mathsf{annot}) \\ \bigsqcup_{(l',l)\in\mathcal{E}} [\![G(l',l)]\!](\overline{\mathsf{annot}}(l')) & \text{otherwise.} \end{cases}$$

For every $l \in \mathsf{dom}(\mathsf{annot})$, the verification condition $\mathsf{vc}(l)$ is defined by the clause

— $\mathsf{vc}(l) := \mathsf{annot}(l) \sqsubseteq \bigsqcap_{(l,l')\in\mathcal{E}} [\![G(l,l')]\!](\overline{\mathsf{annot}}(l'))$ if $f = \mathsf{bwd}$;
— $\mathsf{vc}(l) := \bigsqcup_{(l',l)\in\mathcal{E}} [\![G(l',l)]\!](\overline{\mathsf{annot}}(l')) \sqsubseteq \mathsf{annot}(l)$ if $f = \mathsf{fwd}$.

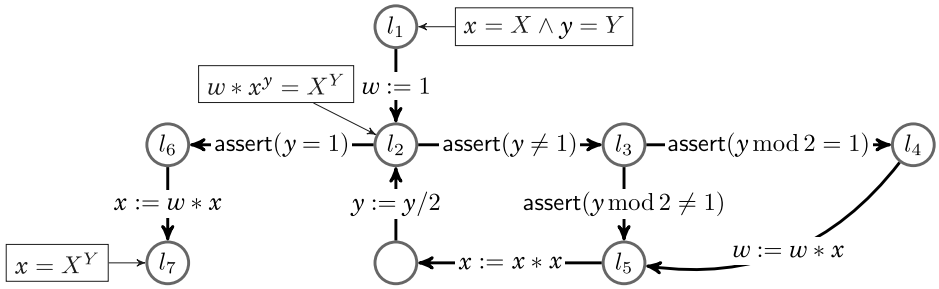

Fig. 8. Annotated program.

Example 3.10. A weakest precondition calculus is a typical example of verification mechanisms that operate on partially labeled programs. Consider the program in Figure 8. One can easily check that it is sufficient to annotate the set of labels $\{l_1, l_2, l_7\}$: a precondition, a postcondition, and a loop invariant, as shown in the figure. Indeed, since the subgraph $\mathcal{N} \setminus \{l_1, l_2, l_7\}$ is acyclic, a total labeling can be reconstructed from a specification with domain $\{l_1, l_2, l_7\}$. The weakest precondition fully annotates the program, and extracts proof obligations as described above.

In the rest of this section, we show that it is sufficient to provide one certificate for each $l \in \{l_1, l_2, l_7\}$. In this particular example, this corresponds to the certificates c_i, c_l, and c_f introduced in Section 2.

The following lemma shows that, given a partial labeling annot, one can build certificates of the verification conditions for $\overline{\mathsf{annot}}$ from certificates of the verification conditions for annot.

LEMMA 3.11. *Let* annot *be a partial labeling for a program* $\langle \mathcal{N}, \mathcal{E}, G \rangle$ *and assume we have a certificate* c_l : vc(l) *for every* $l \in$ dom(annot)*. Then there exists* c' *such that* $\langle \overline{\mathsf{annot}}, c' \rangle$ *is a certified solution.*

PROOF. By definition of $\overline{\mathsf{annot}}$, one sees that c' defined as

$$c'(l) = \begin{cases} c(l) & \text{if } l \in \mathsf{dom}(c) \\ \mathsf{axiom}(\overline{\mathsf{annot}}(l)) & \text{otherwise} \end{cases}$$

is such that $\langle \overline{\mathsf{annot}}, c' \rangle$ is a certified solution. □

In the sequel, we shall abuse language and speak about certified solutions of the form $\langle \mathsf{annot}, c \rangle$ where annot is a partial labeling and c is an indexed family of certificates that establish all verification conditions of annot.

3.6 Concrete Semantics

Although the technical development of the paper only refers to abstract semantics, we provide a definition of program concrete semantics, and recall the definition of soundness of abstract semantics w.r.t. concrete semantics. The concrete semantics of programs is modeled as a transition relation between states.

Definition 3.12 *Concrete semantics*. Let Env be a set of environments. For every $s \in$ Stmt we assume given a transition relation $(\!|s|\!) \subseteq \mathsf{Env} \times \mathsf{Env}$. We define the set of states as State $= \mathcal{N} \times$ Env, and the transition relation $\rightsquigarrow \subseteq$ State $\times$ State by the clause $\langle l, \eta \rangle \rightsquigarrow \langle l', \eta' \rangle$ if and only if $(\eta, \eta') \in (\!|G(l, l')|\!)$.

The reflexive and transitive closure of the relation $\rightsquigarrow$ is denoted by $\rightsquigarrow^\star$. Moreover, we say that a program P with initial memory η, evaluates to η', written $P, \eta \Downarrow_{l_o} \eta'$, iff $\langle l_{\mathsf{init}}, \eta\rangle \rightsquigarrow^\star \langle l_o, \eta'\rangle$ and $l_o \in \mathcal{O}$.

For the statement language μStmt, we take Env to be the set of mappings from variables to integer values, that is, $\mathsf{Env} = \mathsf{Var} \to \mathbb{Z}$, where Var is an infinite set of variables. The concrete semantics is defined using the semantics of expressions, that assigns to each expression e and environment η a value $(\!|e|\!)_\eta$. Formally, the concrete semantics is given by the clauses:

— $(\eta, \eta') \in (\!|\mathsf{assert}\ b|\!)$ iff $\eta = \eta'$ and $(\!|b|\!)_\eta = \mathsf{true}$;
— $(\eta, \eta') \in (\!|x := e|\!)$ iff $\eta' = [\eta : x \mapsto n]$, where $n = (\!|e|\!)_\eta$—and $[\eta : x \mapsto n]$ is the unique mapping such that

$$[\eta : x \mapsto n](y) = \begin{cases} n & \text{if } x = y \\ \eta(y) & \text{otherwise} \end{cases}$$

— $(\eta, \eta') \in (\!|\mathsf{skip}|\!)$ iff $\eta = \eta'$; and
— $(\eta, \eta') \in (\!|s_1; s_2|\!)$ iff there exists η'' s.t. $(\eta, \eta'') \in (\!|s_1|\!)$ and $(\eta'', \eta') \in (\!|s_2|\!)$.

We now define the soundness of an abstract semantics w.r.t. the concrete semantics.

Definition 3.13 *Soundness*. Let $\mathbf{A}$ be an abstract domain and $\langle [\![.]\!], f\rangle$ be an abstract semantics over A. Let $\models_A \subseteq \mathsf{Env} \times A$ be a satisfaction relation. The abstract semantics is sound (w.r.t. the concrete semantics and $\models_A$) iff for all $s \in \mathsf{Stmt}$, $a \in A$ and $\eta, \eta' \in \mathsf{Env}$ such that $(\eta, \eta') \in (\!|s|\!)$:

— $f = \mathsf{bwd}$ and $\models_A \eta : [\![s]\!]a$ implies $\models_a \eta' : a$
— $f = \mathsf{fwd}$ and $\models_A \eta : a$ implies $\models_a \eta' : [\![s]\!]a$.

If an abstract semantics is sound w.r.t. the concrete semantics, then the solutions for a program P provide a sound abstraction of its concrete semantics.

PROPOSITION 3.14 SOUNDNESS OF ABSTRACT INTERPRETATION. *Let* $\mathbf{A}$ *be an abstract domain and* $\langle [\![.]\!], f\rangle$ *be an abstract semantics over* A. *Let* $\models_A \subseteq \mathsf{Env} \times A$ *be a satisfaction relation. Assume that* $\langle [\![.]\!], f\rangle$ *is sound over the concrete semantics and* $\models_A$. *Then for every program* $P = \langle \mathcal{N}, \mathcal{E}, G\rangle$, *solution* S *for* P, *and edges* $(l, l') \in \mathcal{E}$ *such that* $\langle l, \eta\rangle \rightsquigarrow \langle l', \eta'\rangle$, *if* $\models_A \eta : S(l)$ *then* $\models_A \eta' : S(l')$.

It follows from Proposition 3.1.4 that if $P, \eta \Downarrow_{l_o} \eta'$ and S is a solution for P, then $\models_A \eta : S(l_{\mathsf{init}})$ entails $\models_A \eta' : S(l_o)$. When instantiated to program logics, the statement coincides with the usual formulation of soundness.

4. CERTIFYING ANALYZERS

This section addresses the problem of automatically producing certified solutions from solutions. More concretely, our goal is to transform solutions that are inferred by an automatic program analysis, into certified solutions that can be verified using deductive methods. Therefore, we consider abstract domains $\mathbf{A}$ and $\mathbf{A}^\sharp$ that are related by a monotone function $\gamma : A^\sharp \to A$, and abstract semantics $\langle [\![.]\!]_A, f\rangle$ and $\langle [\![.]\!]_{A^\sharp}, f^\sharp\rangle$ over $\mathbf{A}$ and $\mathbf{A}^\sharp$ respectively. The concretization function γ maps static analysis results to the logical representation of the deductive framework. Assuming that $\mathbf{A}$ is endowed with a proof algebra, we want to transform every solution S for a program P over $\langle [\![.]\!]_{A^\sharp}, f^\sharp\rangle$ into a certified solution $\langle \gamma \circ S, c\rangle$ for P over $\langle [\![.]\!]_A, f\rangle$. Our main result gives sufficient conditions for the existence and automated construction of certified solutions. In the next section, we explain how to use the result of certifying analyzers in certificate translation algorithms.

We begin by defining the soundness of an abstract semantics with respect to another.

Definition 4.1 *Soundness*. Let $\mathbf{A}$ and $\mathbf{A}^\sharp$ be abstract domains. Let $\gamma : A^\sharp \to A$ be a monotone function. An abstract semantics $\langle [\![.]\!]_{A^\sharp}, f^\sharp \rangle$ over $\mathbf{A}^\sharp$ is sound (along γ) with respect to an abstract semantics $\langle [\![.]\!]_A, f \rangle$ over $\mathbf{A}$ iff for all $a \in A^\sharp$ and $s \in \mathsf{Stmt}$, the inequality $\Phi_{s,a}$ holds, where $\Phi_{s,a}$ is defined as follows:

— if $f = f^\sharp = \mathsf{fwd}$ then $\Phi_{s,a}$ is $[\![s]\!]_A(\gamma(a)) \sqsubseteq \gamma([\![s]\!]_{A^\sharp}(a))$;
— if $f = f^\sharp = \mathsf{bwd}$ then $\Phi_{s,a}$ is $[\![s]\!]_A(\gamma(a)) \sqsupseteq \gamma([\![s]\!]_{A^\sharp}(a))$;
— if $f = \mathsf{fwd}$ and $f^\sharp = \mathsf{bwd}$ then $\Phi_{s,a}$ is $[\![s]\!]_A(\gamma([\![s]\!]_{A^\sharp}(a))) \sqsubseteq \gamma(a)$;
— if $f = \mathsf{bwd}$ and $f^\sharp = \mathsf{fwd}$ then $\Phi_{s,a}$ is $[\![s]\!]_A(\gamma([\![s]\!]_{A^\sharp}(a))) \sqsupseteq \gamma(a)$.

We now turn to certifying analyzers. Assuming that the abstract domain $\mathbf{A}$ is endowed with a proof algebra, one can consider instead of soundness the stronger notion of provable soundness.

Definition 4.2 *Provable soundness*. Let $\mathbf{A}$ and $\mathbf{A}^\sharp$ be abstract domains, and let $\gamma : A^\sharp \to A$. Assume that $\mathbf{A}$ is endowed with a proof algebra. An abstract semantics $\langle [\![.]\!]_{A^\sharp}, f^\sharp \rangle$ over $\mathbf{A}^\sharp$ is provably sound (along γ) w.r.t. an abstract semantics $\langle [\![.]\!]_A, f \rangle$ over $\mathbf{A}$ iff the following holds:

— for every $a, a' \in A^\sharp$ s.t. $a \sqsubseteq^\sharp a'$, there exists the certificate:

$$\mathsf{monot}_\gamma(a, a') : \mathcal{C}(\gamma(a) \sqsubseteq \gamma(a'))$$

— for every statement s and for every $b, b' \in A$ there exist the certificate function:

$$\mathsf{monot}_{[\![s]\!]_A}(b, b') : \mathcal{C}(b \sqsubseteq b') \to \mathcal{C}([\![s]\!]_A(b) \sqsubseteq [\![s]\!]_A(b'))$$

— for every $a \in A^\sharp$ and $s \in \mathsf{Stmt}$, there exists a certificate $\mathsf{cons}_s(a)$ of $\Phi_{s,a}$, where $\Phi_{s,a}$ is defined as in Definition 4.1.

PROPOSITION 4.3 EXISTENCE OF CERTIFYING ANALYZERS. *Let* $\mathbf{A}$ *and* $\mathbf{A}^\sharp$ *be abstract domains, and let* $\langle [\![.]\!]_A, f \rangle$ *and* $\langle [\![.]\!]_{A^\sharp}, f^\sharp \rangle$ *be abstract semantics over* $\mathbf{A}$ *and* $\mathbf{A}^\sharp$ *respectively. Let* $\gamma : A^\sharp \to A$ *and assume that* $\langle [\![.]\!]_{A^\sharp}, f^\sharp \rangle$ *is provably sound (along* γ*) over* $\langle [\![.]\!]_A, f \rangle$. *Then, every solution for* P *over* $\langle [\![.]\!]_{A^\sharp}, f^\sharp \rangle$ *can be transformed into a certified solution for* P *over* $\langle [\![.]\!]_A, f \rangle$.

PROOF. The proof proceeds by showing a step-by-step construction of the certificates for $\gamma \circ S$ from the certificates assumed by hypothesis and applying the functions of the proof algebra. The certificates are defined from the certificates cons, in Figure 9. In the figure, s stands for $G(l, l')$ and T and $T^\sharp$ stand for $[\![G(l, l')]\!]_{\mathbf{A}}$ and $[\![G(l, l')]\!]_{\mathbf{A}^\sharp}$, respectively. Consider for instance the case in which $f = f^\sharp = \mathsf{fwd}$. As hypothesis, since S is a solution over $\langle [\![.]\!]_{A^\sharp}, f^\sharp \rangle$ we have that $T^\sharp(S(l)) \sqsubseteq^\sharp S(l')$. From monotony of γ, we have a certificate of $\gamma(T^\sharp(S(l))) \sqsubseteq \gamma(S(l'))$, from which we obtain a certificate of the weaker constraint $\gamma(T^\sharp(S(l))) \sqcap T(\gamma(S(l))) \sqsubseteq \gamma(S(l'))$ by application of the proof algebra function $\mathsf{weak}_\sqcap$. Since the abstract semantics $\langle [\![.]\!]_{A^\sharp}, f^\sharp \rangle$ is proved to be sound w.r.t. $\langle [\![.]\!]_A, f \rangle$, we have a certificate for $T(\gamma(S(l))) \sqsubseteq \gamma(T^\sharp(S(l)))$, which is combined with a certificate of $\gamma(T^\sharp(S(l))) \sqcap T(\gamma(S(l))) \sqsubseteq \gamma(S(l'))$ to obtain a certificate for $T(\gamma(S(l))) \sqsubseteq \gamma(S(l'))$. The final certificate for $\bigsqcup_{l:(l,l')\in\mathcal{E}} T(\gamma(S(l))) \sqsubseteq \gamma(S(l'))$ is obtained then by a repeated application of the $\mathsf{intro}_\sqcup$ operator over every predecessor of l'. The remaining cases are symmetrical. □

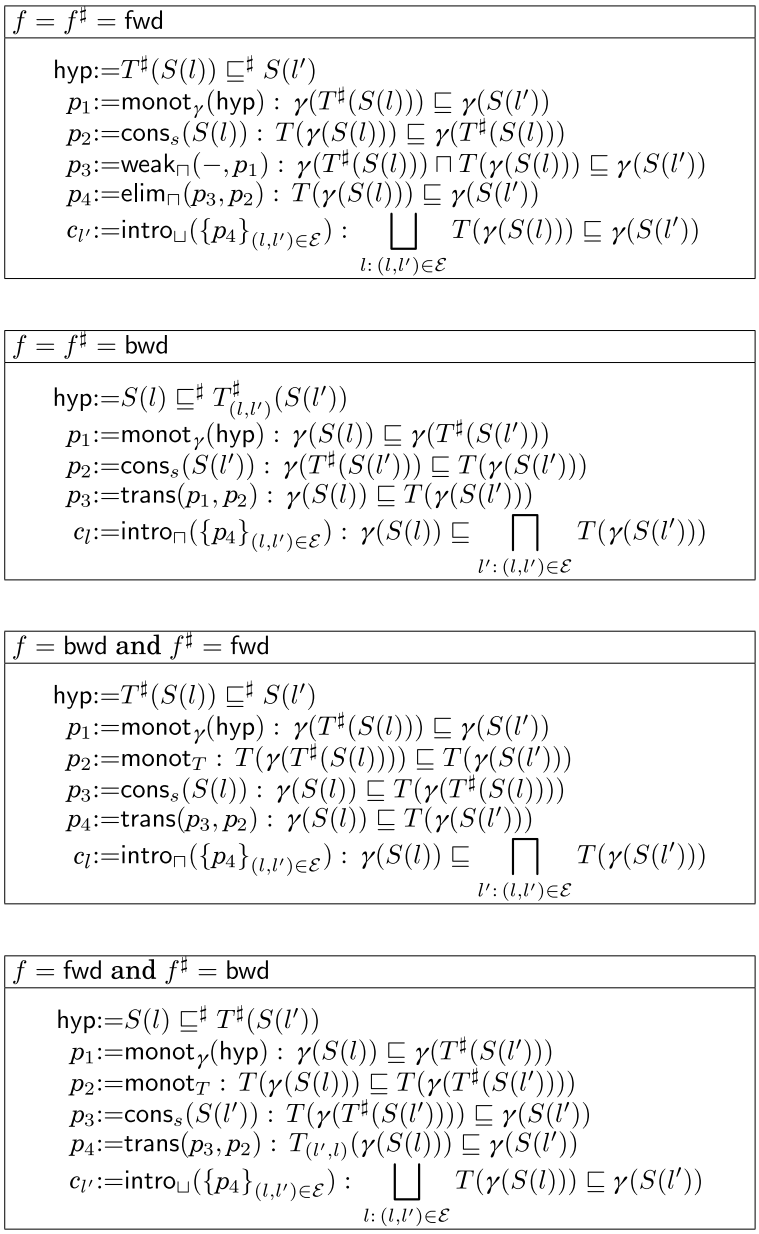

Fig. 9. Construction of a certifying analyzer.

The remaining of this section is devoted to illustrate Proposition 4.3. We consider the certification of a forward analysis with respect to a weakest precondition and strongest postcondition calculus, and the certification of a weakest precondition with respect to a strongest postcondition calculus.

Consider the forward abstract semantics $[\![.]\!]_{A^\sharp}$ over the lattice $(\mathsf{Var} \to \mathsf{Expr})_\perp$, where Expr is the flat join semi-lattice of expressions: that is, the set of expressions plus the $\top$ element. Let $\lfloor e \rfloor_\rho$ be defined as a substitution in e of every variable v s.t. $\rho v \neq \top$ by

ρv. Suppose that $\mathbf{A}$ is the lattice of logical propositions and that γ maps elements of $(\mathsf{Var} \to \mathsf{Expr})_\perp$ to formulae as defined by the clauses:

$$\gamma(\perp) = \mathsf{false}$$
$$\gamma(\rho) = \bigwedge_{\rho v \neq \top} v = \rho v$$

In this particular setting it is not difficult to provide a certificate

$$\mathsf{monot}_\gamma(\rho, \rho') : \gamma(\rho) \Rightarrow \gamma(\rho')$$

for every abstract environments ρ, ρ' such that $\rho \sqsubseteq_A \rho'$. Indeed, notice that the set of terms in the conjunction $\gamma(\rho')$ is a subset of the terms in $\gamma(\rho)$. A procedure to build a certificate $\mathsf{monot}_\gamma(\rho, \rho')$ consists of an iterative application of the $\mathsf{weak}_\sqcap$ operator over the trivial certificate $\mathsf{axiom}(\gamma(\rho')) : \mathcal{C}(\gamma(\rho') \Rightarrow \gamma(\rho'))$:

$$\mathsf{weak}_\sqcap(x_1 = \rho x_1, \ldots \mathsf{weak}_\sqcap(x_k = \rho x_k, \mathsf{axiom}(\gamma(\rho'))) \ldots) : \mathcal{C}(\gamma(\rho) \Rightarrow \gamma(\rho')),$$

where the set of variables $\{x_1, .., x_k\}$ is $\{x \mid \rho x \neq \top \wedge \rho' x = \top\}$.

The soundness of the abstract semantics $\langle [\![.]\!]_{A^\sharp}, \mathsf{fwd}\rangle$ w.r.t. a weakest precondition and a strongest postcondition verification setting is established by proving

$$\gamma(\rho) \Rightarrow \mathsf{wp}(s)(\gamma([\![s]\!]\rho)) \qquad \mathsf{sp}(s)(\gamma(\rho)) \Rightarrow \gamma([\![s]\!]\rho)$$

for all abstract environment $\rho \in A$ and statement s, respectively. Consider, for instance, the statement $x := y + z$ and abstract environment ρ with $\rho x = 5$, $\rho z = 7$, and $\rho v = \top$ for any other variable v. To verify the abstract analysis w.r.t. the weakest precondition calculus, from definition of $[\![.]\!]$, one must prove that

$$\bigwedge_{\rho v \neq \top} v = \rho v \Rightarrow \mathsf{wp}(x := y + z)(x = \lfloor y + z \rfloor_\rho \wedge \bigwedge_{\rho v \neq \top \wedge v \neq x} v = \rho v),$$

which from definition of wp is equivalent to

$$x = 5 \wedge z = 7 \Rightarrow y + z = y + 7 \wedge z = 7.$$

Assume we have for all expressions e, e_1, e_2, e_1', e_2' the certificate $\mathsf{eq}(e) : \mathcal{C}(e = e)$ and the certificate function $\mathsf{plus\_eq} : \mathcal{C}(e_1 = e_1') \to \mathcal{C}(e_2 = e_2') \to \mathcal{C}(e_1 + e_2 = e_1' + e_2')$. Then the certificate for the proof obligation above can be defined as

$$\lambda\langle c, c'\rangle : \mathcal{C}(x = 5 \wedge z = y).\langle \mathsf{plus\_eq}(\mathsf{eq}(y), c'), c'\rangle.$$

In Section 5, we make use of the certificates introduced in this section to certify the partial analysis labeling shown in Figure 15. From Proposition 4.3 and the existence of the certificates monot_γ, $\mathsf{monot}_{\mathsf{wp}}$, and cons, there exists a certified solution $\langle S, c\rangle$ such that S associates the assertion $y = 2 * p$ to the node l_1, the assertion $y' = 2 * p \wedge x = x'$ to the nodes $\{l_2', l_3', l_5'\}$ and true to any other node. In the next section, we make the implicit assumption that the analysis results used in the examples are certified.

To verify the abstract analysis w.r.t. the strongest postcondition calculus, one must prove that

$$\mathsf{sp}(x := y + z)(\bigwedge_{\rho v \neq \top} v = \rho v) \Rightarrow x = \lfloor y + z \rfloor_\rho \wedge \bigwedge_{\rho v \neq \top \wedge v \neq x} v = \rho v,$$

which from definition of sp is equivalent to

$$\exists x'.\, x' = 5 \wedge z = 7 \wedge x := y + z \Longrightarrow x = y + 7 \wedge z = 7.$$

Proposition 4.3 also considers the certification of verification settings at the same level of abstraction, i.e., with $\gamma : A \to A^\sharp$ defined as the identity function. For example,

to verify that a weakest precondition calculus is sound w.r.t. a strongest postcondition calculus, one must show for every formula ϕ and statement s that $\mathsf{sp}(s)(\mathsf{wp}(s)\,\phi) \Rightarrow \phi$. For the assignment $x := e$, one must discharge proof obligations of the form

$$\exists x'.\ x = e[{}^{x'}/_{x}] \wedge \phi[{}^{e}/_{x}][{}^{x'}/_{x}] \Rightarrow \phi.$$

5. CERTIFICATE TRANSLATION

In this section, we provide sufficient conditions for the existence for certificate translators, that map certificates of a program P into certificates of another program P', derived from P by a program transformation. Rather than attempting to prove a general result where P and P' are related in some complex manner, we establish three results for basic transformations that can be used in combination to cover many cases of interest.

In a first instance, Section 5.1 considers a program transformation that consists in duplicating fragments of the graph representation of P, modeling transformations such as loop unrolling and function inlining. In a second instance, Section 5.2 requires that the transformed program P' is a subgraph of the original program P. This is the case, for example, when P' is derived from P by applying optimizations such as constant propagation or common subexpression elimination. In a third instance, in Section 5.3, we abstract away some of the structure of the program to deal with optimizations that do not preserve so tightly the structure of programs, such as code motion. Finally, we generalize certificate translation to cover optimizations such as dead variable elimination.

5.1 Code Duplication

Some program transformations change the program size with no effect in the execution performance, enabling further program optimizations. Typical cases of code duplication are loop unrolling and function inlining. In this section, we model this class of transformations as a duplication of fragments of the program graph representation. Consider for instance the case of loop unrolling, in which one or more iterations of the loop body are executed separately before entering the loop:

l_1: $x := 0$;	l_a: $x := 0$;
l_2: **while** b **do**	l_b: **if** b **then**
s	s;
l_3: **done**	l_c: **while** b **do**
	s
	done
	l_d: **fi**

In the program at the right, we duplicate one iteration of the statement s representing the loop body. It is executed under the condition b in order to preserve the original program semantics. In the transformed program, the unrolled iteration of the loop body s is executed with the variable x holding the value 0, a condition that may be invalidated by the successive loop iterations. To illustrate this transformation in our program representation, consider the Figure 10. The graph on the left shows the structure of the original program, where node l_2 represents the head of a loop, and the edge (l_2, l_2) the execution of the loop body. The graph on the right shows the result of unrolling the first execution of the loop body. When execution reaches node l_b, the transition (l_b, l_c) (representing one execution of the loop body) is followed if the loop guard is satisfied;

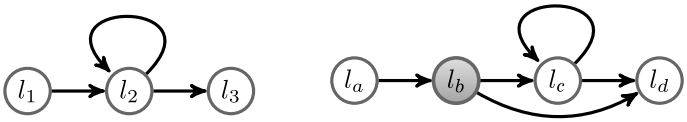

Fig. 10. Loop unrolling example.

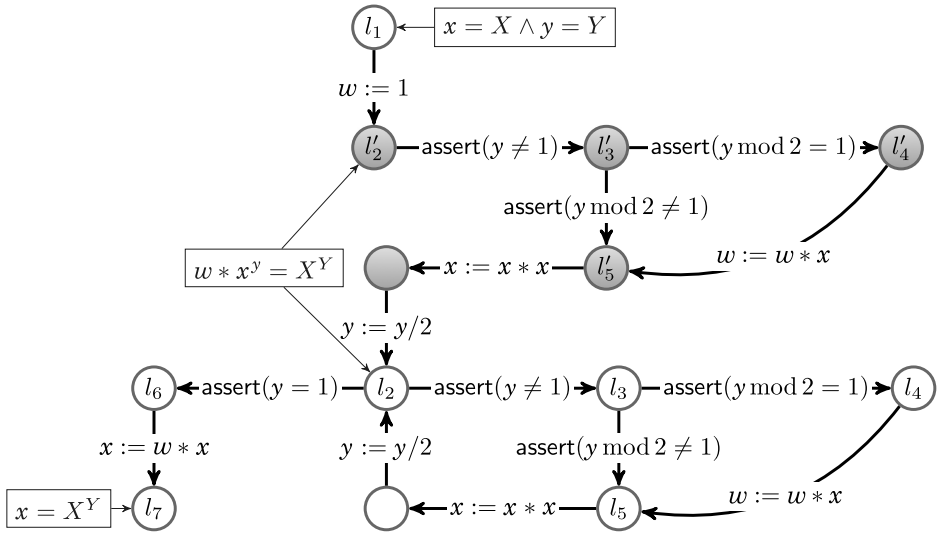

Fig. 11. Original program after loop unrolling.

then the remaining loop iterations may proceed. If the loop guard is not satisfied, then the execution continues to node l_d.

In the rest of this section, we formalize code duplication in our abstract setting, and show how to deal with certificate translation for this class of transformations.

Definition 5.1 *Node duplication*. A program $\mathring{P} = \langle \mathring{\mathcal{N}}, \mathring{\mathcal{E}}, \mathring{G} \rangle$ is the result of duplicating nodes of program $P = \langle \mathcal{N}, \mathcal{E}, G \rangle$ if there exists a surjective function $\pi : \mathring{\mathcal{N}} \to \mathcal{N}$ such that for every $(l, l') \in \mathring{\mathcal{E}}$

— $(\pi\, l, \pi\, l') \in \mathcal{E}$ and
— $\mathring{G}(l, l') = G(\pi\, l, \pi\, l')$.

Notice from this definition that $\langle l_1, \sigma_1 \rangle, \langle l_2, \sigma_2 \rangle, \ldots, \langle l_k, \sigma_k \rangle$ is an execution trace in $\mathring{P}$ only if $\langle \pi\, l_1, \sigma_1 \rangle, \langle \pi\, l_2, \sigma_2 \rangle, \ldots, \langle \pi\, l_k, \sigma_k \rangle$ is a trace in P.

Example 5.2. Figure 11 shows the result of applying loop unrolling: nodes l_2, l_3, l_4 and l_5 are duplicated into the nodes l'_2, l'_3, l'_4, and l'_5, respectively, and a new subset of edges is defined accordingly. Formally, the duplicated graph is such that $\mathcal{N} \subseteq \mathring{\mathcal{N}}$ and and a projection function $\pi : \mathring{\mathcal{N}} \to \mathcal{N}$ can be defined as $\pi\, l'_2 = l_2$, $\pi\, l'_3 = l_3$, $\pi\, l'_4 = l_4$, $\pi\, l'_5 = l_5$, and $\pi\, l = l$ for any other node.

PROPOSITION 5.3. *Let* $\mathbf{A}$ *be an abstract domain,* $I = \langle [\![.]\!], f \rangle$ *an abstract semantics over* $\mathbf{A}$*, and* $\mathcal{C}$ *a proof algebra for* $\mathbf{A}$*. Assume the certificates of Figure 12 exist for every abstract elements* $a_1, a_2, b_1, b_2 \in A$*. Then every certified solution* $\langle \mathsf{annot}, c \rangle$ *for* P *can be*

$$\mathsf{monot}_{[\![s]\!]} : a_1 \sqsubseteq a_2 \rightarrow [\![s]\!](a_1) \sqsubseteq [\![s]\!](a_2)$$
$$\mathsf{distr}^{\leftarrow}_{([\![s]\!],\sqcap)} : [\![s]\!](a_1) \sqcap [\![s]\!](a_2) \sqsubseteq [\![s]\!](a_1 \sqcap a_2)$$
$$\mathsf{distr}^{\rightarrow}_{([\![s]\!],\sqcap)} : [\![s]\!](a_1 \sqcap a_2) \sqsubseteq [\![s]\!](a_1) \sqcap [\![s]\!](a_2)$$
$$\mathsf{assoc}^{\leftarrow}_{\sqcap} : a_1 \sqcap (b_1 \sqcap b_2) \sqsubseteq (a_1 \sqcap b_1) \sqcap b_2$$
$$\mathsf{assoc}^{\rightarrow}_{\sqcap} : (a_1 \sqcap b_1) \sqcap b_2 \sqsubseteq a_1 \sqcap (b_1 \sqcap b_2)$$
$$\mathsf{commut}_{\sqcap} : a_1 \sqcap a_2 \sqsubseteq a_2 \sqcap a_1$$

Fig. 12. Certificates required for certificate translation.

transformed into a certified solution $\langle \mathring{\mathsf{annot}}, \mathring{c} \rangle$ *for* $\mathring{P}$*, where* $\mathring{\mathsf{annot}}(l) = \mathsf{annot}(\pi\, l)$ *for all* $l \in \mathring{\mathcal{N}}$ *s.t.* $\pi\, l \in \mathsf{dom}(\mathsf{annot})$.

PROOF. We proceed by induction, using the principle derived from the fact that annot is a sufficient annotation. More concretely, one can attach to every node a weight that corresponds to the length of the longest path to an annotated node, that is, a node l in dom(annot).

To generate a certificate for the transformed program it is sufficient to provide the following certificates:

— $\mathsf{goal}(l) : \overline{\mathsf{annot}}(\pi\, l) \sqsubseteq \overline{\mathring{\mathsf{annot}}}(l)$, if $f = \mathsf{bwd}$, or
— $\mathsf{goal}(l) : \overline{\mathring{\mathsf{annot}}}(l) \sqsubseteq \overline{\mathsf{annot}}(\pi\, l)$, if $f = \mathsf{fwd}$,

for all node $l \in \mathring{\mathcal{N}}$. Indeed, for instance when $f = \mathsf{bwd}$, one can make use of the certificate $\mathsf{monot}_{[\![.]\!]}$ in Figure 12 to generate certificates of the form

$$\mathring{\mathsf{annot}}(l) \sqsubseteq \bigsqcap_{(l,l') \in \mathring{\mathcal{E}}} \overline{\mathring{\mathsf{annot}}}(l')$$

from the original certificates

$$\mathsf{annot}(l) \sqsubseteq \bigsqcap_{(l,l') \in \mathcal{E}} \overline{\mathsf{annot}}(l'),$$

the definition $\mathring{\mathsf{annot}}(l) = \mathsf{annot}(\pi l)$, and the certificate $\mathsf{goal}(l)$ for all $l \in \mathsf{dom}(\mathsf{annot})$, and the certificate operator $\mathsf{monot}_{[\![.]\!]}$.

We describe now the construction of certificate goal. For $l \in \mathring{\mathcal{N}}$ such that $\pi\, l$ is in dom(annot), the certificate goal is trivial by definition of $\mathring{\mathsf{annot}}$, that is, an application of the axiom operation of the proof algebra. For every $l \in \mathring{\mathcal{N}}$ s.t. $\pi\, l \notin \mathcal{N}$, the certificate $\mathsf{goal}(l)$ is defined by the following derivation steps:

$$p(l') := \mathsf{goal}(l') : \overline{\mathsf{annot}}(\pi\, l') \sqsubseteq \overline{\mathring{\mathsf{annot}}}(l')$$
$$q(l') := \mathsf{monot}_{[\![s]\!]}(p(l')) : [\![s]\!](\overline{\mathsf{annot}}(\pi\, l')) \sqsubseteq [\![s]\!](\overline{\mathring{\mathsf{annot}}}(l'))$$
$$r(l') := \mathsf{weak}_{\sqcap}(q(l')) : \textstyle\bigsqcap_{(\pi\, l, \pi\, l') \in \mathcal{E}} [\![s]\!](\overline{\mathsf{annot}}(\pi\, l')) \sqsubseteq [\![s]\!](\overline{\mathring{\mathsf{annot}}}(l'))$$
$$\mathsf{goal}(l) := \mathsf{intro}_{\sqcap}(\{r(l')\}_{(l,l') \in \mathring{\mathcal{E}}}) : \textstyle\bigsqcap_{(\pi\, l, \pi\, l') \in \mathcal{E}} [\![s]\!](\overline{\mathsf{annot}}(\pi\, l')) \sqsubseteq \bigsqcap_{(l,l') \in \mathring{\mathcal{E}}} [\![s]\!](\overline{\mathring{\mathsf{annot}}}(l')),$$

where s stands for $[\![\mathring{G}(l_1, l_2)]\!]$ and for any label l' with $(l, l') \in \mathring{\mathcal{E}}$, $\mathsf{goal}(l')$ stands an application for the inductive hypothesis.

The expression $\mathsf{weak}_{\sqcap}(X)$ stands for $\mathsf{weak}_{\sqcap}(c_1, \mathsf{weak}_{\sqcap}(\{c_2, \ldots, c_k\}))$ for any sequence $X = \{c_1, c_2, \ldots, c_k\}$ (and similarly with $\mathsf{intro}_{\sqcap}(X)$). The inductive step for the forward case is similar. □

Example 5.4. Consider again the program example shown in Figure 11, which is the result of unrolling the loop of the original program shown in Figure 8. Assuming that we have a certificate for the original program, we show how to obtain a certificate for the transformed program.

First, the partial specification labeling is defined. In this case, the labeling is extended by setting $\mathring{\mathsf{annot}}(l'_2) = \mathsf{annot}(l_2)$, and $\mathring{\mathsf{annot}}(l) = \mathsf{annot}(l)$ for any other l in $\mathsf{dom}(\mathsf{annot})$. As the structure of the transformed program is not modified significantly, the computation of a total labeling $\mathring{\mathsf{annot}}$ is not affected either. A visual inspection on the graph of Figure 11 should be sufficient to deduce that the graph structure is almost preserved. Indeed, with the exception of the duplicated node l'_2, for the duplicated nodes $l \in \mathring{\mathcal{N}}$ such that there is an edge $(\pi l, l') \in \mathcal{E}$ we have a corresponding edge $(l, l'') \in \mathring{\mathcal{E}}$ with $\pi l'' = l'$. For the duplicated node l'_2, there is the original edge (l_2, l_6) but not a corresponding edge (l'_2, l_6): only the edge (l'_2, l'_3) is present in the transformed program. However, one can see that this condition does not affect the computation of the total labeling $\overline{\mathring{\mathsf{annot}}}$ from the partial labeling $\mathring{\mathsf{annot}}$, as l'_2 is an annotated node (i.e., $l'_2 \in \mathsf{dom}(\mathring{\mathsf{annot}})$). The lack of edge (l'_2, l_6) does affect, however, the computation of requirements for $\mathring{\mathsf{annot}}$ to be a solution, as

$$\mathring{\mathsf{annot}}(l'_2) \Rightarrow \mathsf{wp}(\mathring{G}(l'_2, l'_3))\overline{\mathring{\mathsf{annot}}}(l'_3)$$

is now required instead of the original constraint

$$\mathsf{annot}(l_2) \Rightarrow \mathsf{wp}(G(l_2, l_3))\overline{\mathsf{annot}}(l_3) \wedge \mathsf{wp}(G(l_2, l_6))\overline{\mathsf{annot}}(l_6).$$

From Proposition 5.3, we have a certificate for the goal $\overline{\mathsf{annot}}(l_3) \sqsubseteq \overline{\mathring{\mathsf{annot}}}(l'_3)$. The existence of the certificate for $\mathring{\mathsf{annot}}(l'_2) \Rightarrow \mathsf{wp}(\mathring{G}(l'_2, l'_3))\overline{\mathring{\mathsf{annot}}}(l'_3)$ follows from the definitions $\mathsf{annot}(l_2) = \mathring{\mathsf{annot}}(l'_2)$ and $\mathring{G}(l'_2, l'_3) = G(l'_2, l'_3)$.

5.2 Edge Transformation

In this section, we consider a fundamental class of program transformations. These transformations represent the effect of replacing statements when the result of an analysis ensures that the semantics is preserved. This is the case, for instance, for constant propagation, in which a static analysis infers whether at a given program point some variable always holds the same constant value. In a second step, constant propagation optimizes statements assuming the validity of the analysis results. Consider the following fragment of code:

$$\begin{aligned} &l_1 : x := 0; \\ &l_2 : \textbf{if } b \textbf{ then } y := y + x;\ x := x + 1 \textbf{ fi} \\ &l_3 : \textbf{while } b \textbf{ do } y := y + x;\ x := x + 1 \textbf{ done} \end{aligned}$$

Assume a static analysis infers that, for every program execution, x holds the value 0 at the program point l_2. Then, the first occurrence of the statement $y := y + x; x := x + 1$ can be replaced by the semantically equivalent statement $\mathsf{skip};\ x := 1$. This program transformation is reflected in the abstract program graph by a transformation on the statements associated to the corresponding edges. Moreover, we also consider the case in which some of the original edges are removed.

We first provide a formal characterization of the transformations under consideration. We then state the existence of certificate translators based on the certifiability of the analysis result that motivates the transformation.

Let $a = S(l)$, $a' = S(l')$, $T = [\![G(l,l')]\!]$ and $T' = [\![G'(l,l')]\!]$ in:

$$
\begin{aligned}
p_1 &:= \mathsf{goal}(l') : a' \sqcap \overline{\mathsf{annot}}(l') \sqsubseteq \overline{\mathsf{annot}}'(l') \\
p_2 &:= \mathsf{monot}_{T'}(p_1) : T'(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
p_3 &:= \mathsf{justif}_{(l,l')} : a \sqcap T(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq T'(a' \sqcap \overline{\mathsf{annot}}(l')) \\
p_5 &:= \mathsf{elim}_{\sqcap}(\mathsf{weak}_{\sqcap}(-, p_2), p_3) : a \sqcap T(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
p_6 &:= \mathsf{distrib}_T : T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq T(a' \sqcap \overline{\mathsf{annot}}(l')) \\
p_7 &:= \mathsf{axiom} : a \sqsubseteq a \\
p_8 &:= \mathsf{intro}_{\sqcap}(\mathsf{weak}_{\sqcap}(p_7), \mathsf{weak}_{\sqcap}(p_6)) : a \sqcap T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq a \sqcap T(a' \sqcap \overline{\mathsf{annot}}(l')) \\
p_9 &:= \mathsf{elim}_{\sqcap}(\mathsf{weak}_{\sqcap}(p_5), p_8) : a \sqcap T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
p_{10} &:= c_l^S : a \sqsubseteq T(a') \\
p_{11} &:= \mathsf{elim}_{\sqcap}(p_9, p_{10}) : a \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
p_{12} &:= \mathsf{weak}_{\sqcap}(p_{11}) : a \sqcap \textstyle\bigsqcap_{(l,l') \in \mathcal{E}} T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
\mathsf{goal}(l) &:= \mathsf{intro}_{\sqcap}(\{p_{12}\}_{(l,l') \in \mathcal{E}}) : a \sqcap \textstyle\bigsqcap_{(l,l') \in \mathcal{E}} T(\overline{\mathsf{annot}}(l')) \sqsubseteq \bigsqcap_{(l,l') \in \mathcal{E}} T'(\overline{\mathsf{annot}}'(l'))
\end{aligned}
$$

Fig. 13. Definition of $\mathsf{goal}(l)$ for certificate translation (case $f = \mathsf{bwd}$).

Let P be a program $\langle \mathcal{N}, \mathcal{E}, G \rangle$. The program $P' = \langle \mathcal{N}', \mathcal{E}', G' \rangle$ is a transformation of P if P' is a subgraph of P such that $\mathcal{N}' \subseteq \mathcal{N}$ and $\mathcal{E}' \subseteq \mathcal{E}$.

The following result states that it is possible to translate the certificates as long as there exists a certificate justif establishing a relation between the abstract semantics of the original and transformed programs. Assume that we have a certified representation of the analysis result that enables the program transformation. The existence of certifying analyzers follows from the results of Section 4.

PROPOSITION 5.5 EXISTENCE OF CERTIFICATE TRANSLATORS. *Let $\langle S, c^S \rangle$ be a certified solution over I such that for every $(l_1, l_2) \in \mathcal{E}'$ and $a \in A$:*

— *if* $f = \mathsf{bwd}$ *then* $\mathsf{justif}_{(l_1,l_2)} : S(l_1) \sqcap [\![G(l_1,l_2)]\!](a) \sqsubseteq [\![G'(l_1,l_2)]\!](a)$*;*
— *if* $f = \mathsf{fwd}$ *then* $\mathsf{justif}_{(l_1,l_2)} : [\![G'(l_1,l_2)]\!](a) \sqsubseteq S(l_2) \sqcap [\![G(l_1,l_2)]\!](a)$.

Then, provided the certificates in Figure 12 are given for every $a_1, a_2, b_1, b_2 \in A$, one can transform every certified labeling $\langle \mathsf{annot}, c \rangle$ for P into a certified labeling $\langle \mathsf{annot}', c' \rangle$ for P', where $\mathsf{annot}'(l)$ is defined as $\mathsf{annot}(l) \sqcap S(l)$ for every node l in $\mathsf{dom}(\mathsf{annot}')$, that is, in $\mathsf{dom}(\mathsf{annot}) \cap \mathcal{N}'$.

PROOF. We build for every l in $\mathcal{N}'$ the certificate

— $\mathsf{goal}(l) : S(l) \sqcap \overline{\mathsf{annot}}(l) \sqsubseteq \overline{\mathsf{annot}}'(l)$ if $f = \mathsf{bwd}$, or
— $\mathsf{goal}(l) : \overline{\mathsf{annot}}'(l) \sqsubseteq S(l) \sqcap \overline{\mathsf{annot}}(l)$ if $f = \mathsf{fwd}$,

from which the existence of a certificate for annot' follows.

We proceed by induction, using the principle derived from the fact that annot is a sufficient annotation. More concretely, one can attach to every node a weight that corresponds to the length of the longest path to an annotated node, that is, a node $l \in \mathsf{dom}(\mathsf{annot})$. In the base case, where $l \in \mathsf{dom}(\mathsf{annot}')$, the certificate $\mathsf{goal}(l)$ is defined trivially, since $\overline{\mathsf{annot}}'(l) = S(l) \sqcap \overline{\mathsf{annot}}(l)$. For the inductive step, where $l \notin \mathsf{dom}(\mathsf{annot}')$, the proof is given in Figures 13 and 14 for the backward and forward case, respectively. The application of certificates $\mathsf{assoc}_{\sqcap}^{\leftarrow}$, $\mathsf{assoc}_{\sqcap}^{\rightarrow}$ and $\mathsf{commut}_{\sqcap}$ is omitted in the proof for readability. □

The set of certificates justif required for certificate translation in Proposition 5.5 must be provided for each program optimization. It is thus an additional task based on the definition of each optimization, and it can be interpreted as a proof of soundness for each local transformation. Consider for instance the standard verification setting

Let $a = S(l)$, $a' = S(l')$, $T = [\![G(l',l)]\!]$ and $T' = [\![G'(l',l)]\!]$ in:

$$
\begin{aligned}
p_1{:=}&\mathsf{goal}'(l') : \overline{\mathsf{annot}}'(l') \sqsubseteq a' \sqcap \overline{\mathsf{annot}}(l')\\
p_2{:=}&\mathsf{monot}_{T'} : T'(\overline{\mathsf{annot}}'(l')) \sqsubseteq T'(a' \sqcap \overline{\mathsf{annot}}(l'))\\
p_3{:=}&\mathsf{justif} : T'(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq a \sqcap T(a' \sqcap \overline{\mathsf{annot}}(l'))\\
p_4{:=}&\mathsf{distrib}_T : T(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq T(a') \sqcap T(\overline{\mathsf{annot}}(l'))\\
p_5{:=}&\mathsf{weak}_{\sqcap}(p_4) : a \sqcap T(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq T(a') \sqcap T(\overline{\mathsf{annot}}(l'))\\
p_6{:=}&c^S_{(l',l)} : T(a') \sqsubseteq a\\
p_7{:=}&\mathsf{weak}_{\sqcap}(p_6) : T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq a \sqcap T(\overline{\mathsf{annot}}(l'))\\
p_8{:=}&\mathsf{trans}(p_2, \mathsf{trans}(p_3, \mathsf{trans}(p_5,))) : T'(\overline{\mathsf{annot}}'(l')) \sqsubseteq a \sqcap T(\overline{\mathsf{annot}}(l'))\\
p_9{:=}&\mathsf{weak}_{\sqcup}(\mathsf{axiom}) : T(\overline{\mathsf{annot}}(l')) \sqsubseteq \bigsqcup_{(l,l')\in\mathcal{E}} T(\overline{\mathsf{annot}}(l'))\\
p_{10}{:=}&\mathsf{intro}_{\sqcap}(\mathsf{weak}_{\sqcap}(\mathsf{axiom}), \mathsf{weak}_{\sqcap}(p_9)) :\\
&\qquad\qquad a \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq a \sqcap \bigsqcup_{(l,l')\in\mathcal{E}} T(\overline{\mathsf{annot}}(l'))\\
p_{11}{:=}&\mathsf{trans}(p_8, p_{10}) : T'(\overline{\mathsf{annot}}'(l')) \sqsubseteq a \sqcap \bigsqcup_{(l',l)\in\mathcal{E}} T(\overline{\mathsf{annot}}(l'))\\
P_{12}{:=}&\mathsf{intro}_{\sqcup}(\{p_{13}\}_{(l',l)\in\mathcal{E}'}) : \bigsqcup_{(l',l)\in\mathcal{E}} T'(\overline{\mathsf{annot}}'(l')) \sqsubseteq a \bigsqcup_{(l',l)\in\mathcal{E}} T(\overline{\mathsf{annot}}(l'))
\end{aligned}
$$

Fig. 14. Definition of $\mathsf{goal}(l)$ for certificate translation (case $f = \mathsf{fwd}$).

based on weakest precondition calculus. Let s be a program statement and s' its transformed version. The predicate justif entails verifying that the proof obligation $\psi \wedge \mathsf{wp}(s,\varphi) \Rightarrow \mathsf{wp}(s',\varphi)$ holds for every φ, where ψ represents the condition that makes the transformation of s into s' a semantics preserving optimization. Assume, for instance, the statement $s \doteq x{:=}y{+}z$ that is optimized to $s' \doteq x{:=}y{+}7$ based on the condition $\psi \doteq z = 7$. The proof obligation justif can be interpreted as the semantics soundness of the transformation: consider an initial state σ satisfying $z = 7$, if for all σ' such that $(\sigma,\sigma') \in [\![x{:=}y{+}z]\!]$ we have $\sigma' \in \varphi$ then for all σ'' such that $(\sigma,\sigma'') \in [\![x{:=}y{+}7]\!]$ we have $\sigma'' \in \varphi$.

The rest of certificates required in Proposition 5.5 are not dependent on the program transformation but on the underling abstract domain. For simple domains as the value analysis example of the next section, the constraints hold trivially and no certificate is needed. In the lattice of logical formulae, $\mathsf{assoc}^{\leftarrow}_{\wedge}$, $\mathsf{assoc}^{\rightarrow}_{\wedge}$, and $\mathsf{commut}_{\wedge}$ are simply built from the associativity and commutativity of logical conjunction: for every ϕ, ψ, and φ we require a certificate for $\phi \wedge \psi \Rightarrow \psi \wedge \phi$ and $\phi \wedge (\psi \wedge \varphi) \Rightarrow (\phi \wedge \psi) \wedge \varphi$ (and the reciprocal of the latter). Discharging the distributivity of the wp operator over conjunctions of logical formulae, i.e. defining $\mathsf{distr}^{\leftarrow}_{(\mathsf{wp},\wedge)}$ and $\mathsf{distr}^{\rightarrow}_{(\mathsf{wp},\wedge)}$ entails verifying the equivalence of $\mathsf{wp}(s, \phi \wedge \phi')$ and $\mathsf{wp}(s,\phi) \wedge \mathsf{wp}(s,\phi')$, which can be easily done in our setting by simple case analysis in the statement s.

Using the results of Proposition 4.3, Proposition 5.5 can be instantiated to prove the existence of certificate transformers for many common optimizations considered in previous work [Barthe et al. 2009]. These include, in particular, optimizations that can be classified as substitutions on the expressions defining the program, such as constant and copy propagation, common subexpression elimination, and induction variable strength reduction. In a nutshell, enabling certificate translation consists in first discharging the proof obligations necessary to make the analyzer certifying, as explained in Section 4. Based on the computed analysis result, the compiler optimizes the program. Then, one provides a certificate justif_e for each edge e whose statement has been modified. The translation procedure then merges these certificates with the original one to produce a certificate for the optimized program. The following section illustrates further this process for a particular instance of edge transforming optimizations.

Optimizations based on expression simplification. In this section, we instantiate the transformation described above with a class of optimizations based on expression simplification.

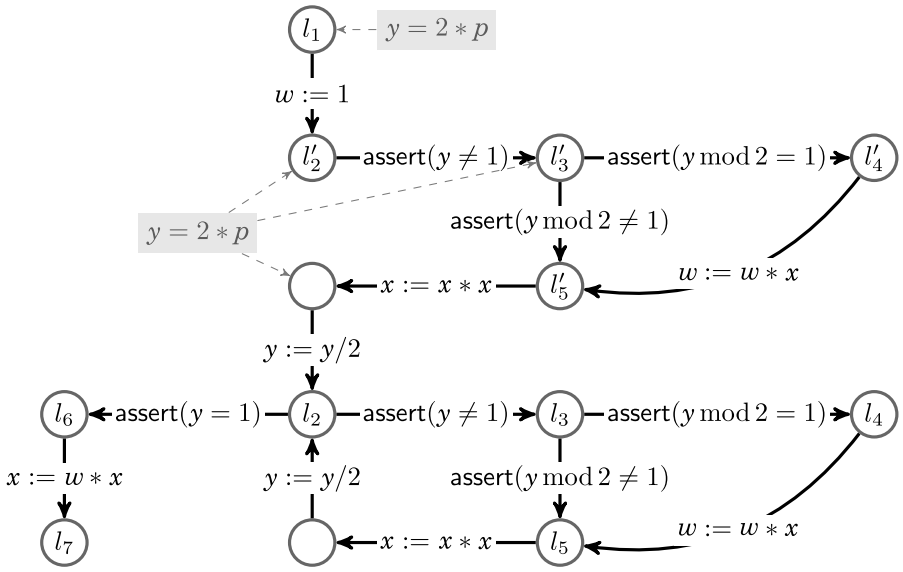

Fig. 15. Program analysis after loop unrolling.

To simplify the presentation, we illustrate this kind of optimizations with a simple transformation applied to the running example. Induction variable strength reduction is postponed to next section, since it must be defined as a combination with transformations described in the following sections.

We now define the program transformation. Let $\lfloor e \rfloor_\rho$ stand for the substitution in e of every variable v s.t. $\rho\, v \neq \top$ by $\rho\, v$. For a program $\langle \mathcal{N}, \mathcal{E}, G \rangle$ we define the optimized program as $\langle \mathcal{N}, \mathcal{E}, G' \rangle$, such that for every $e \in \mathcal{E}$ the statement $G'(e)$ is defined as:

— $\lfloor b \rfloor_\rho$ if $G(e) = \mathsf{assert}\ b$, and
— $a := \lfloor e \rfloor_\rho$ or $a[\lfloor e' \rfloor_\rho] := \lfloor e \rfloor_\rho$ if $G(e)$ is equal to the assignments $x := e$ and $a[e'] := e$, respectively.

Example 5.6. Consider the program in Figure 15. Suppose that we know (e.g., from the execution context) that the program is called with the variable y satisfying the precondition $y = 2 * p$. Then, one can consider a static analysis that propagates this information forward. As shown in the figure, an analyzer infers that the condition $y = 2 * p$ can be propagated to the nodes l'_2, l'_3, l'_4, and l'_5. One can check that a labeling S defined as $S(l) \doteq y = 2 * p$ for $l \in \{l_{\mathsf{init}}, l'_2, l'_3, l'_4, l'_5\}$, and $S(l) \doteq \top$ for any other node $l \in \mathcal{N}$ is a valid solution. In a forward data-flow analysis this entails checking for instance that $S(l_2)$, which is trivial since the statement $G(l'_2, l'_3)$ represents the identity over the abstract value $y = 2 * p$ (and similarly with nodes l'_3, l'_4, and l'_5):

$$G(l'_2, l'_3)\ S(l'_2) \sqsubseteq S(l'_3).$$

In the rest of this section we assume, however, that the analyzer is certifying. That is, the abstract value $y = 2 * p$ is an actual logical formula, and there is a certificate of the correctness of the specification S w.r.t. a weakest precondition verification setting.

Figure 16 shows a transformed version of the program of Figure 15, in which the variable y' has been replaced by the equivalent expression $2 * p$. We now describe the requirements for certificate transformation, instantiated to the particular analysis, verification framework, and program transformation considered in this case.

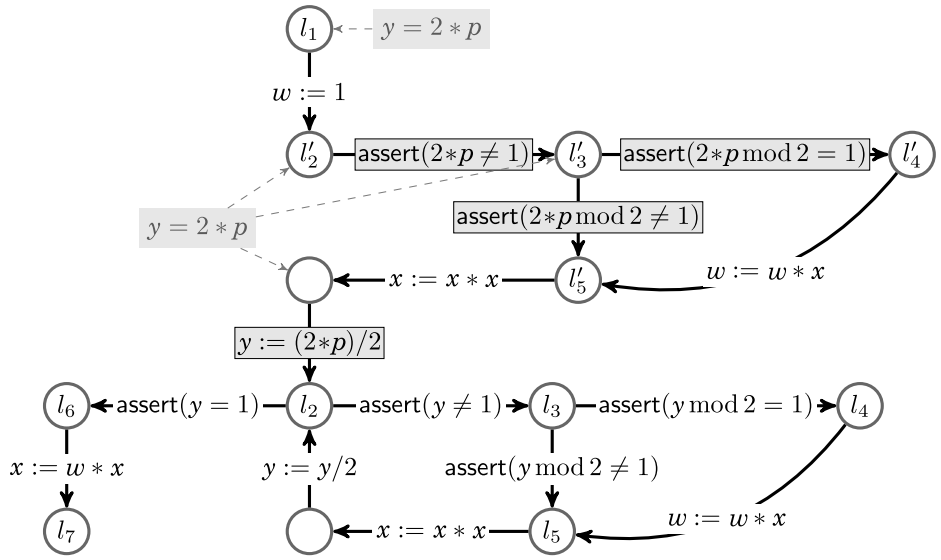

Fig. 16. Program analysis after first transformation step.

Consider the proof obligation that states that the transformation preserves the verification result, assuming valid the result of the analysis. More precisely, for every formula ϕ and edge (l, l'), one must provide a certificate

$$\mathsf{justif} : S'(l) \wedge \mathsf{wp}\,(G(l,l'))\,\phi \Rightarrow \mathsf{wp}\,(G'(l,l'))\,\phi,$$

where $S' = \gamma \circ S$. To discharge this proof obligation one proceeds by case analysis on $G((l,l'))$ and $S(l)$. Let ρ stand for $S(l)$. Consider for instance an edge (l,l') such that $G((l,l')) = x := e$. Then, from the definition of the program transformation, we have $G'((l,l')) = x := \lfloor e \rfloor_\rho$. From the definition of $\lfloor . \rfloor_\rho$, it is straightforward to prove $\gamma(\rho) \Rightarrow e = \lfloor e \rfloor_\rho$ by variable rewriting. By definition, $\mathsf{wp}\,(G(l,l'))\,\phi$ and $\mathsf{wp}\,(G'(l,l'))\,\phi$ are equal to $\phi[{}^{e}/_{x}]$ and $\phi[{}^{\lfloor e \rfloor_\rho}/_{x}]$, respectively. Therefore, in this case, the proof obligation justif is

$$\gamma(\rho) \wedge \phi[{}^{e}/_{x}] \Rightarrow \phi[{}^{\lfloor e \rfloor_\rho}/_{x}],$$

which can be proved simply by expression rewriting as well. By Proposition 5.5, from the existence of the certificate justif shown above, one can build a certificate for the optimized program in Figure 16, with labeling $\mathsf{annot}'(l) = \mathsf{annot}(l) \wedge S(l)$ for all nodes $l \in \mathsf{dom}(\mathsf{annot})$.

Figure 17 shows the final version of the optimized program, in which trivially decidable jump statements have been eliminated (nodes l'_2 and l'_3) and unreachable nodes have been removed (node l'_4), and in which assignments have been simplified (node l'_5). The corresponding transformation of the certificate is not explained since it relies on a trivial analysis result ($S(l) = \mathsf{true}$ for all $l \in \mathcal{N}$).

5.3 Program Skeletons

Proposition 5.5 requires that the transformation is justified for each edge of the program; this rules out several well known optimizations such as statement swapping or code motion, whose justification involves more than one statement. Consider, for example, a program $\langle \mathcal{N}, \mathcal{E}, G \rangle$ with $\mathcal{N} = \{l_1, l_2, l_3\}$, $\mathcal{E} = \{(l_1, l_2), (l_2, l_3)\}$, and $G(l_1, l_2) = c_1$

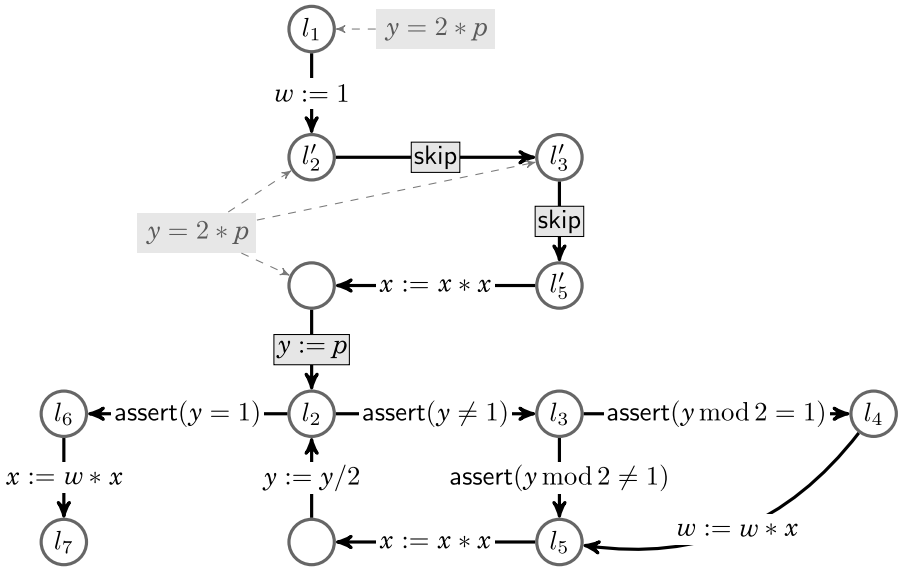

Fig. 17. Program after optimizing transformations.

and $G(l_2, l_3) = c_2$. Suppose that $(\!|c_1; c_2|\!) = (\!|c_2; c_1|\!)$, that is, that the order in which they are executed does not alter the final state. Let the program $\langle \mathcal{N}, \mathcal{E}, G' \rangle$ be the result of swapping statements c_1 and c_2, that is $G'(l_1, l_2) = c_2$ and $G'(l_2, l_3) = c_1$. Certificate translation for this transformation as an instance of the class of optimizations in Section 5.2, requires proving that $(\!|c_1|\!) = (\!|c_2|\!)$, which does not necessarily hold. To overcome this limitation, one must abandon the direct representation of programs, where each edge represents one statement, and cluster several statements into a single edge.

In this section, we formally capture this idea of clustering, and use it to extend the applicability of the results of Section 5.2. We define a program skeleton by selecting a set $\hat{\mathcal{N}} \subseteq \mathcal{N}$ of program points to be preserved in the final graph. The final graph preserves the program structure, but abstracts away the program points represented by the nodes that are not in $\hat{\mathcal{N}}$. In the following paragraphs, we formally define a program $\hat{P} = \langle \hat{\mathcal{N}}, \hat{\mathcal{E}}, \hat{G} \rangle$ as a skeleton of program $P = \langle \mathcal{N}, \mathcal{E}, G \rangle$. For notational simplicity, we assume the set of statements closed under sequential composition.

Definition 5.7. Let $P = \langle \mathcal{N}, \mathcal{E}, G \rangle$ be a program. Let l be a node with a single input node l_1 and a single output node l_2, and assume $(l_1, l_2) \notin \mathcal{E}$. The program $P \setminus \{l\}$ is defined as $\langle \hat{\mathcal{N}}, \hat{\mathcal{E}}, \hat{G} \rangle$, where $\hat{\mathcal{N}} = \mathcal{N} \setminus \{l\}$, $\hat{\mathcal{E}} = \mathcal{E} \setminus \{(l_1, l), (l, l_2)\} \cup \{(l_1, l_2)\}$, $\hat{G}(l_a, l_b)$ is equal to $G(l_a, l_b)$, and $\hat{G}(l_1, l_2) = G(l_1, l); G(l, l_2)$ for all $(l_a, l_b) \neq (l_1, l_2)$. A program $\hat{P}$ is a skeleton of P if there exists $l_1 \ldots l_k$ s.t. $\hat{P}$ is equal to $P \setminus \{l_1\} \ldots \setminus \{l_k\}$.

In the rest of the section we let $\hat{P} = \langle \hat{\mathcal{N}}, \hat{\mathcal{E}}, \hat{G} \rangle$ be a skeleton of the program $P = \langle \mathcal{N}, \mathcal{E}, G \rangle$. One can show that for every $l, l' \in \hat{\mathcal{N}}$, $\langle l, \eta \rangle \rightsquigarrow^{\star} \langle l', \eta' \rangle$ in $\hat{P}$ only if $\langle l, \eta \rangle \rightsquigarrow \langle l', \eta' \rangle$ in P.

Let $I = \langle [\![.]\!]_{\mathbf{A}}, f \rangle$ be an abstract semantics over a domain $\mathbf{A}$. We say that I is closed under sequential composition if for all statements s_1 and s_2,

— $[\![s_1; s_2]\!] = [\![s_1]\!] \circ [\![s_2]\!]$. if f = bwd
— $[\![s_1; s_2]\!] = [\![s_2]\!] \circ [\![s_1]\!]$. if f = fwd.

LEMMA 5.8. *Let* $I = \langle [\![.]\!]_{\mathbf{A}}, f\rangle$ *be an abstract semantics. Suppose that* I *is closed under sequential composition, and let* S *be a labeling such that* $\mathsf{dom}(S) \subseteq \hat{\mathcal{N}}$. *Then* $\langle S, c\rangle$ *is a certified solution for* P *iff* $\langle S, c\rangle$ *is a certified solution for* $\hat{P}$.

The results of Section 5.2 are immediately extended to a broader set of proof transformations:

COROLLARY 1. *Assume* $\langle S, c\rangle$ *is a certified solution for* P *with* $\mathsf{dom}(S) \subseteq \hat{\mathcal{N}}$. *Let* $P' = \langle \mathcal{N}', \mathcal{E}', G'\rangle$ *be transformed from* P, *and* $\hat{P}' = \langle \hat{\mathcal{N}}', \hat{\mathcal{E}}', \hat{G}'\rangle$ *a skeleton of* P'. *Suppose that* $\hat{\mathcal{N}}' \subseteq \hat{\mathcal{N}}$ *and* $\hat{\mathcal{E}}' \subseteq \hat{\mathcal{E}}$ *and for every* $(l_1, l_2) \in \hat{\mathcal{E}}'$ *and* $a \in A$:

— *if* $f = \mathsf{bwd}$ *then* $\mathsf{justif}_{(l_1,l_2)} : S(l_1) \sqcap [\![\hat{G}(l_1, l_2)]\!](a) \sqsubseteq [\![\hat{G}'(l_1, l_2)]\!](a)$;
— *if* $f = \mathsf{fwd}$ *then* $\mathsf{justif}_{(l_1,l_2)} : [\![\hat{G}'(l_1, l_2)]\!](a) \sqsubseteq S(l_2) \sqcap [\![\hat{G}(l_1, l_2)]\!](a)$.

Then, provided the certificates in Figure 12 are given for every $a_1, a_2, b_1, b_2 \in A$, *one can provide a certified labeling* $\langle \mathsf{annot}', c'\rangle$ *for* P', *where* $\mathsf{annot}'(l)$ *is defined as* $\mathsf{annot}(l) \sqcap S(l)$ *for every node* l *in* $\mathsf{dom}(\mathsf{annot}') = \mathsf{dom}(\mathsf{annot}) \cap \mathcal{N}'$.

PROOF. Notice that $\hat{P}$ is a program as defined in Definition 3.1. From Proposition 5.8, $\langle S, c\rangle$ is a certified solution for P and $\langle \mathsf{annot}, c\rangle$ is a certified solution for $\hat{P}$. The corollary follows from the existence of the certificate justif and Lemma 5.5. □

Corollary 1 restates the result of Proposition 5.5, but requiring the certificate justif to be defined for every edge in $\hat{\mathcal{E}}$ instead of every edge in $\mathcal{E}$. Therefore, it is a generalization that covers a wider range of program transformations. Consider, for instance, the case of statement swapping, represented in the program graph by swapping the interpretation of the edges (l_1, l_2) and (l_2, l_3). In this case, the structure of the graph is preserved by the transformation, that is, $\mathcal{N}' = \mathcal{N}$ and $\mathcal{E}' = \mathcal{E}$, but we have $[\![G'(l_1, l_2)]\!] = [\![G(l_2, l_3)]\!]$ and $[\![G'(l_2, l_3)]\!] = [\![G(l_1, l_2)]\!]$. We cannot show a correspondence between $[\![Ge]\!]$ and $[\![G'e]\!]$ for any $e \in \{(l_1, l_2), (l_2, l_3)\}$. However, by abstracting the intermediate node l_2, we are in a position to analyze whether $[\![G'(l_1, l_2)]\!] \circ [\![G'(l_2, l_3)]\!]$ is equal to $[\![G(l_1, l_2)]\!] \circ [\![G(l_2, l_3)]\!]$, that is, whether $[\![\hat{G}'(l_1, l_3)]\!] = [\![\hat{G}(l_1, l_3)]\!]$ holds.

Example 5.10. We illustrate how to apply the result of this section to the program of Figure 17 to obtain the program of Figure 18. The program in the figure shows that, after a program optimization, the edges (l'_2, l'_3) and (l'_3, l'_5) represent program transitions that do not modify the execution state. That is modeled in the analysis domain by defining $[\![G(l'_2, l'_3)]\!]$ and $[\![G(l'_3, l'_5)]\!]$ as the identity function in A. Let P' be the result of removing the nodes l'_3 and l'_5, and replacing the edges (l'_2, l'_3), (l'_3, l'_5) and (l'_5, l_2) by a single edge (l'_2, l_2). The abstract interpretation is such that $[\![\hat{G}(l'_2, l_2)]\!]$ is equal to $[\![G(l'_5, l_2)]\!] \circ [\![G(l'_3, l'_5)]\!] \circ [\![G(l'_2, l'_3)]\!]$. However, this transformation is not considered in Section 5.2, since it is not a single-edge by single-edge replacement.

We can apply, however, a preliminary node clustering transformation before applying the results of Proposition 5.5. We define then the set of edges $\mathcal{N}_1$ to be removed as $\{l'_3, l'_5\}$; see Figure 18 for the result of node coalescing, plus a trivial statement transformation that removes skip statements. It is straightforward to transform the certificates after the removal of skip statements, by application of Proposition 5.5, since we have $\hat{\mathcal{E}} = \hat{\mathcal{E}}'$ and $[\![\hat{G}\, e]\!] = [\![\hat{G}'\, e]\!]$ for every $e \in \hat{\mathcal{E}}$. To that end, the result of the analysis S is defined as true for every annotated program node (that is, $\{l_1, l'_2, l_2, l_7\}$.) In the figure, we simplify the presentation by writing formulas of the form $\varphi \wedge \mathsf{true}$ as φ. For the translation of the certificate we require also the definition of the certificate justif, which in this case is trivially provided by the proof algebra operators axiom and $\mathsf{weak}_{\sqcap}$ since $\mathsf{wp}(\hat{G}\, e) = \mathsf{wp}(\hat{G}'\, e)$.

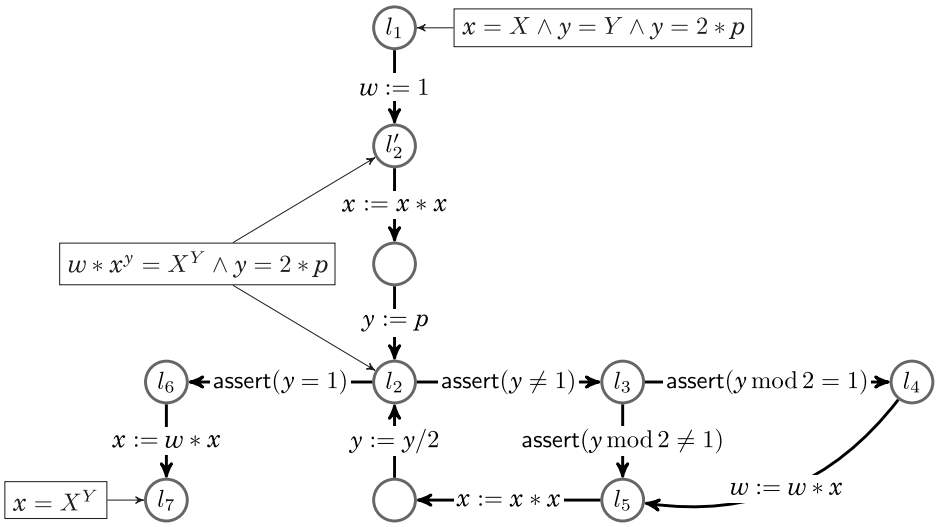

Fig. 18. Program after node coalescing.

Statement insertion. Consider a programming language as in the running example and a specification provided by a labeling S. We say that a variable is fresh if it does not occur in the program nor in the specification.

In our setting, the insertion of statements affecting fresh variables is modeled as the addition of new nodes and edges to the graph representation. The new variables must be chosen fresh not only with respect to the program syntax, but also with respect to previous analysis or verification results. In consequence, the transfer functions of the abstract interpretation that correspond to the inserted statements are defined as the identity in the domain of the specification (although there may be a wider abstract domain in which the transfer function is not the identity function).

This intuition is captured in the following result, which together with Corollary 1 enables us to establish the preservation of certified results in the presence of code insertion.

PROPOSITION 5.11. *Let $\langle \mathcal{N}, \mathcal{E}, G\rangle$ be a program and $I = \langle [\![.]\!], f\rangle$ be an abstract interpretation over the abstract domain $\mathbf{A}$. Let $\mathbf{A}'$ be a sublattice of $\mathbf{A}$ that is closed under $[\![Ge]\!]$ for all edge $e \in \mathcal{E}$. Then, the pair $\langle S, c\rangle$ such that $S(l) \in A'$ for every $l \in \mathcal{N}$, is a certified solution over the abstract domain $\mathbf{A}$ iff it is a certified solution over the abstract domain $\mathbf{A}'$.*

Consider for example the abstract interpretation $I = \langle \mathsf{wp}, \mathsf{bwd}\rangle$ over the lattice $\mathbf{A}$ of logical formulae, where wp is defined as a weakest precondition transformer. Consider a program $P = \langle \mathcal{N}, \mathcal{E}, G\rangle$ with $(l_1, l_2) \in \mathcal{E}$ and $\langle S, c\rangle$ a certified solution over I. Let P' be the result of inserting an extra program point between nodes l_1 and l_2 to represent an affectation to a fresh variable x, occurring neither in the program nor in the specification (that is, labeling S). One can formalize this transformation by setting $\mathcal{N}' = \mathcal{N} \cup \{l\}$, $\mathcal{E}' = (\mathcal{E} \setminus \{(l_1, l_2)\}) \cup \{(l_1, l), (l, l_2)\}$ and $G'(l_1, l) = G(l_1, l_2)$ and $G'(l_1, l)$ defined as an assignment to x. In this case, we cannot immediately apply Corollary 1, since $\hat{G}'(l_1, l_2)$ is not necessarily equal to $\hat{G}(l_1, l_2)$. Consider instead the abstract domain $\mathbf{A}_x$ defined as the sublattice of the logical formulae that do not contain the variable x. Since x is a fresh variable, it does not appear in $S(l)$ for any l, and for all $e \in \mathcal{E}$ the functions $\mathsf{wp}(G\ e)$ are closed on A_x. Then $\langle S, c\rangle$ is a certified solution for P over

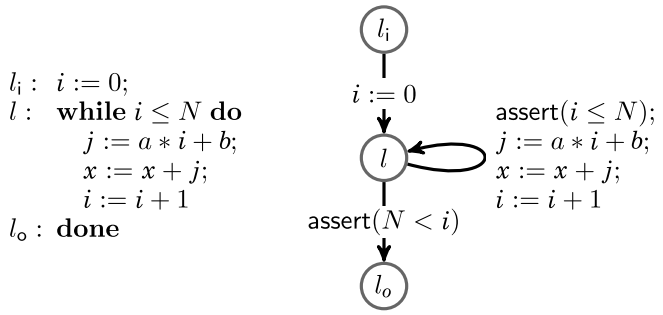

Fig. 19. Strength reduction example. Original program.

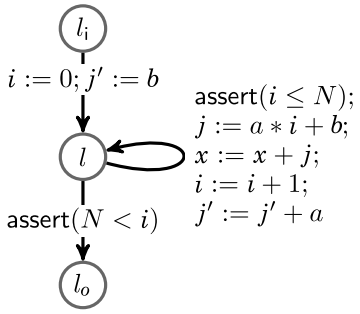

Fig. 20. Strength reduction example. Program after statement insertion.

the domain $\mathbf{A}_x$ by Proposition 5.11. We are now in a condition in which we can apply Corollary 1, since for every $\varphi \in A_x$, $[\![\hat{G}(l_1, l_2)]\!](\varphi) = [\![\hat{G}'(l_1, l_2)]\!](\varphi)$. Then, the pair $\langle S, c\rangle$ is a certified solution for P' over the abstract domain $\mathbf{A}_x$. And again by Proposition 5.11 it is also a certified solution for P' over the abstract domain $\mathbf{A}$.

Example 5.12 *Strength Reduction*. Induction variable strength reduction can be modeled as a composition of statement insertion and expression simplification. A basic induction variable is a variable that is incremented (or decremented) by a constant value in a program loop. A derived induction variable is a variable that is assigned in the loop an expression that is a linear function on a basic induction variable. In the code shown in Figure 19, i is a basic induction variable and j is an induction variable derived from i (assume that c does not modify neither the value of i nor j). Strength-reduction reduces the cost of computing the expression $a * i + b$, by computing an addition operation instead. The optimization can be modeled as a combination of three steps. The final certificate is defined by composition of the certificate translation procedures corresponding to each of the optimization steps.

Suppose we have a certified solution $\langle \mathsf{annot}, c\rangle$ for the program example. As in Barthe et al. [2009], we split the transformation in two steps. We first insert a pair of assignments to a fresh variable j'. We then use the value held by the variable j' to optimize the assignment $j := a * i + b$. The code in Figure 20 shows the result of inserting fresh statements as a first optimization step.

From Proposition 5.11, since j' is fresh w.r.t. both the program and the original specification annot, we do not need to transform the original certificate.

After inserting the assignments to the fresh variable j', a static analyzer can infer that the invariant $j' = a * i + b$ holds in the loop header. This is represented in our formalism as a labeling S such that $S(l) \doteq j' = a * i + b$. Certifying the correctness of

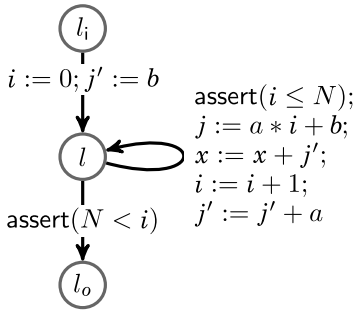

Fig. 21. Strength reduction example. Program after optimizing transformation.

this analysis result require discharging the proof obligations $\mathsf{true} \implies b = a * 0 + b$ and $j' = a*i+b \implies j'+a = a*(i+1)+b$, corresponding respectively to the establishment and preservation of $j' = a * i + b$ as invariant. The remaining program transformation is an instance of the class of optimizations based on expression simplification, as described in Section 5.2. That is, since $j = j'$ holds, one can substitute some occurrences of the variable j by j', returning the program shown in Figure 21. To transform the original certificate, from Proposition 5.5, one must provide a certificate justif for every statement that has been modified. This requires discharging the verification condition of the form

$$j' = a * i + b \wedge \varphi[^{x+a*i+b}/_x] \implies \varphi[^{x+j'}/_x].$$

A final step, which is left for Section 5.4, consists of removing the assignments to the now dead variable j.

5.4 Optimizations Based on Relational Analyses

This section provides a generalization of the results of Section 5.2 to allows one to deal with optimizations such as dead variable elimination. The generalization consists in relying on a generic composition operator instead of the lattice meet operator to merge two analysis results. The results of the previous sections are recovered by defining the composition as the meet operation of the lattice, which in logical terms amounts to strengthening the annotations with the results of the analysis. For the particular case of dead variable elimination, the composition operator is instantiated as a weakening of the original annotations.

At the end of this section we compare the method proposed for dead variable elimination to ghost variable introduction, an alternative technique developed in previous work [Barthe et al. 2009].

For the simple imperative language considered in this paper, a variable is live at a certain program point if the value it holds is used in at least one of the program points reachable from it. However, a typical definition of variable liveness is given in terms of the syntactic program representation: a variable x is live at a program node l if there is a path from l that reaches a statement that reads the value of x, without traversing an assignment to x.

Inspired by Benton's Relational Hoare Logic [Benton 2004], we follow a relational approach instead. Let X denote the set of variables that are *relevant* after the execution of a program. Given as input a set of variables X, a variable liveness analysis computes a set of variables Y in which two environments must coincide in order to reach corresponding environments that coincide in X. For instance, in order for two environments η and η' to coincide in the set $\{z\}$ after the execution of $x := y + z$, they

must coincide in $\{z\}$. If the final environments are required to coincide in $\{x, z\}$, then η and η' must coincide in $\{y, z\}$.

To express the results of the liveness analysis, we generalize the abstract domain A of the certificate infrastructure to include relational propositions. An abstract domain A is relational if the associated satisfaction relation $\models_A$ is a subset of $(\mathsf{Env} \times \mathsf{Env}) \times A$. Hence, a relational proposition will be interpreted as a relation on execution environments. In the rest of this section we assume that every abstract domain represents a relation on environments.

One particular relational domain consists of first-order formulae in which variables are decorated with indices of the form $\_\langle 1\rangle$ or $\_\langle 2\rangle$.

The interpretation of a relational formula φ over a pair (η_1, η_2) is defined by mapping every variable $v\langle 1\rangle$ in φ to $\eta_1 v$ and every variable $v\langle 2\rangle$ to $\eta_2 v$.

The predicate transformer wp must be redefined accordingly. For instance, $\mathsf{wp}(x := e)\ \phi$ is defined as the substitution $\phi[^{e\langle 1\rangle}/_{x\langle 1\rangle}][^{e\langle 2\rangle}/_{x\langle 2\rangle}]$, where $e\langle i\rangle$ is the result of indexing every variable occurring at e with $\_\langle i\rangle$.

Example 5.13. In this example we illustrate a liveness analysis result as a relational property. The analysis domain is defined as the powerset of program variables. The order relation of the abstract lattice is the inverse of the powerset lattice: $X \sqsubseteq Y$ iff $X \supseteq Y$. A set of variables X associated to a program label l represents the variables that are live at that program point. Under a relational interpretation, we say that two states η and η' are related by an abstract element (i.e., a set of variables) X if $\eta x = \eta' x$ for all $x \in X$.

In our programming language setting, the abstract semantics for liveness analysis is backwards and defined as follows:

$$[\![\mathsf{assert}(b)]\!]\, X = X \cup FV(b)$$
$$[\![v := e]\!]\, X = \begin{cases} (X \cup FV(e)) \setminus \{v\} & \text{if } v \in X \\ X & \text{otherwise.} \end{cases}$$

Consider the program in Figure 21. One can check that the labeling S defined as:

$$S(l_{\mathsf{in}}) = \{x\}$$
$$S(l) = \{i, j', x\}$$
$$S(l_{\mathsf{o}}) = \{x\}.$$

is a solution over the abstract semantics defined above. Indeed, it is clear that the following constraints hold

$$\begin{aligned} \{x\} &\sqsubseteq \{x\} = [\![i{:=}0; j'{:=}b]\!]\{x, i, j'\} \\ \{x, j', i\} &\sqsubseteq \{x, i\} = [\![\mathsf{assert}(N < i)]\!]\{x\} \\ \{x, j', i\} &\sqsubseteq \{x, j', i\} = [\![c]\!]\{x, j', i\}, \end{aligned}$$

where c denotes the loop body of the program in Figure 21.

As discussed in the previous sections, this analysis results can also be subject to certification. To that end, we first need to provide a convenient representation of abstract values: a set X of live variables is represented as the relational formula $\bigwedge_{v \in X} v\langle 1\rangle = v\langle 2\rangle$.

$$
\begin{array}{l}
\text{Let } a = \gamma(S(l)),\ a' = \gamma(S(l')),\ T = [\![G(l,l')]\!] \text{ and } T' = [\![G'(l,l')]\!] \text{ in:}\\
\quad p_1{:=}\mathsf{goal}(l') : a' \,.\, \overline{\mathsf{annot}}(l') \sqsubseteq \overline{\mathsf{annot}}'(l')\\
\quad p_2{:=}\mathsf{monot}_{T'}(p_1) : T'(a' \,.\, \overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l'))\\
\quad p_3{:=}\mathsf{justif} : a \,.\, T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(a' \,.\, \overline{\mathsf{annot}}(l'))\\
\quad p_4{:=}\mathsf{trans}(p_3,p_2) : a \,.\, T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l'))\\
\quad p_5{:=}\mathsf{axiom} : T(\overline{\mathsf{annot}}(l')) \sqsubseteq T(\overline{\mathsf{annot}}(l'))\\
\quad p_6{:=}\mathsf{axiom} : a \sqsubseteq a\\
\quad p_7{:=}\mathsf{weak}_{\sqcap}(p_5) : \bigsqcap_{(l,l')\in\mathcal{E}} T_{(l,l')}(\overline{\mathsf{annot}}(l')) \sqsubseteq T(\overline{\mathsf{annot}}(l'))\\
\quad p_8{:=}\mathsf{monot}_{\,.}(p_6,p_7) : a \,.\, \bigsqcap_{(l,l')\in\mathcal{E}} T_{(l,l')}(\overline{\mathsf{annot}}(l')) \sqsubseteq a \,.\, T(\overline{\mathsf{annot}}(l'))\\
\quad p_9{:=}\mathsf{trans}(p_8,p_4) : a \,.\, \bigsqcap_{(l,l')\in\mathcal{E}} T_{(l,l')}(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l'))\\
\mathsf{goal}(l){:=}\mathsf{intro}_{\sqcap}(\{p_9\}_{(l,l')\in\mathcal{E}}) : a \,.\, \bigsqcap_{(l,l')\in\mathcal{E}} T_{(l,l')}(\overline{\mathsf{annot}}(l')) \sqsubseteq \bigsqcap_{(l,l')\in\mathcal{E}} T'_{(l,l')}(\overline{\mathsf{annot}}'(l'))
\end{array}
$$

Fig. 22. Definition of goal for relational certificate translation (case f = bwd).

In this concrete example, the certification of the given analysis result requires the existence of certificates for the following proof obligations

$$
\begin{array}{l}
x_{\langle 1\rangle} = x_{\langle 2\rangle} \Longrightarrow x_{\langle 1\rangle} = x_{\langle 2\rangle} \wedge b = b \wedge 0 = 0\\
x_{\langle 1\rangle} = x_{\langle 2\rangle} \wedge i_{\langle 1\rangle} = i_{\langle 2\rangle} \wedge j'_{\langle 1\rangle} = j'_{\langle 2\rangle} \Longrightarrow i_{\langle 1\rangle} = i_{\langle 2\rangle} \wedge x_{\langle 1\rangle} + j'_{\langle 1\rangle} = x_{\langle 2\rangle} + j'_{\langle 2\rangle}\\
\qquad\qquad \wedge i_{\langle 1\rangle} + 1 = i_{\langle 2\rangle} + 1 \wedge j'_{\langle 1\rangle} + a = j'_{\langle 2\rangle} + a\\
x_{\langle 1\rangle} = x_{\langle 2\rangle} \wedge i_{\langle 1\rangle} = i_{\langle 2\rangle} \wedge j'_{\langle 1\rangle} = j'_{\langle 2\rangle} \Longrightarrow i_{\langle 1\rangle} = i_{\langle 2\rangle} \wedge x_{\langle 1\rangle} = x_{\langle 2\rangle}.
\end{array}
$$

Existence of certificate translators. Let $P = \langle \mathcal{N}, \mathcal{E}, G\rangle$ be a program, and $\langle \mathsf{annot}, c\rangle$ a certified solution for P, in the abstract interpretation $I = \langle [\![.]\!], f\rangle$ over the abstract domain $\mathbf{A}$. Let $P = \langle \mathcal{N}', \mathcal{E}', G'\rangle$ be the result of transforming edges in P, justified by a solution S over the abstract interpretation I. The following result generalizes Proposition 5.5 in terms of an arbitrary composition operator $\,.\, : A \times A \to A$.

PROPOSITION 5.14. *Let* $\,.\, : A \times A \to A$ *be a composition operator such that for every* $a_1, a_2, b_1, b_2 \in A$ *there exists a certificate*

$$\mathsf{monot}_{\,.} : a_1 \sqsubseteq a_2 \to b_1 \sqsubseteq b_2 \to a_1 \,.\, b_1 \sqsubseteq a_2 \,.\, b_2.$$

Let $\langle S, c^S\rangle$ *be a certified solution over* I *such that for every* $(l_1, l_2) \in \mathcal{E}'$ *and* $a \in A$*:*

— if f = bwd *then* $\mathsf{justif}_{(l_1,l_2)} : S(l_1) \,.\, [\![G(l_1,l_2)]\!](a) \sqsubseteq [\![G'(l_1,l_2)]\!](a \,.\, S(l_2))$*;*
— if f = fwd *then* $\mathsf{justif}_{(l_1,l_2)} : [\![G'(l_1,l_2)]\!](a \,.\, S(l_1)) \sqsubseteq S(l_2) \,.\, [\![G(l_1,l_2)]\!](a)$*.*

Then, provided the certificate $\mathsf{monot}_{[\![s]\!]}$ *defined in Figure 12 exists for all statement* s *and* $a_1, a_2 \in A$*, every certified labeling* $\langle \mathsf{annot}, c\rangle$ *for* P *can be transformed into a certified labeling* $\langle \mathsf{annot}', c'\rangle$ *for* P'*, where* $\mathsf{annot}'(l) = \mathsf{annot}(l).S(l)$ *for every node* l *in* $\mathsf{dom}(\mathsf{annot}') = \mathsf{dom}(\mathsf{annot}) \cap \mathcal{N}'$.

PROOF. The proof is similar to that of Proposition 5.5. The definition of the certificate goal such that

$$\mathsf{goal}(l) : a \,.\, \bigsqcap_{(l,l')\in\mathcal{E}} [\![G(l,l')]\!](\overline{\mathsf{annot}}(l')) \sqsubseteq \bigsqcap_{(l,l')\in\mathcal{E}} [\![G'(l,l')]\!](\overline{\mathsf{annot}}'(l')) \ \text{ if } f = \mathsf{bwd}$$

and

$$\mathsf{goal}(l) : \bigsqcup_{(l',l)\in\mathcal{E}} [\![G'(l',l)]\!](\overline{\mathsf{annot}}'(l')) \sqsubseteq a \,.\, \bigsqcup_{(l',l)\in\mathcal{E}} [\![G(l',l)]\!](\overline{\mathsf{annot}}(l')) \ \text{ if } f = \mathsf{fwd},$$

from which the existence of c' follows, is given in Figures 22 and 23. □

$$
\begin{array}{l}
\text{Let } a = \gamma(S(l)),\ a' = \gamma(S(l')),\ T = [\![G(l',l)]\!] \text{ and } T' = [\![G'(l',l)]\!] \text{ in:} \\
\quad p_1{:=}\mathsf{goal}(l') : \overline{\mathsf{annot}}'(l') \sqsubseteq a' \centerdot \overline{\mathsf{annot}}(l') \\
\quad p_2{:=}\mathsf{monot}_{T'}(p_1) : T'(\overline{\mathsf{annot}}'(l')) \sqsubseteq T'(a' \centerdot \overline{\mathsf{annot}}(l')) \\
\quad p_3{:=}\mathsf{justif} : T'(a' \centerdot \overline{\mathsf{annot}}(l')) \sqsubseteq a \centerdot T(\overline{\mathsf{annot}}(l')) \\
\quad p_4{:=}\mathsf{trans}(p_2, p_3) : T'(\overline{\mathsf{annot}}'(l')) \sqsubseteq a \centerdot T(\overline{\mathsf{annot}}(l')) \\
\quad p_5{:=}\mathsf{axiom} : T(\overline{\mathsf{annot}}(l')) \sqsubseteq T(\overline{\mathsf{annot}}(l')) \\
\quad p_6{:=}\mathsf{axiom} : a \sqsubseteq a \\
\quad p_7{:=}\mathsf{weak}_{\sqcup}(p_5) : T(\overline{\mathsf{annot}}(l')) \sqsubseteq \bigsqcup_{(l',l)\in\mathcal{E}} T_{(l',l)}(\overline{\mathsf{annot}}(l')) \\
\quad p_8{:=}\mathsf{monot}_{\centerdot}(p_6, p_7) : a \centerdot T(\overline{\mathsf{annot}}(l')) \sqsubseteq a \centerdot \bigsqcup_{(l',l)\in\mathcal{E}} T_{(l',l)}(\overline{\mathsf{annot}}(l')) \\
\quad p_9{:=}\mathsf{trans}(p_4, p_8) : T'(\overline{\mathsf{annot}}'(l')) \sqsubseteq a \centerdot \bigsqcup_{(l',l)\in\mathcal{E}} T_{(l',l)}(\overline{\mathsf{annot}}(l')) \\
\mathsf{goal}(l){:=}\mathsf{intro}_{\sqcup}(\{p_9\}_{(l',l)\in\mathcal{E}}) : \bigsqcup_{(l',l)\in\mathcal{E}} T'_{(l',l)}(\overline{\mathsf{annot}}'(l')) \sqsubseteq a \centerdot \bigsqcup_{(l',l)\in\mathcal{E}} T_{(l',l)}(\overline{\mathsf{annot}}(l'))
\end{array}
$$

Fig. 23. Definition of goal for relational certificate translation (case f = fwd).

Example 5.15. Consider again the liveness labeling S defined in Example 5.13. The set of live variables $\{i, j', x\}$ associated to label l, indicates that j is not live. Since the loop body at node l contains an assignment to the variable j and this variable is not live at node l, i.e. $j \notin S(l)$, we may safely simplify the loop body by removing such assignment. The transformed program is given in Figure 24.

By Proposition 4.3, we know that a certified solution $\langle \gamma \circ S, c'' \rangle$ exists, where S is the labeling defined in previous example.

In order to generate a certificate for the optimized program, we apply Proposition 5.14, using as $\centerdot$ operator the composition of relations:

$$\phi \centerdot \psi = \exists Z\,.\, \phi[{}^{Z}/_{X_{(2)}}] \wedge \psi[{}^{Z}/_{X_{(1)}}],$$

where X is the set of free variables in ϕ or ψ. Notice that, given a set of live variables X, one can equivalently define the composition $\gamma(X)\centerdot\phi$ as the existential quantification in ϕ of the variables that are not in X.

From Proposition 5.14 we can transform the current certificate if we provide a certificate for the following goal

$$\mathsf{justif}_{(l,l)} : \gamma(S(l)) \centerdot [\![G(l,l)]\!](\phi) \sqsubseteq [\![G'(l,l)]\!](\gamma(S(l)) \centerdot \phi).$$

If ϕ is a nonrelational proposition, $\gamma(X)\centerdot\phi$ is equivalently denoted $\exists y_1, \ldots, y_m.\ \phi$ where $\{y_1, \ldots, y_m\}$ is the set of free variables in ϕ that are not in X. Then, the goal of the certificate $\mathsf{justif}_{(l,l)}$ has the form

$$\vdash \varphi[{}^{a*i+b}/_{j}] \Rightarrow \exists j\, \varphi,$$

which can be easily discharged.

Comparison with ghost variable introduction. In this section we briefly review ghost variable introduction [Barthe et al. 2009], an alternative certificate translation technique in the presence of dead variable elimination. In the last paragraph, we position this alternative technique with the relational method explained earlier in this section.

For simplicity, we consider a particular analysis and verification framework: a liveness analysis that computes a set of live variables for each program point, and a weakest-precondition based verification framework over first-order formulas.

Consider a set of scalar variables $\mathcal{V}^g$ disjoint of $\mathcal{V}$. Assume that for every variable v in $\mathcal{V}$ there is a corresponding variable in $\mathcal{V}^g$, denoted v^g. Ghost variables are used only for specification purposes, and thus can never affect a program behavior.

To that end, one must restrict the occurrence of ghost variables in a program code. For instance, only ghost variables can be assigned the value of expressions containing

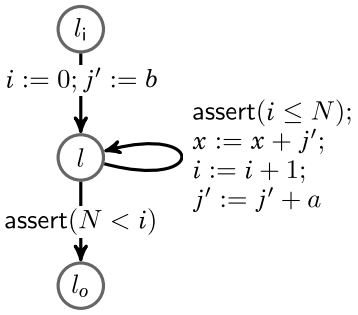

Fig. 24. Program after dead variable elimination.

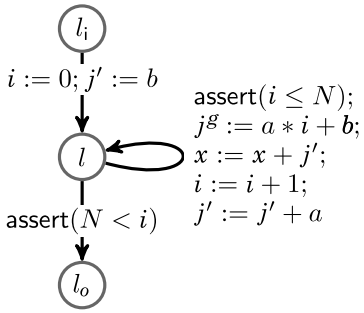

Fig. 25. Ghost variable introduction.

ghost variables. The following conditions characterize precisely the proper use of ghost variables in a program $\langle \mathcal{N}, \mathcal{E}, G \rangle$:

— if $G(l, l') = x := e$ with $x \in \mathcal{V}$ then e must not contain ghost variables,
— if $G(l, l') = a[e'] := e$, then ghost variables cannot occur neither in e nor in e', and
— if $G(l, l') = b$ then b does not refer to a ghost variable.

In the rest of this section we assume that every program satisfies the constraints above.

Ghost variable introduction [Barthe et al. 2009] consists of the replacement, instead of removal, of dead variables by ghost variables. Let $P = \langle \mathcal{N}, \mathcal{E}, G \rangle$ be a program. $\langle \mathcal{N}, \mathcal{E}, G' \rangle$ is the result of introducing ghost variables if for all $(l, l') \in \mathcal{E}$:

— if $G(l, l') = x := e$ then $G'(l, l') = \llbracket x \rrbracket_{S(l')} := \llbracket e \rrbracket_{S(l)}$, and
— in any other case $G'(l, l') = G(l, l')$.

where for a set of live variables X, $\llbracket e \rrbracket_X$ stands for the result of substituting in e every variable v that is not in X by v^g.

Example 5.16. Consider the liveness labeling S defined above in this section with $S(l) = \{x, i, j'\}$. Figure 25 shows the introduction of ghost variables to remove dead assignments in the program in Figure 21. The assignment $j^g := a * i + b$ is a proper use of ghost variables and, as part of the specification, has no effect in the program execution. Before analyzing how proof obligations change after ghost variable introduction in this example, we first anticipate a transformation on the original annotation. Let the specification annot^g for the transformed program be such that for all $l \in \mathcal{N}$ the assertion $\mathsf{annot}^g(l)$ is the result of substituting in $\mathsf{annot}(l)$ every variable v that is not live at l (i.e., variable j in this particular example) by the corresponding

variable v^g. Then, for the proof obligation enforcing the preservation of the loop invariant:

$$\mathsf{annot}(l) \Rightarrow N < i \implies \mathsf{annot}(l)[\cdot/_{\ldots}][^{a*i+b}/_{j}][\cdot/_{\ldots}]$$

we have the corresponding proof obligation after ghost variable introduction:

$$\mathsf{annot}^g(l) \Rightarrow N < i \implies \mathsf{annot}^g(l)[\cdot/_{\ldots}][^{a*i+b}/_{j'}][\cdot/_{\ldots}]$$

Independently of the assertion $\mathsf{annot}(l)$, notice that the new proof obligation coincides with the original one up to renaming of dead variables. In the following, we formalize this result, enabling to translate certificates after ghost variable introduction, and compare this technique with the results of Proposition 5.14.

LEMMA 5.17. *Let the labeling S represent the result of a variable liveness analysis. Let $\langle \mathcal{N}, \mathcal{E}, G' \rangle$ be the result of ghost variable introduction in the program $\langle \mathcal{N}, \mathcal{E}, G \rangle$. Let* annot *be a labeling from $\mathcal{N}$ to the domain of first order formulae. Let $[\![\mathsf{annot}]\!]_S$ be a labeling such that for every $l \in \mathcal{N}$ $[\![\mathsf{annot}]\!]_S(l) = [\![\mathsf{annot}(l)]\!]_{S(l)}$. Then, if* annot *is a solution over a* wp*-based verification framework for $\langle \mathcal{N}, \mathcal{E}, G \rangle$, then $[\![\mathsf{annot}]\!]_S$ is a solution for $\langle \mathcal{N}, \mathcal{E}, G' \rangle$.*

In other words, one can show that for every proof obligation in the original program

$$\mathsf{annot}(l) \Rightarrow \bigwedge_{(l,l') \in \mathcal{E}} \mathsf{wp}(G(l,l'))\,(\mathsf{annot}(l')),$$

we have the following proof obligation for the transformed program

$$[\![\mathsf{annot}(l)]\!]_{S(l)} \Rightarrow \bigwedge_{(l,l') \in \mathcal{E}} [\![\mathsf{wp}(G(l,l'))(\mathsf{annot}(l'))]\!]_{S(l)}.$$

That is, verification conditions differ only on a substitution of program variables by ghost variables, and thus certificates can be easily transformed.

One advantage of ghost variable introduction over existential quantification is a stronger final specification. Consider for instance a program P that satisfies the triple $\{x = y\}P\{x = y\}$, and suppose that the variable y is not modified in P. If y is considered dead, the transformed specification after ghost variable introduction is $\{x = y^g\}P\{x = y^g\}$, whereas the existential quantification of dead variables yields $\{\exists y.\ x = y\}P\{\exists y.\ x = y\}$. From the former, one can interpret that the original value of x is preserved by P, whereas in the latter this is not the case. On a negative side, however, ghost variable introduction requires the verification framework to deal with ghost variables in the specification.

6. HYBRID CERTIFICATES

Section 4 describes a general method to certify the results of program analysis using verification environments based on weakest preconditions or symbolic execution. An alternative is to develop hybrid verification methods that combine program analysis and deductive verification; such methods are common in modern verification tools [Chalin and James 2007; Barnett et al. 2005; Wildmoser et al. 2005]. They allow users to write weaker specifications, and generate smaller proof obligations.

In this section, we provide an abstract formalization of hybrid verification, and lift the notion of solution and certified solution to this setting. We then apply the results of the previous section to conclude that every hybrid certified solution can be transformed into a certified solution. A direct consequence is that the soundness of hybrid verification methods can be derived from the soundness of its components.

```
l1 : x := 0
     i := 0
l2 : while (i < |a|) do
       z := a[i]
l3 :   x := x + z
       i := i + 1
l4 : ...
     done
```

Fig. 26. Hybrid verification example. Source code.

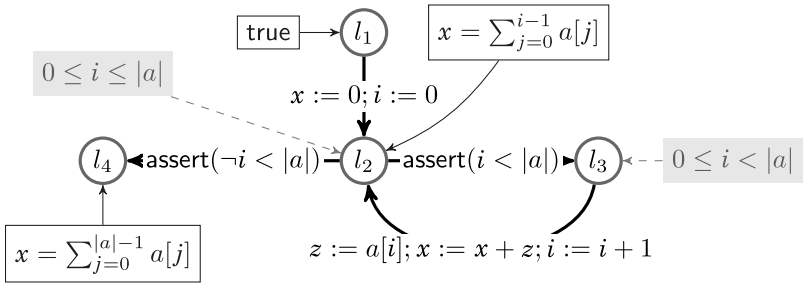

Fig. 27. Hybrid verification example. Graph representation.

6.1 Motivating Example

Consider the imperative program shown in Figure 26; its graph representation is given in Figure 27. In comparison with the example of Section 2, the program is written in a language that features constructs to store and load the values stored in an array. This extended language distinguishes between scalar and array variables and considers for each array a a new integer expression $|a|$ that denotes the size of a; in the sequel, we let a range over arrays. Moreover, the language features basic instructions $a[e_1] := e_2$ to store the value of e_2 at the position determined by e_1 in a and $x := a[e]$ that assigns to x the value stored in a at the position determined by e—it is customary to view $a[e]$ as an expression, but it is simpler for our presentation to consider an instruction that combines array access with assignment to a scalar variable.

The semantics of statements is extended as follows. First, we define the set Env of environments as $(\mathsf{Env}_s \times \mathsf{Env}_a) + \mathsf{Error}$ where Error is a distinguished constant denoting an error state, and Env_s is the set of mappings from variables to integer values, that is, $\mathsf{Env} = \mathsf{Var} \to \mathbb{Z}$, where Var is an infinite set of variables, and Env_a is the set of mappings from array variables and valid indexes to integer values, that is, $\mathsf{Env}_a = \Pi a \in \mathsf{AVar}.\{0 \ldots |a| - 1\} \to \mathbb{Z}$, where AVar is an infinite set of array variables. Then, the concrete semantics is extended for the assignment $x := a[e]$ by setting $(\eta, \eta') \in (\!|x := a[e]|\!)$ iff one of the following holds.

— $\eta = (\eta_s, \eta_a)$ and $\eta' = ([\eta_s : x \mapsto \eta_a\, a\, n], \eta_a)$, where e evaluates to n in η, such that $0 \le n < |a|$, and e' evaluates to n' in η, and $[\eta_s : x \mapsto n]$ is defined as in Section 3;
— $\eta = (\eta_s, \eta_a)$ and $\eta' = \mathsf{Error}$, where e evaluates to n in η, with $n < 0$ or $|a| \le n$;
— $\eta = \eta' = \mathsf{Error}$.

The semantics for the array assignment $a[e] = e'$ is defined in a similar fashion. Evaluation semantics is extended to error states: we say that a program P with initial memory η, evaluates to η', written $P, \eta \Downarrow \eta'$, iff $\langle l_{\mathsf{init}}, \eta\rangle \rightsquigarrow^\star \langle l_o, \eta'\rangle$ for some $l_o \in \mathcal{O}$, or $\eta' = \mathsf{Error}$ and $\langle l_{\mathsf{init}}, \eta\rangle \rightsquigarrow^\star \langle l'', \eta''\rangle$ for some $\eta'' \in \mathsf{Env}$, and $(\!|G(l'', l')|\!)\ \eta''\ \mathsf{Error}$ for some successor l' of l''.

In presence of error states, it is common to consider two postconditions, a postcondition for abnormal termination, as well as the postcondition for normal termination. For illustrative purposes, we implicitly assume that the postcondition for abnormal termination is false, which is the standard way to state that a program will not terminate abnormally. Under this convention, we consider triples of the form $\{\varphi\}P\{\psi\}$, whose validity is given by the clause $\models \{\varphi\}P\{\psi\}$ iff for all environments η and η', if $P, \eta \Downarrow \eta'$ and φ is satisfied by η then $\eta' \neq \mathsf{Error}$ and ψ is satisfied by η'.

Thus, a sound weakest precondition calculus must enforce the absence of out-of-bound array accesses; in other words, it must ensure for every assignment $x := a[e]$ that the variable e evaluates to a value within the bounds of the array a. We must therefore define $\mathsf{wp}(x := a[e])\ \psi$ to be $(0 \leq e < |a|) \wedge (\psi[^{x}/_{a[e]}])$.

Let us know turn to establishing that the program in Figure 27 is correct. We want to show that the final value of x is $\sum_{j=0}^{|a|-1} a[j]$. A partial labeling appears inside squares in Figure 27. There are two issues with this partial labeling.

The first issue is that it does not yield a solution: indeed, the loop invariant cannot be established unless we strengthen the loop invariant at node l_3 with the condition $0 \leq i < |a|$. In practice, a hybrid verification method would let users prove the program without having to provide in their annotations conjuncts that can be inferred by static analysis. In the first phase, a hybrid verification method relies on a static analysis that infers partial information from the program; in this case, the static analyzer infers the solution indicated inside a shadowed square in Figure 27—we omit the abstract labeling for nodes l_1 and l_4 that are set to $\top$. Then, a partial program specification can be used, as it is not necessary to include the redundant information computed in the first phase. However, the generation of verification conditions relies on the analysis results. In this case, the weakest precondition considers the conjunction of the assertion provided by the user and of the assertions inferred in the first phase. In practice, from a logical representation S of the analysis result, and a partial labeling $\underline{\mathsf{annot}}$, the strengthened specification maps each node $l \in \mathsf{dom}(\mathsf{annot})$ to the formula $\overline{\mathsf{annot}}(l) = \mathsf{annot}(l) \wedge S(l)$.

The second issue is the size and complexity of proof obligations in the presence of array accesses. Hybrid settings attempt to simplify the enforcement of inbound array accesses by removing them from the verification conditions whenever a static analyzer can discharge them automatically. In practice, a hybrid verification method would let users prove the program without having to prove the annotations that can be inferred by static analysis. In the first phase, the method will rely on a certifying analysis that will prove that some array accesses are safe. In the second phase, it will remove redundant fragments of the generated verification conditions. As a result, verification conditions become smaller and thus the verification effort is reduced. For example, we would define the weakest precondition as a function taking, in addition to the postcondition, the logical characterization r of the analysis at the program point considered, and set

$$\mathsf{wp}(x := a[e])\ \psi\ r = \begin{cases} (\psi[^{x}/_{a[e]}]) & \text{if } r \Rightarrow 0 \leq e < |a| \\ (0 \leq e < |a|) \wedge (\psi[^{x}/_{a[e]}]) & \text{otherwise.} \end{cases}$$

Note that one can use a decidable approximation of implication in the first condition.

These issues are handled by two technical artifacts: solutions modulo and hybrid abstract semantics. Solutions modulo characterize when a partial labeling can yield a solution using the results of a previous solution. In contrast, hybrid abstract semantics characterize when two solutions from distinct abstract semantics can be combined into a single solution.

6.2 Solution Modulo

This section formalizes the notion of solution modulo. Solutions modulo capture the idea that hybrid methods allow users to write specifications that yield solutions when combined with the result computed in the first phase, but that are not necessarily solutions on their own.

Definition 6.1 *Solution Modulo*. Let $\mathbf{A}$ be an abstract domain. Let $\langle [\![.]\!], f\rangle$ be an abstract semantics over $\mathbf{A}$. Let $P = \langle \mathcal{N}, \mathcal{E}, G\rangle$ be a program and let S be a solution for P. A labeling S' for P is a solution modulo S if

— $f = \mathsf{fwd}$ and for every $(l, l') \in \mathcal{E}$, $[\![G(l, l')]\!](S(l) \sqcap S'(l)) \sqsubseteq S'(l')$
— $f = \mathsf{bwd}$ and for every $(l, l') \in \mathcal{E}$, $S(l) \sqcap S'(l) \sqsubseteq [\![G(l, l')]\!](S'(l'))$.

Consider the program in Figure 27, and the labeling S' defined as:

$$S'(l_1) \doteq \mathsf{true}$$
$$S'(l_2) \doteq x = \textstyle\sum_{j=0}^{i-1} a[j]$$
$$S'(l_4) \doteq x = \textstyle\sum_{j=0}^{|a|-1} a[j].$$

The labeling S' does not satisfy the constraints required to be a solution over the wp-based framework.

One can see, however, that S' is a solution modulo S, where S is the labeling represented by a boxed annotation in the figure. For instance, one can check the validity of the constraint $S'(l_2) \sqcap S(l_2) \sqsubseteq \mathsf{wp}(G(l_2, l_4))\, S'(l_4)$ for the edge (l_2, l_4):

$$x = \sum_{j=0}^{i-1} a[j] \wedge 0 \leq i \leq |a| \Rightarrow \neg(i < |a|) \Rightarrow x = \sum_{j=0}^{|a|-1} a[j].$$

The following result states that the result of strengthening S' with S is a solution.

LEMMA 6.2. *Let S be a solution for a program $P = \langle \mathcal{N}, \mathcal{E}, G\rangle$. Suppose that, for every statement s, $[\![s]\!]$ is distributive with respect to $\sqcap$. If S' is a solution modulo S, then $S \sqcap S'$ is a solution.*

The definition of solution modulo can be easily adapted to accommodate certificates.

Definition 6.3. Let $\mathbf{A}$ be an abstract domain. Let $\langle [\![.]\!], f\rangle$ be an abstract semantics over $\mathbf{A}$. Let $P = \langle \mathcal{N}, \mathcal{E}, G\rangle$ be a program and let S be a solution.

A certified solution modulo S consists of a labeling S' and a family of certificates $(c_{(l,l')})_{(l,l')\in\mathcal{E}}$ such that for every $(l, l') \in \mathcal{E}$:

— if $f = \mathsf{bwd}$ then $c((l, l'))$ is a certificate for $S(l) \sqcap S'(l) \sqsubseteq [\![G(l, l')]\!](S'(l'))$
— if $f = \mathsf{fwd}$ then $c((l, l'))$ is a certificate for $[\![G(l, l')]\!](S(l) \sqcap S'(l)) \sqsubseteq S'(l')$

LEMMA 6.4. *Let $\mathbf{A}$ be an abstract domain. Let $\langle [\![.]\!], f\rangle$ be an abstract semantics over $\mathbf{A}$. Let $P = \langle \mathcal{N}, \mathcal{E}, G\rangle$ be a program and let $\langle S, c\rangle$ be a certified solution. Suppose that we have the certificates $\mathsf{distr}^{\leftarrow}_{([\![.]\!],\sqcap)}$ and $\mathsf{distr}^{\rightarrow}_{([\![.]\!],\sqcap)}$ shown in Figure 12. If a labeling S' is a certified solution modulo S, then one can build a certificate $c \oplus c'$ such that $\langle S \sqcap S', c \oplus c'\rangle$ is a certified solution.*

PROOF.

— Case $f = \mathsf{fwd}$. From the definition of certified solution modulo, we have that for every $(l, l') \in \mathcal{E}$, $c'((l, l'))$ is a certificate for $[\![G(l, l')]\!](S(l) \sqcap S'(l)) \sqsubseteq S'(l')$. The certificate

$c \oplus c'$ for the goal $[\![G(l,l')]\!](S(l) \sqcap S'(l)) \sqsubseteq S(l') \sqcap S'(l')$ is built by the following derivation steps:

$$\begin{aligned} p_1 &:= \mathsf{distr}^{\rightarrow}_{([\![.]\!],\sqcap)} : \mathcal{C}([\![G(l,l')]\!](S(l) \sqcap S'(l)) \sqsubseteq [\![G(l,l')]\!]\, S(l) \sqcap [\![G(l,l')]\!]\, S'(l)) \\ p_2 &:= \mathsf{weak}_{\sqcap}(c((l,l'))) : \mathcal{C}([\![G(l,l')]\!](S(l)) \sqcap [\![G(l,l')]\!](S'(l)) \sqsubseteq S(l')) \\ p_3 &:= \mathsf{trans}(p_1, p_2) : \mathcal{C}([\![G(l,l')]\!](S(l) \sqcap S'(l)) \sqsubseteq S(l')) \\ c \oplus c' &:= \mathsf{intro}_{\sqcap}(p_3, c'((l,l'))) : \mathcal{C}([\![G(l,l')]\!](S(l) \sqcap S'(l)) \sqsubseteq S(l') \sqcap S'(l')) \end{aligned}$$

— Case $f = \mathsf{bwd}$. From the definition of certified solution modulo, for every $(l,l') \in \mathcal{E}$, $c'((l,l'))$ is a certificate for $S(l) \sqcap S'(l) \sqsubseteq [\![G(l,l')]\!](S'(l'))$. The certificate $c \oplus c'$ for the goal $S(l) \sqcap S'(l) \sqsubseteq [\![G(l,l')]\!](S(l') \sqcap S'(l'))$ is built by the following derivation steps.

$$\begin{aligned} p_1 &:= \mathsf{distr}^{\leftarrow}_{([\![.]\!],\sqcap)} : \mathcal{C}([\![G(l,l')]\!]\, S(l') \sqcap [\![G(l,l')]\!]\, S'(l') \sqsubseteq [\![G(l,l')]\!](S(l') \sqcap S'(l'))) \\ p_2 &:= \mathsf{weak}_{\sqcap}(c((l,l'))) : \mathcal{C}(S(l') \sqcap S'(l') \sqsubseteq [\![G(l,l')]\!](S(l'))) \\ p_3 &:= \mathsf{intro}_{\sqcap}(p_2, '((l,l'))) : \mathcal{C}(S(l') \sqcap S'(l') \sqsubseteq [\![G(l,l')]\!]\, S(l') \sqcap [\![G(l,l')]\!]\, S'(l')) \\ c \oplus c' &:= \mathsf{trans}(p_3, p_1) : S(l') \sqcap S'(l') [\![G(l,l')]\!](S(l') \sqcap S'(l')) \end{aligned}$$

□

6.3 Hybrid Semantics

We now formalize hybrid verification environments that take advantage of static analysis results to simplify the verification conditions. This is done by considering a labeling and two abstract semantics. The first abstract semantics is sound w.r.t. the concrete semantics, the second one is defined with a simplified semantics function $[\![.]\!]_{hyb}$. The simplified abstract semantics is not necessarily sound, but can safely describe the concrete semantics if the labeling is correct.

Definition 6.5 *Hybrid semantics, sound hybrid semantics*. Let $\mathbf{A}$ be an abstract domain, and $\langle [\![.]\!], f \rangle$ be an abstract semantics over $\mathbf{A}$.

— A hybrid abstract semantics over $\mathbf{A}$ is a pair of the form $\langle [\![.]\!]_{hyb}, f_{hyb} \rangle$, where $[\![.]\!]_{hyb}$ has the type $\mathsf{Stmt} \to A \to A \to A$.
— A hybrid abstract semantics $\langle [\![.]\!]_{hyb}, f_{hyb} \rangle$ over A is sound w.r.t. $\langle [\![.]\!], f \rangle$ if for all statements $s \in \mathsf{Stmt}$ and $a, b \in A$,
  — if $f_{hyb} = \mathsf{bwd}$ then $b \sqcap [\![s]\!]_{hyb}\ b\ a \sqsubseteq [\![s]\!]\ a$;
  — if $f_{hyb} = \mathsf{fwd}$ then $[\![s]\!]\ a \sqsubseteq b \sqcap [\![s]\!]_{hyb}\ b\ a$.

Example 6.6. Consider the example of Figure 27. Assume the labeling S with $S(l_3) \doteq 0 \le i < |a|$ is a solution over an abstract semantics $\langle [\![.]\!], f \rangle$. Let wp_{hyb} a predicate transformer that ignores out-of-bound array accesses, that is, such that $\mathsf{wp}_{hyb}(z := a[i])\,(0 \le i < |a|)\,\varphi$ is defined as $\varphi[^{a[i]}/_{z}]$). Recall that $\mathsf{wp}(z := a[i])\,\varphi$ is defined as

$$0 \le i < |a| \wedge \Rightarrow \varphi[^{a[i]}/_{z}]$$

Then, wp_{hyb} is a sound hybrid semantics over wp since for every φ:

$$0 \le i < |a| \wedge \mathsf{wp}_{hyb}(z := a[i])(0 \le i < |a|)\ \varphi \sqsubseteq \mathsf{wp}(z := a[i])\ \varphi$$

The next definition characterizes hybrid solutions.

Definition 6.7. Let $\mathbf{A}$ be an abstract domain, let $\langle [\![.]\!], f \rangle$ be an abstract semantics over $\mathbf{A}$, and let $\langle [\![.]\!]_{hyb}, f_{hyb} \rangle$ be a hybrid abstract semantics that is sound w.r.t. $\langle [\![.]\!], f \rangle$.

A hybrid solution for a program $P = \langle \mathcal{N}, \mathcal{E}, G\rangle$ over the abstract semantics $\langle [\![.]\!], f\rangle$ and $\langle [\![.]\!]_{hyb}, f_{hyb}\rangle$ consists of a solution $S : \mathcal{N} \to A$ for P over $\langle [\![.]\!], f\rangle$, and of a labeling $S_{hyb} : \mathcal{N} \to A$ such that

— if $f = \mathsf{bwd}$, then for every l in $\mathcal{N}$, $S_{hyb}(l) \sqsubseteq \bigsqcap_{l':(l,l')\in\mathcal{E}} [\![G(l,l')]\!]_{hyb}\ S(l)\ S_{hyb}(l')$; or

— if $f = \mathsf{fwd}$, then for every node l' in $\mathcal{N}$, $\bigsqcup_{l:(l,l')\in\mathcal{E}} [\![G(l,l')]\!]_{hyb}\ S(l')\ S_{hyb}(l) \sqsubseteq S_{hyb}(l')$.

Let $\langle S, c\rangle$ be a static analysis result represented and certified over a sound verification framework I. Let I_{hyb} be a sound hybrid semantics w.r.t. I. Let $\langle S_{hyb}, c_{hyb}\rangle$ be a certified solution over the hybrid semantics I_{hyb}. In the rest of this section we show that it is possible, provided there is a formal proof of the soundness of I_{hyb} w.r.t. I, to build a certified solution $\langle S \sqcap S_{hyb}, c''\rangle$ over I.

The next lemma states that every hybrid solution can be turned into a solution.

LEMMA 6.8. *Let* $\mathbf{A}$ *be an abstract domain,* $I = \langle [\![.]\!], f\rangle$ *be an abstract semantics over* $\mathbf{A}$*, and let* $\langle [\![.]\!]_{hyb}, f_{hyb}\rangle$ *be a sound (w.r.t. I) hybrid abstract semantics. For every program P, and hybrid solution* $\langle S, S_{hyb}\rangle$ *for P, if* $S_{hyb} \sqsubseteq S$ *then* S_{hyb} *is a solution for P over I.*

The following result extends the lemma above over certified hybrid solutions.

THEOREM 1. *Let* $\mathbf{A}$ *be an abstract domain,* $I = \langle [\![.]\!], f\rangle$ *be an abstract semantics over* $\mathbf{A}$*, and* $\langle S, c\rangle$ *be a certified solution of I. Let* $I_{hyb} = \langle [\![.]\!]_{hyb}, f\rangle$ *be a hybrid abstract semantics.*

Assume that there is a certificate of the soundness of I_{hyb} *w.r.t. I, that is:*

— $f = \mathsf{bwd}$ *and for every statement s and* $a, b \in A$ *there is a certificate*

$$\mathsf{cert} : b \sqcap [\![s]\!]_{hyb}\ b\ a \sqsubseteq [\![s]\!]\ a \ \textit{ or}$$

— $f = \mathsf{fwd}$ *and for every statement s and* $a, b \in A$ *there is a certificate*

$$\mathsf{cert} : [\![s]\!]\ a \sqsubseteq b \sqcap [\![s]\!]_{hyb}\ b\ a.$$

Then, if $\langle S_{hyb}, c'\rangle$ *is a certified hybrid solution of* I_{hyb}*, and for every* $l \in \mathcal{N}$ *there is a certificate* $c_S : S_{hyb} \sqsubseteq S$*, then, there is a procedure to build a certificate* c''*, such that* $\langle S_{hyb}, c''\rangle$ *is a certified solution over I.*

PROOF. Consider the case $f = \mathsf{bwd}$. Let (l, l') be any edge in $\mathcal{E}$. The following is a derivation of a certificate for the goal $S_{hyb}(l) \sqsubseteq [\![G(l,l')]\!]\ S_{hyb}(l')$:

$$p_1{:=}\mathsf{intro}_{\sqcap}(c_S, c') : S_{hyb}(l) \sqsubseteq S(l) \sqcap [\![G(l,l')]\!]_{hyb}\ S(l)\ S_{hyb}(l')$$
$$p_2{:=}\mathsf{trans}(\mathsf{cert}((l,l')), p_1) : S_{hyb}(l) \sqsubseteq [\![G(l,l')]\!]\ S_{hyb}(l').$$

The case $f = \mathsf{fwd}$ is similar. □

Consider again the example in Figure 27. Assume that we have a certificate for the labeling S over the nonhybrid verification environment I, for instance, at node l_3 a certificate of

$$0 \le i < |a| \Rightarrow 0 \le i < |a| \wedge 0 \le i + 1 \le |a|.$$

Assume also that we have a certificate for the labeling S_{hyb} over the verification environment I_{hyb}, for instance, at node l_3 a certificate of

$$x = \sum_{j=0}^{i-1} a[j] \Rightarrow x + a[i] = \sum_{j=0}^{i+1-1} a[j].$$

Therefore, from the theorem above one can generate a certificate for the specification $S \sqcap S_{hyb}$ over the non-hybrid verification environment I, e.g., at node l_3 a certificate of

$$0 \leq i < |a| \wedge x = \sum_{j=0}^{i-1} a[j] \Rightarrow 0 \leq i < |a| \wedge 0 \leq i+1 \leq |a| x + a[i] = \sum_{j=0}^{i+1-1} a[j].$$

7. RELATED WORK

This section provides a brief overview of related work; see Barthe et al. [2009] for additional material. We begin with a review of Proof Carrying Code in Section 7.1. Section 7.2 discusses general methods for proving correctness of compilers and program optimizations, whereas Section 7.3 is concerned with certifying compilation, type-preserving compilation, and proof-preserving compilation. Section 7.4 focuses on certified solutions; finally, Section 7.5 considers hybrid certificates.

7.1 Proof Carrying Code

Proof Carrying Code [Necula and Lee 1996; Necula 1998] is a general framework to ensure security of mobile code through verifiable evidence. In its seminal form, the consumer side of a Proof Carrying Code architecture relies upon three main elements: logical assertions to express policies, a verification condition generator that extracts proof obligations from annotated programs, and a proof checker that verifies that the certificate establishes the desired proof obligations. Both the verification condition generator and the certificate checker form part of the Trusted Computing Base.

Foundational Proof Carrying Code [Appel 2001; Appel and Felten 2001; Wu et al. 2003] is an alternative approach that gives stronger semantic foundations to Proof Carrying Code. In this approach, the code producer gives a direct proof in higher-order logic that the code respects a given security policy. With this technique, the verification condition generator is removed entirely, and the Trusted Computing Base is minimalist. Reflective Proof Carrying Code [Barthe et al. 2008] is an alternative to Foundational Proof Carrying Code, in which an executable verification condition generator is implemented in higher-order logic, and verified. Compared to Foundational Proof Carrying Code, Reflective Proof Carrying Code yields more compact certificates. An early instance of verified verification condition generator appears in Wildmoser and Nipkow [2004, 2005].

While the original proposal for Proof Carrying Code relies on program logics as enabling technology, the most successful instance and widely deployed application of Proof Carrying Code technology to date, namely Java bytecode verification [Rose 2003], uses type systems as its enabling technology. In this setting, logical annotations are substituted by typing information, verification condition generation is substituted by constraint-based type checking, and certificates establish the validity of the constraints. Instances of type-based Proof Carrying Code include variants of lightweight bytecode verification, for instance, the lightweight information flow checker of Barthe et al. [2007]. As for Proof Carrying Code based on logic, some authors have taken a more foundational approach; in particular, there have been several instances

of certified static analyzers, in which the analyzer is implemented and verified in a higher-order logic [Cachera et al. 2004].

7.2 Certified Compilation and Translation Validation

Certificate translation is a general method to relate properties of two programs that are related by principled program transformations, for instance, optimization and compilation. It complements existing methods such as certified compilation and translation validation, and also subsumes other methods such as type-preserving compilation. The purpose of this section is to provide a brief comparison with these methods.

Certified compilation [Leroy 2006] is a general method to prove that the semantics of programs is preserved by compilation. Thus, instead of showing that the compiler preserves a specific property of a particular program, certified compilation establishes that the compiler preserves all properties for all programs.[1]

An alternative to certified compilation is to build an infrastructure for providing a rule-based description of program optimizations, and for proving the semantic correctness of these rules. This alternative, which has been explored, for instance, in Rhodium [Lerner et al. 2005], allows additional flexibility w.r.t. a certified compiler, especially in terms of the extensibility of correctness proofs.

Translation validation [Barrett et al. 2005; Pnueli et al. 1998; Zuck et al. 2002] can be seen as a specialization of certified compilation where semantics preservation is proved for individual programs, rather than for all programs. Translation validation establishes for specific programs that the compiler preserves all properties that are compatible with the statement of semantics preservation. Therefore, a certified compiler provides direct support for translation validation.

Translation validation can be used to yield the same guarantees as certificate translation. There are, however, some practical issues with achieving certificate translation via certified compilation or translation validation. Specifically, the resulting certificates include the proof of compiler correctness—for all programs in the case of certified compilation and for one program in the case of translation validation—and hence the definition of the compiler and the code of the source programs. Hence these methods are not appropriate when the size of the certificate is an issue or when the code producer wants to withhold the source code.

As a final note, observe that if the language of properties is sufficiently expressive, semantic preservation can be expressed by Hoare triples, and hence certifying compilation and certificate translation provide direct support for translation validation [Leroy 2006].

7.3 Certifying Compilation and Analysis; Type- and Proof-Preserving Compilation

The goal of certified compilation and translation validation is to establish a relationship between a source and target program. In contrast, certifying compilation [Necula and Lee 1998] aims to generate certificates of program correctness for target programs. Certifying compilation is the primary means to generate certificates in Proof Carrying Code architectures; while its scope was originally confined to safety properties, such as

[1]Strictly speaking, the guarantees only hold for properties that are compatible with the statement of semantics preservation. For example, if the statement of semantics preservation is based on an evaluation-semantics that capture the input/output behavior of programs, then certified compilation will show that all input/output properties of programs are preserved by compilation. An important topic in certified compilation is to strengthen the statement of semantics preservation, so that it also accounts for execution traces, or even more intensional properties of programs such as execution time or memory usage.

type safety and memory safety, recent works have built certifying compilers for other properties, including information flow [Beringer and Hofmann 2007].

Certifying analysis may be viewed as a variant of certifying compilation, where the compilation function is the identity. Previous works on certifying, or proof-producing, program analyses include [Chaieb 2006; Seo et al. 2003]. Seo et al. [2003] consider a generic backwards abstract interpretation for a simple imperative language and provide an algorithm that automatically constructs safety proofs in Hoare logic from abstract interpretation results. Chaieb [2006] considers a flow chart language equipped with a weakest precondition calculus, and provides sufficient conditions of the existence of certificates for solutions of backwards abstract interpretations; the case of Proposition 4.3 where $f = \mathsf{bwd}$ and $f^\sharp = \mathsf{fwd}$ recovers his result. Chaieb applies the algorithm to generate automatically proofs of safety of programs.

One issue with the automatic generation of proofs is the amount of irrelevant information they may contain: specifically, only a small fragment of the invariants inferred by automatic mechanisms may be relevant for a particular purpose, and it is therefore of interest to trim the proof so that it justifies a weaker specification than the one inferred automatically. In a follow up to Seo et al. [2003], Seo et al. [2007] develop slicing methods to remove unused parts of the specification, simplifying the next proof-producing phase.

While certifying compilation does not aim to provide an explicit relation between source and target programs, it is of interest to establish that sufficiently many source programs can be certified. Certificate translation provides a means of achieving this guarantee, by showing that the compiler transforms certified source programs into programs that can be certified at target level. Proof-transforming compilation is a particular instance of certificate translation in which the verification frameworks are program logics. It has been studied, for instance, in [Barthe et al. 2005; Saabas and Uustalu 2007] for core imperative languages, and in [Bannwart and Müller 2005; Barthe et al. 2008; Müller and Nordio 2007] for sequential Java. While some works [Barthe et al. 2005, 2008] study frameworks based on verification condition generators, other works [Bannwart and Müller 2005; Müller and Nordio 2007; Saabas and Uustalu 2007] consider Hoare logics for source and bytecode programs, and provide an algorithm to transform a Hoare proof of the original program to a Hoare proof of the transformed program. Most of these results on proof-transforming compilation focus on nonoptimizing compilers. In contrast, works like [Barthe et al. 2009; Saabas and Uustalu 2008] explicitly consider program optimizations. The work of Saabas and Uustalu is centered on casting program analyses and optimizations as type-systems; proof-transformation then follows by a constructive verification of the type system rules.

In order to address multiple policies, certifying compilation targets verification infrastructures based on program logics. An alternative is to target dedicated verification frameworks based on type systems and static analysis. An example of this technique is Java lightweight bytecode verification [Rose 2003], in which programs come equipped with partial type information that allows efficient type verification. Type-preserving compilation is the counterpart of proof-transforming compilation for type systems, see Chen et al. [2010] for a recent application to security, and for instance, Grossman and Morrisett [2000], Morrisett et al. [1999a; 1999b], Tarditi et al. [1996] for early works in type-directed compilation. In the setting of abstract interpretation, Rival [2003, 2004] proposes a method to translate analysis results along program compilation; result validation is restricted to post-fixpoint checking, that is, there is no notion of certificate. In contrast to these positive results, Logozzo and Fähndrich [2008] indicate that it may not be possible to preserve the results of an analysis, notably when considering numerical domains.

7.4 Certified Solutions

The definition of certified solution unifies ad hoc notions that have appeared in the literature. For example, Besson et al. [2007] propose a program analysis framework in which certificates are used to verify inclusions between elements of the abstract domain of polyhedra. The results of their analysis are instances of certified solutions. Lightweight bytecode verification methods [Rose 2003; Barthe et al. 2007] provide another instance of certified solutions.

Certified solutions are also closely related to Abstraction Carrying Code [Albert et al. 2005], a variant of Proof Carrying Code where programs come with a partial labeling that can be used by the code consumer to verify the program without having to approximate a fixpoint. In this scenario, the code consumer just needs to check that the partial labeling is a solution of the abstract interpretation and that it entails the desired behavior. Certified solutions allow to extend the scope of Abstraction Carrying Code to settings where the preorder relation is either undecidable, or too expensive for the code consumer to compute. In particular, Proof Carrying Code based on verification conditions can be seen as an instance of Abstraction Carrying Code with certified solutions.

7.5 Hybrid Certificates

Hybrid verification is heavily used, both for type based analyses and functional verification, and many concrete instances of hybrid verification methods appear in the literature and in verification tools. For example, many verification condition generators rely on a null pointer analysis to reduce the number of proof obligations; see, for instance, Barnett et al. [2005], and also Grégoire and Sacchini [2008], in which Grégoire and Sacchini prove the soundness of a hybrid verification condition generator that relies on a null pointer analysis, and Wildmoser et al. [2005], in which the authors prove the soundness of a hybrid verification condition generator that relies on an interval analysis.

Another example of hybrid method comes from information flow, in which preliminary analyses are fundamental for reducing the control flow of programs, and achieving better results; see, for instance, Myers [1999], and Barthe et al. [2007], where prove the soundness of an information flow type system that relies on many preliminary analyses.

8. CONCLUSION

We have provided a crisp formalization of certificate translation in a mild extension of abstract interpretation in which solutions carry a certificate of their correctness. Our formalization allows us to give a rational reconstruction of our earlier work, and to establish the scalability of certificate translation.

There are several additional benefits to our framework. It allows to derive the existence of certificate translators in settings that have not been considered before, for instance, concurrency [Kunz 2010]. Moreover, it provides leverage to prove the existence of certificate translators in more general settings, for instance, when the program analyses are justified by relational program logics. Finally, mild generalizations can be used to justify hybrid certificates, that combine simultaneously several verification methods.

Further work includes extending our results to relational program logics, and refine the results on concurrent programs from Kunz [2010] to verification methods that mitigate the explosion of verification conditions. Such verification methods should be expressible in our framework, using program skeletons to cluster atomically executable subsets of adjacent nodes into single nodes.

REFERENCES

ALBERT, E., PUEBLA, G., AND HERMENEGILDO, M. V. 2005. Abstraction-carrying code. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. F. and A. Voronkov Eds., Lecture Notes in Computer Science, vol. 3452, Springer-Verlag, 380–397.

APPEL, A. W. 2001. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*. J. Halpern Ed., IEEE Press, 247.

APPEL, A. W. AND FELTEN, E. W. 2001. Models for security policies in proof-carrying code. Tech. rep. TR-636-01, Princeton University.

BANNWART, F. Y. AND MÜLLER, P. 2005. A program logic for bytecode. *Electron. Notes Theor. Comput. Sci. 141*, 255–273.

BARNETT, M., LEINO, K. R. M., AND SCHULTE, W. 2005. The Spec# programming system: An overview. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS'04)*. G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean Eds., Lecture Notes in Computer Science Series, vol. 3362, Springer-Verlag, 151–171.

BARRETT, C. W., FANG, Y., GOLDBERG, B., HU, Y., PNUELI, A., AND ZUCK, L. D. 2005. Tvoc: A translation validator for optimizing compilers. In *Proceedings of the 17th International Conference on Computer-Aided Verification (CAV'05)*. K. Etessami and S. K. Rajamani Eds., Lecture Notes in Computer Science Series, vol. 3576, Springer-Verlag, 291–295.

BARTHE, G. AND KUNZ, C. 2008. Certificate translation in abstract interpretation. In *Proceedings of the European Symposium on Programming*. S. Drossopoulou Ed., Lecture Notes in Computer Science Series, vol. 4960, SpringerVerlag, 368–382.

BARTHE, G., REZK, T., AND SAABAS, A. 2005. Proof obligations preserving compilation. In *Proceedings of the Workshop on Formal Aspects in Security and Trust*. T. Dimitrakos, F. Martinelli, P. Ryan, and S. Schneider Eds., Lecture Notes in Computer Science, vol. 3866, Springer-Verlag, 112–126.

BARTHE, G., PICHARDIE, D., AND REZK, T. 2007. A certified lightweight non-interference Java bytecode verifier. In *Proceedings of the 16th European Symposium on Programming (ESOP'07)*. Lecture Notes in Computer Science, vol. 4421, Springer-Verlag, 125–140.

BARTHE, G., CRÉGUT, P., GRÉGOIRE, B., JENSEN, T., AND PICHARDIE, D. 2008. The MOBIUS proof carrying code infrastructure. In *Proceedings of the 6th International Symposium on Formal Methods for Components and Objects (FMCO'07)*. Springer-Verlag, 1–24.

BARTHE, G., GRÉGOIRE, B., AND PAVLOVA, M. 2008. Preservation of proof obligations from Java to the Java virtual machine. In *Proceedings of the International Joint Conference on Automated Reasoning*. A. Armando, P. Baumgartner, and G. Dowek Eds., Lecture Notes in Computer Science Series, vol. 5195, Springer, 83–99.

BARTHE, G., GRÉGOIRE, B., KUNZ, C., AND REZK, T. 2009. Certificate translation for optimizing compilers. *ACM Trans. Program. Lang. Syst. 31*, 5, 18:1–18:45.

BENTON, N. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. N. D. Jones and X. Leroy Eds., ACM Press, 14–25.

BERINGER, L. AND HOFMANN, M. 2007. Secure information flow and program logics. In *Proceedings of the IEEE Computer Security Foundations Symposium*. IEEE Press, 233–248.

BESSON, F., JENSEN, T., PICHARDIE, D., AND TURPIN, T. 2007. Result certification for relational program analysis. Resear. rep. 6333, IRISA.

CACHERA, D., JENSEN, T., PICHARDIE, D., AND RUSU, V. 2004. Extracting a data flow analyser in constructive logic. In *Proceedings of the European Symposium on Programming*. Lecture Notes in Computer Science Series, vol. 2986, 385–400.

CHAIEB, A. 2006. Proof-producing program analysis. In *Proceedings of the International Colloquium on Theoretical Aspects of Computing*. K. Barkaoui, A. Cavalcanti, and A. Cerone Eds., Lecture Notes in Computer Science Series, vol. 4281, Springer-Verlag, 287–301.

CHALIN, P. AND JAMES, P. R. 2007. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proceedings of the European Conference on Object-Oriented Programming*. 227–247.

CHEN, J., CHUGH, R., AND SWAMY, N. 2010. Type-preserving compilation of end-to-end verification of security enforcement. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation*. B. G. Zorn and A. Aiken Eds., ACM, 412–423.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 238–252.

COUSOT, P. AND COUSOT, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 269–282.

COUSOT, P. AND COUSOT, R. 1992. Abstract interpretation frameworks. *J. Log. Comput. 2*, 4, 511–547.

GRÉGOIRE, B. AND SACCHINI, J. 2008. Combining a verification condition generator for a bytecode language with static analyses. In *Proceedings of the 3rd Symposium on Trustworthy Global Computing: Revised Selected Papers*. Lecture Notes in Computer Science, vol. 4912, Springer-Verlag, 23–40.

GROSSMAN, D. AND MORRISETT, J. G. 2000. Scalable certification for typed assembly language. In *Proceedings of the 3rd International Workshop on Types in Compilation*. R. Harper Ed., Lecture Notes in Computer Science Series, vol. 2071, Springer, 117–146.

HANKIN, C., NIELSON, F., AND NIELSON, H. R. 2005. *Principles of Program Analysis* 2nd Ed. Springer-Verlag.

KUNZ, C. 2010. Certificate translation for the verification of concurrent programs. In *Proceedings of the Symposium on Trustworthy Global Computing*. M. Hofmann and M. Wirsing Eds., Lecture Notes in Computer Science Series, vol. 6084, Springer-Verlag.

LERNER, S., MILLSTEIN, T., RICE, E., AND CHAMBERS, C. 2005. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 364–377.

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

LOGOZZO, F. AND FÄHNDRICH, M. 2008. On the relative tompleteness of bytecode analysis versus source code analysis. In *Proceedings of the International Conference on Compiler Construction*. L. Hendren Ed., Lecture Notes in Computer Science Series, vol. 4959, Springer, 197–212.

MORRISETT, G., CRARY, K., GLEW, N., GROSSMAN, D., SAMUELS, R., SMITH, F., WALKER, D., WEIRICH, S., AND ZDANCEWIC, S. 1999a. Talx86: A realistic typed assembly language. In *Proceedings of the Workshop on Compiler Support for System Software*. 25–35.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1999b. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst. 21*, 3, 527–568.

MÜLLER, P. AND NORDIO, M. 2007. Proof-transforming compilation of programs with abrupt termination. Tech. rep. 565, ETH Zurich.

MYERS, A. C. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM Press, 228–241. (Ongoing development at http://www.cs.cornell.edu/jif/.)

NECULA, G. C. 1998. Compiling with proofs. Tech. rep. CMU-CS-98-154, Carnegie Mellon University.

NECULA, G. C. AND LEE, P. 1996. Safe kernel extensions without run-time checking. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation.* USENIX Assoc., Berkeley, CA, 229–243.

NECULA, G. C. AND LEE, P. 1998. The design and implementation ofa certifying compiler. In *Proceedings of the Conference on Programming Languages Design and Implementation*. ACM Press, New York, NY, 333–344.

PNUELI, A., SINGERMAN, E., AND SIEGEL, M. 1998. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. B. Steffen Ed., Lecture Notes in Computer Science Series, vol. 1384, Springer-Verlag, 151–166.

RIVAL, X. 2003. Abstract interpretation-based certification of assembly code. In *Proceedings of the 4th International Conference on Verification, Model Checking and Abstract Interpretation*. L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay Eds., Lecture Notes in Computer Science Series, vol. 2575, Springer-Verlag, 41–55.

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

ROSE, E. 2003. Lightweight byte code verification. *J. Automat. Reason. 31*, 3–4,303–334.

SAABAS, A. AND UUSTALU, T. 2007. Type systems for optimizing stack-based code. *Electronic Notes Theor. Comput. Sci. 190*, 1, Elsevier, 103–119.

SAABAS, A. AND UUSTALU, T. 2008. Proof optimization for partial redundancy elimination. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, 91–101.

SEO, S., YANG, H., AND YI, K. 2003. Automatic construction of Hoare proofs from abstract interpretation results. In *Proceedings of the Asian Programming Languages and Systems Symposium*. A. Ohori Ed., Lecture Notes in Computer Science, vol. 2895, Springer-Verlag, 230–245.

SEO, S., YANG, H., YI, K., AND HAN, T. 2007. Goal-directed weakening of abstract interpretation results. *ACM Trans. Program. Lang. Syst. 29*, 6, 39:1–39:39.

SØRENSEN, M. H. AND URZYCZYN, P. 2006. *Lectures on the Curry-Howard Isomorphism.* Elsevier.

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

WILDMOSER, M. AND NIPKOW, T. 2004. Certifying machine code safety: Shallow versus deep embedding. In *Proceedings of the 17th International Conference on Theorem Proving in Higher-Order Logics*. K. Slind, A. Bunker, and G. Gopalakrishnan Eds., Lecture Notes in Computer Science, vol. 3223, Springer-Verlag, 305–320.

WILDMOSER, M. AND NIPKOW, T. 2005. Asserting bytecode safety. In *Proceedings of the European Symposium on Programming*. M. Sagiv Ed., Lecture Notes in Computer Science, vol. 3444, Springer-Verlag, 326–341.

WILDMOSER, M., CHAIEB, A., AND NIPKOW, T. 2005. Bytecode analysis for proof carrying code. *Electron. Notes Theor. Comput. Sci. 141*, Elsevier.

WU, D., APPEL, A. W., AND STUMP, A. 2003. Foundational proof checkers with small witnesses. In *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM Press, New York, NY, 264–274.

ZUCK, L. D., PNUELI, A., FANG, Y., AND GOLDBERG, B. 2002. Voc: A translation validator for optimizing compilers. *Electron. Notes. Theor. Comput. Sci. 65*, 2.

Received January 2010; revised August 2010, December 2010; accepted February 2011